

# Глава 1.1. Что такое JavaScript

**JavaScript** – это язык сценариев. Что же такое язык сценариев? Это язык программирования, разработанный для воздействия на существующий объект или систему. В нашем случае объектом будет веб-страница. Получается JavaScript используется для создания и управления динамическим содержимым веб-сайта – всем, что перемещается, обновляется или иным образом изменяется на веб-странице без перезагрузки. Это и анимированная графика, и слайд-шоу фотографий, и автозаполнение текста, и интерактивные формы. Одним словом с помощью JavaScript мы можем запрограммировать поведение.

*Автодополнение поискового запроса на Яндексе возможно благодаря JavaScript*

Мы все привыкли, что лента «ВКонтакте» автоматически обновляется на экране без перезагрузки страницы, а поисковые сервисы показывают результаты поиска на основе нескольких введённых букв. Всё это возможно благодаря JavaScript.

JavaScript является неотъемлемой частью веб-функциональности, поэтому все основные веб-браузеры поставляются со встроенными механизмами, которые могут выполнять JavaScript. Это означает, что команды JavaScript можно вводить непосредственно в HTML-документ, и веб-браузеры смогут их понять. Другими словами, использование JavaScript не требует установки дополнительных программ или компиляторов.

Однако, браузеров существует множество, и JavaScript в каждом должен выполняться одинаково, другим словом «стандартно». Стандартом для реализации JavaScript в браузере является **ECMAScript** – это язык сценариев, разработанный в сотрудничестве с Netscape и Microsoft. Это именно стандарт, а JavaScript – реализация стандарта в вебе.

Наличие стандарта ECMAScript помогает обеспечить большую согласованность между реализациями JavaScript в разных браузерах. Сам ECMAScript является объектно-ориентированным и задуман как базовый язык, к которому могут быть добавлены объекты любой конкретной области или контекста. Про встроенные объекты мы ещё поговорим в будущем.

Чтобы проиллюстрировать пример «стандарта», вспомним об обычной клавиатуре компьютера, которую мы используем каждый день. У подавляющего большинства клавиатур буквы расположены в одном и том же порядке. Пробел, клавиша `Enter`, стрелки и цифровой блок также расположены приблизительно одинаково. Это связано с тем, что большинство производителей клавиатур опираются на стандарт раскладки QWERTY. Так и производители браузеров в реализации JavaScript опираются на стандарт ECMAScript.



Однаковая раскладка QWERTY на разных клавиатурах

Подробнее о том, что такое стандарт, какие у него бывают версии и чем стандарт отличается от реализации стандарта рассказываем в отдельной главе для дополнительного чтения:

– [Спецификация](#)

## Глава 1.2. Спецификация

**Спецификация** — это документация к языку программирования. Обычно это большой документ с подробнейшим описанием языка. В первую очередь спецификация необходима разработчикам языка программирования. По ней может совершенствоваться и дорабатываться язык.

Для прикладных разработчиков, например фронтендеров, спецификация также важна. В любом языке программирования всегда есть неочевидные вещи. Понять, почему они работают именно так — поможет спецификация. Как говорится, это последняя инстанция.

### ECMA-262

Язык JavaScript определяет [спецификация ECMA-262](#). По приведённой ссылке доступна как актуальная версия спецификации, так и архив предыдущих.

Спецификация ECMA-262 — это объёмный документ. Его не осилить ни за один, ни за два подхода. Читать спецификацию от корки до корки и не нужно. В этом документе собрана полная информация по JavaScript, и работать с ней нужно по частям, обращаясь, когда прикладная документация не даёт ответа.

Развитием спецификации ECMAScript занимается комитет TC39. Членами комитета являются крупные поставщики браузеров и другие компании. Несколько лет назад комитет TC39 перешёл на цикл ежегодного обновления спецификации ECMA-262. Каждый год спецификация дополняется и обновляется. Понять, что появится в очередной версии спецификации можно, заглянув в черновики. Последний черновик всегда доступен [по короткой ссылке](#).

У TC39 также есть отдельный [репозиторий на GitHub](#) с предложениями к спецификации. Здесь можно получить информацию о возможностях, которые планируется включить в спецификацию или которые были отклонены.

Очередная версия спецификации приносит в язык новые возможности. Однако, не все они сразу становятся доступны к применению. Производители браузеров сами определяют приоритет внедрения новых функций. Одни могут появиться раньше, чем выйдет спецификация. Другие наоборот, позже. Поэтому, прежде, чем использовать какую-нибудь возможность недавно вышедшей версии спецификации, убедитесь, что она поддерживается нужными вам браузерами.

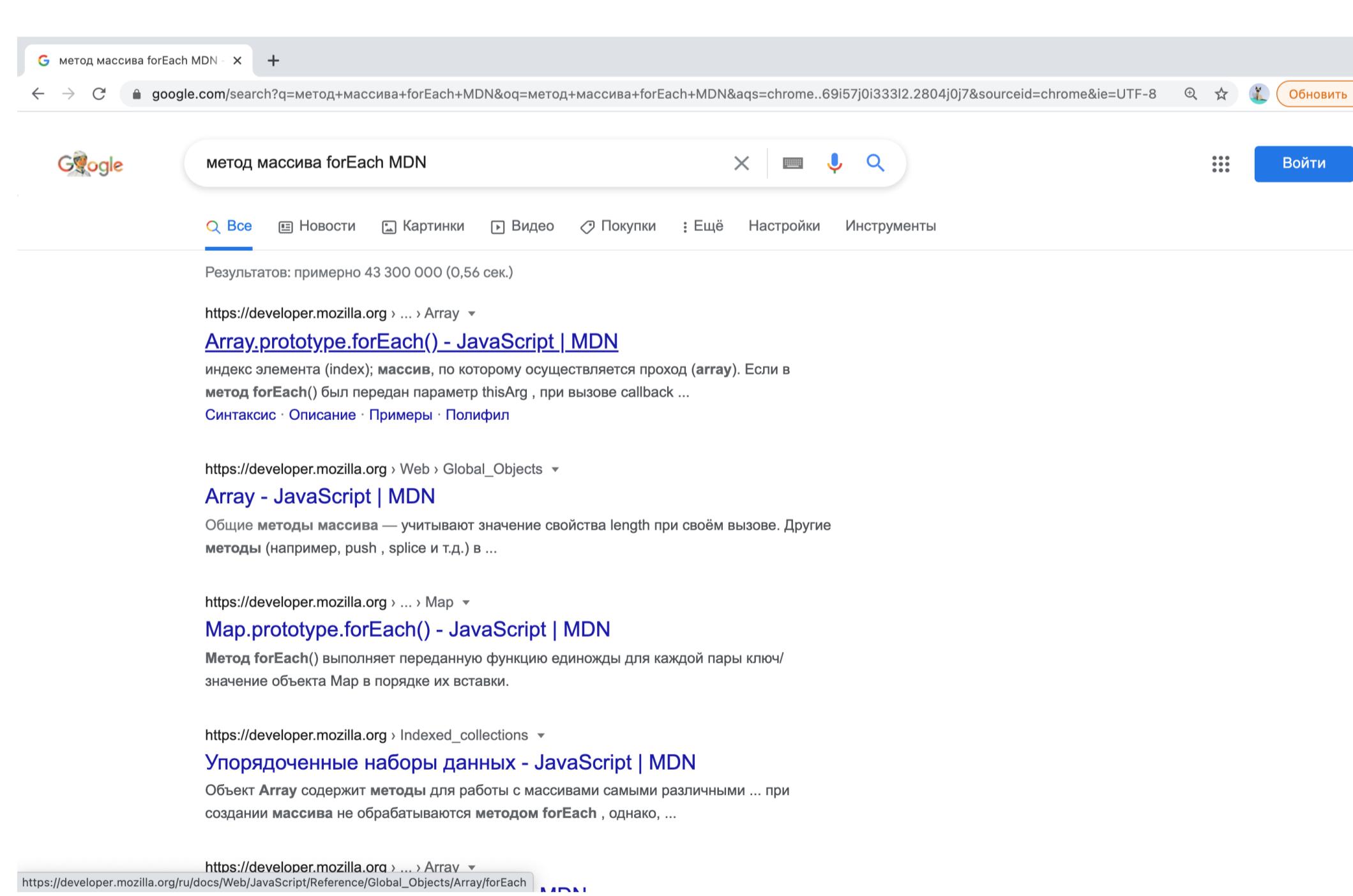
### DOM. Living Standard

В спецификации ECMA-262 приведена информация только касательно самого языка программирования. Сведений, которые относятся к применению JavaScript в браузере в ней нет. Они описаны в отдельном документе — [DOM. Living Standard](#).

### Где искать информацию

Во время изучения JavaScript перед вами постоянно будут возникать вопросы. Для поиска ответов во время обучения спецификация не очень подходит. В ней находится сухое описание без практических примеров. Найти примеры и ознакомиться с описанием той или иной возможности на начальных этапах удобнее на портале [MDN Web Docs](#), в народе просто MDN. Многие разделы переведены на русский язык!

Искать ответ на вопрос можно при помощи внутреннего поиска в MDN, но зачастую удобней пользоваться поисковым сервисом вроде [DuckDuckGo](#), [Google](#) или [Яндекс](#), добавляя в конце «MDN». Например, «метод массива forEach MDN».



Пример поисковой выдачи в Google

### Резюме

Спецификация — главный источник правды о языке программирования. В нём нет примеров кода, но детально описывается поведение всех возможностей языка. Изучать спецификацию непросто, но зачастую лишь она может пролить свет на сложный вопрос.

## Глава 1.3. Синтаксис

Как и в любом языке, не важно в языке программирования или в естественном языке, в JavaScript существуют языковые конструкции. Расположенные в определённой последовательности, как слова в предложении, эти конструкции составляют синтаксис языка. И расставляя в нужном порядке запятые, точки, точки с запятыми, скобки, знаки равенства и прочие знаки, а также некоторые из английских слов, разработчик пишет программу. Чем опытнее разработчик, тем больше языковых конструкций он знает и умеет использовать. Однако выучить их все не получится, потому что с каждым обновлением языка добавляются всё новые и новые элементы синтаксиса. Поэтому в случае с языком программирования работают все те же правила, что в случае с естественными языками: важно выучить базовый синтаксис – самые часто используемые конструкции – «на зубок», а после постепенно и постоянно расширять свой набор знаний.

### Синтаксис на практике

Рассмотрим синтаксис двух главных строительных блоков любой программы: переменной и функции.

#### Переменные

Чтобы объявить переменную в JavaScript, нужны:

1. ключевое слово `let` или `const`;
2. имя переменной (разработчик должен придумать его самостоятельно, фантазия ограничена латиницей и несколькими спецсимволами);
3. оператор присваивания `=` (знак равенства);
4. значение переменной (разработчик должен определить его самостоятельно).

Пункты 3 и 4 опциональны, потому что бывают случаи, когда значение переменной в момент объявления *ещё не известно*.

И тогда, чтобы объявить переменную с именем `company`, разработчик должен написать:

```
let company;
```

А чтобы объявить ту же переменную, но уже со значением, разработчик должен написать:

```
let company = 'HTML Academy';
```

Каждый знак в коде важен. Например `;` (точка с запятой) показывает, что языковая конструкция закончилась. Значит после можно начинать другую языковую конструкцию:

```
let company = 'HTML Academy';
let city = 'Санкт-Петербург';
```

Компьютеру без разницы, а вот человеку удобнее читать код с новой строки, поэтому обычно на одной строке располагается только одна языковая конструкция:

```
let company = 'HTML Academy';
let city = 'Санкт-Петербург';
```

А *ещё* надо помнить, что язык программирования – это не английский язык, хотя в JavaScript используются некоторые английские слова. Поэтому нельзя просто написать:

```
let company = HTML Academy;
```

JavaScript не знает слов `HTML` и `Academy`. Чтобы в JavaScript определить какую-либо текстовую информацию, нужно её представить в виде строки, для этого используются кавычки: одинарные `'` или двойные `"`. На каком языке будет сообщение в кавычках не имеет значения: хоть на английском, хоть на русском, хоть на JavaScript.

С числами проще. В JavaScript целые числа похожи на числа в обычном языке:

```
let est = 1703;
```

Дробные числа называются числами с плавающей точкой, потому что дробная часть отделяется `.` (точкой):

```
let pi = 3.14;
```

#### Функции

Чтобы объявить функцию в JavaScript, нужны:

1. ключевое слово `function`;
2. имя функции (разработчик должен придумать его самостоятельно, фантазия ограничена латиницей и несколькими спецсимволами);
3. пара круглых скобок `()`, внутри которых через запятую можно перечислить параметры функции, а можно не перечислять;
4. пара фигурных скобок `{}`, с помощью которых ограничиваются тело функции;
5. ключевое слово `return`.

Что такое параметры функции, зачем `return` и что значит «функция возвращает значение» вы узнаете из отдельного материала про функции, пока рассматриваем только синтаксис.

И тогда, чтобы объявить функцию с именем `getCompanyName`, разработчик должен написать:

```
function getCompanyName () { return 'HTML Academy' }
```

Кстати, после конструкций с телом, *вроде* функции, можно точку с запятой не ставить.

Опять же, человеку будет удобнее читать код с новой строки, поэтому обычно в теле функции также используют переносы строки и символ табуляции, чтобы вложенностью показать принадлежность:

```
function getCompanyName () {
    return 'HTML Academy';
}
```

Переносы строки не регламентированы JavaScript, поэтому некоторые разработчики предпочитают писать так:

```
function getCompanyName ()
{
    return 'HTML Academy';
}
```

В этом нет никакой ошибки, и всё же мы не рекомендуем использовать такой стиль.

Для объявления функции используют сразу два типа скобок – круглые и фигурные. Однако назначение тех или иных скобок зависит от места, где они используются. Например, круглые скобки *ещё* используются для вызова (выполнения) функции, если поставить их следом за именем функции:

```
getCompanyName();
```

А фигурные для обособления частей кода (редко):

```
{
    let company = 'HTML Academy';
}
```

Или для определения границ тела других конструкций *вроде* условий (часто):

```
if (2 > 1) {
    let company = 'HTML Academy';
}
```

Но если скобки *всё* же одни и те же, как понять, что перед нами за конструкция? *Эззубрить!* По контексту! Возьмём две похожие по порядку символов конструкции (функцию и условие):

```
function getCompanyName () {
    return 'HTML Academy';
}
```

```
if (2 > 1) {
    let company = 'HTML Academy';
}
```

Первое и главное отличие – разные ключевые слова `function` и `if`. Второе, конструкция условия не предполагает имени после `if` (и вообще). Третье, если у функции в круглых скобках перечисляются параметры, то в условии в круглых скобках указывается само условие. И так далее.

В любом похожем случае задача того или иного знака зависит от контекста использования, поэтому при написании JavaScript-кода нужно проявлять особое внимание, когда используете знаки равенства, кавычки, различные скобки и прочие символы.

Весь базовый синтаксис JavaScript мы будем изучать постепенно, чтобы в конце «пазл сложился», и вы смогли читать программу на JavaScript как обычную книгу. Поэтому если какие-то примеры из этой главы для вас пока непонятны, ничего страшного. Скоро мы это исправим.

## Глава 1.4. Подключение JavaScript к странице

Для добавления JavaScript-кода к странице применяется тег `<script>`. Он позволяет включить в страницу как самостоятельный блок кода, так и внешние файлы — сценарии с кодом на JavaScript.

### Внутри HTML-кода

Начнём с самого простого и наглядного варианта: размещение JavaScript-кода внутри тега `<script>`:

```
<!DOCTYPE html>
<html lang="ru">
<body>
<script>
    alert('Пауза...');

</script>
<p>Мороз и солнце; день чудесный!</p>
<p>Ещё ты дремлешь, друг прелестный</p>
</body>
</html>
```

Внутри тега `<script>` написана одна строка кода на JavaScript — вызов функции `alert`, которая отвечает за показ всплывающих сообщений.

Внутри тега `<script>` само собой может быть не одна строка кода, а много. В этом плане разработчик никак не ограничен. Разве что здравым смыслом, писать много кода внутри `<script>` не очень хорошая идея.

Перед тем, как перейти к альтернативным способам добавления JavaScript на страницу, давайте обсудим, как браузер будет выполнять этот код. Встретив в HTML-документе тег `<script>`, браузер приостановит разбор (или парсинг от англ. parse) документа. Вместо этого он займётся контентом тега `<script>` — разберёт и выполнит JavaScript-код. После этого продолжит вновь парсить документ.

Почему мы фокусируем на этом внимание? Пока браузер занимается разбором и выполнением JavaScript-кода, отрисовка страницы приостанавливается. В нашем примере пока пользователь не скроет всплывающее сообщение, стихотворение не будет отрисовано на странице:

В этом нет ничего страшного, если внутри тега `<script>` несложный фрагмент кода. Код выполнится молниеносно и отрисовка страницы продолжится. Визуально вы не заметите разницы. Однако, если код объёмный и сложный, вот здесь разница может стать заметной.

### Внешний сценарий

Способ с написанием кода прямо внутри тега `<script>` хорош, когда кода немного. При разработке полноценного фронтенд-приложения кода придётся написать больше, и удобнее всего это делать в отдельных файлах. А как же потом подключить эти файлы с кодом к странице? Для решения этой задачи опять же применяется тег `<script>`.

Для подключения к странице внешнего сценария следует задействовать атрибут `src`. Значением атрибута станет путь к внешнему файлу с JavaScript. Рассмотрим на примере:

```
<!-- Подключаем сценарий script.js, используя относительный путь -->
<script src="/path/script.js"></script>

<!-- Подключаем сценарий script.js, используя полный путь -->
<script src="https://myserver/js/script.js"></script>
```

В остальном этот способ похож на предыдущий. Браузер, встретив тег `<script>`, приостановит отрисовку страницы, пока внешний JavaScript-сценарий не будет загружен, разобран и исполнен.

А как быть если требуется подключить к странице сразу несколько сценариев? Точно так же. Каждый следующий сценарий подключаем при помощи тега `<script>`. Они загружаются и выполняются последовательно.

### Простая оптимизация

В обоих случаях возможна ситуация, когда разбор и выполнение сценария требует много времени. Что же, пользователю теперь придётся наблюдать частично отрисованное содержимое страницы или вообще белый пустой экран? Нет. К счастью есть несколько простых техник, позволяющие решить эту проблему. Наиболее простая из них заключается в подключении внешних сценариев в конце страницы, перед закрывающим тегом `</body>`. Таким образом, браузер отрисует содержимое страницы, а только потом возвратится за JavaScript.

### Внешние сценарии и код внутри `script`

Внимательные читатели наверняка задаются вопросом: «А что будет, если в атрибуте `src` мы укажем путь к внешнему сценарию, а затем ещё напишем код внутри самого тела `<script>`?»

```
<script src="https://myserver/js/script.js">
    alert('Привет, мир!');
</script>
```

Не будем ходить вокруг да около: если задан атрибут `src`, то содержимое тела `<script>` игнорируется. Сколько бы вы ни написали кода внутри тела `<script>`, выполнен он не будет.

## Глава 1.4.1. Атрибуты defer и async

У тега `<script>` есть два атрибута, позволяющие указать, как загружать и выполнять внешние сценарии. Речь об атрибутах `defer` и `async`. Применив любой из них, браузер не прервёт парсинг html-документа, когда доберётся до тега `<script>`. Он продолжит разбирать документ, а загрузка сценария начнётся параллельно, в фоне.

Таким образом, становится неважно в каком месте html-документа производится подключение внешних сценариев. Прерывания обработки документа не произойдёт, поэтому контент пользователь увидит сразу, не дожидаясь загрузки и выполнения внешнего сценария. Звучит здорово, но зачем тогда два атрибута?

### defer

Сценарии, подключаемые с применением атрибута `defer`, браузер загрузит в фоновом режиме, не прерывая отрисовку страницы. Выполнение таких сценариев произойдёт после завершения разбора html-документа. Причём неважно сколько сценариев подключается таким образом (с применением атрибута `defer`). Браузер дождётся завершения их загрузки и начнёт выполнять.

Рассмотрим на примере. Если в HTML-странице происходит подключение пяти сценариев с применением атрибута `defer`, то браузер дождётся их полной загрузки, а затем приступит к последовательному выполнению. Сценарии будут выполнены в порядке подключения к странице.

При использовании атрибута `defer` стоит помнить об ещё одном важном нюансе. После разбора html-документа, браузер генерирует событие `DOMContentLoaded`. Это означает, что браузер разобрал HTML и построил DOM-дерево, при этом некоторые внешние ресурсы (изображения, стили и так далее) могли ещё не успеть загрузиться. Событие `DOMContentLoaded` – своего рода сигнал для сценариев, которые опираются на DOM-элементы.

Так вот, если для загрузки сценариев применялся атрибут `defer`, то событие `DOMContentLoaded` генерируется только после окончания загрузки и выполнения всех таких сценариев. Об этом важно помнить.

Маленький вопрос для самопроверки понимания работы атрибута `defer`. Взгляните на код страницы. Попробуйте угадать, какой сценарий будет выполнен первым:

```
<!DOCTYPE html>
<html lang="ru">
  <head>
    <title>JavaScript. Профессиональная разработка веб-интерфейсов</title>
  </head>
  <body>
    <h1>Hello, JavaScript</h1>
    <h2>Второй заголовок</h2>

    <!-- Подключим библиотеку jQuery (размер 31.1 KB) -->
    <script defer src="https://code.jquery.com/jquery-3.5.1.min.js"></script>

    <!-- Подключим библиотеку lodash (размер 26 KB) -->
    <script defer src="https://cdn.jsdelivr.net/npm/lodash@4.17.20/lodash.min.js"></script>
  </body>
</html>
```

Если ваш ответ – `jQuery`, поздравляем! Вы ничего не упустили. Загрузка обоих сценариев произойдёт в фоновом режиме. Браузер загружает внешние ресурсы в несколько потоков. Скорей всего `lodash` загрузится раньше, так как он меньше по размеру. Несмотря на это, выполнение сценариев произойдёт последовательно – в порядке подключения. Атрибут `defer` это гарантирует.

### async

Второй атрибут, позволяющий изменить поведение загрузки внешних сценариев – `async`. Основный смысл остаётся неизменным: браузер, встретив тег `<script>` с атрибутом `async`, продолжит разбор HTML-документа и отрисовку страницы. Как и в случае с `defer`, сценарий с кодом загрузится в фоновом режиме. Чем же тогда отличается поведение `async` и `defer`?

Главное отличие кроется в независимости внешнего сценария по отношению к странице и другим сценариям, подключаемым с атрибутом `async`. Сценарии, подключаемые с применением атрибута `async`, загружаются в фоне и выполняются по мере готовности. Последовательность выполнения не гарантируется. Кто первый загрузился, тот и выполнится раньше. При этом браузер не будет дожидаться загрузки и выполнения таких сценариев, чтобы сгенерировать событие `DOMContentLoaded`. Поэтому сценарии с `async` могут выполниться как перед готовностью DOM, так и после.

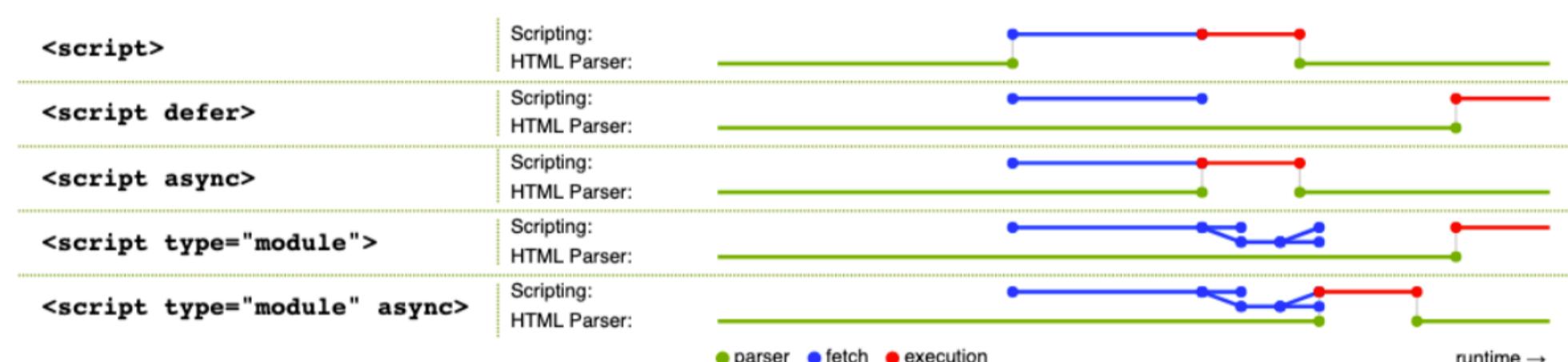


Диаграмма отличий между `async` и `defer`

### defer, async без src

Ещё один важный нюанс. Применять атрибуты `defer` и `async` имеет смысл только при подключении внешних сценариев. Если тег `<script>` используется без атрибута `src`, то атрибуты `defer` и `async` игнорируются. Поэтому, если воспользоваться одним из разобранных атрибутов и написать код внутри тега `<script>`, ожидаемого эффекта не будет.

### Когда и что использовать?

Когда внешний сценарий независим от выполнения других сценариев и не опирается на готовность DOM, имеет смысл воспользоваться атрибутом `async`. Примеры таких сценариев: всевозможные счётчики, внешние сервисы аналитики и так далее.

`defer` следует применять, когда важна последовательность выполнения сценариев и готовность DOM. Применение `defer` снимает ограничение на подключение внешних сценариев в самом конце страницы. Поскольку разбор html-документа не прерывается, подключение может производиться в любом месте страницы.

Если вам требуется подключить только один сценарий к странице и вы подключаете его в самом конце страницы, перед закрывающим тегом `</body>`, то атрибуты `defer` и `async` можно не применять.

### Резюме

Оптимизировать загрузку и выполнение кода помогут дополнительные атрибуты: `async` и `defer`. Первый удобен для подключения счётчиков, аналитики и других подобных сценариев. Таким сценариям, как правило, не важна готовность DOM и последовательность выполнения. `defer`, наоборот, полезен там, где важна последовательность.

## Глава 2.1. Переменные

Переменные позволяют хранить данные и использовать их в любом месте программы, обратившись по имени переменной. Переменную можно представить в виде подписанной коробки. В любую коробку можно что-то поместить, а затем из неё достать. Надпись на коробке (имя переменной) поможет найти нужную, если одинаковых коробок будет много.

Переменные объявляются по схеме:

```
ключевое_слово имя_переменной = значение_переменной
```

Давайте обявим переменную в JavaScript:

```
let nickname = 'Кекс';
```

Готово, мы только что обявили переменную `nickname` со значением `'Кекс'` с помощью ключевого слова `let`.

### Имя

У любой переменной должно быть имя, иначе мы не сможем к ней обратиться. Имя переменной может быть почти любым, но не должно начинаться с цифры, а из спецсимволов разрешены только `_` и `$`.

Правильные имена переменных:

```
let nickname = 'Кекс';
let first_man_in_space = 'Юрий Гагарин';
let $ = 'jQuery';
```

Имена переменных, которые вызовут ошибку:

```
let nick-name = 'Кекс';
let 1_man_in_space = 'Юрий Гагарин';
```

Кроме того, в JavaScript есть [зарезервированные слова](#), которые нельзя использовать для именования переменных. Иными словами, если обявить переменную с любым словом из этого списка в качестве имени, произойдёт ошибка.

Имена переменных чувствительны к регистру: `header`, `Header` и `HEADER` — это разные переменные. Но самое главное, чтобы переменная действительно делала код понятнее, её имя должно описывать то, что в ней хранится.

### Постоянные переменные

Кроме ключевого слова `let`, для обявления переменных в JavaScript используется ключевое слово `const`:

```
const nickname = 'Кекс';
```

Способы обявления переменной с помощью ключевых слов `let` и `const` равнозначны, но у `const` есть одно отличие. Возможно, вы уже догадались об этом по названию, `const` — это сокращение от слова `constant` (константа или постоянная, англ.) Переменную, обяленную с помощью `const`, нельзя перезаписать:

```
const x = 1;
x = 2; // Получим ошибку
```

В то время, как `let`-переменную перезаписать можно:

```
let x = 1;
x = 2; // Ошибки не будет, значение переменной x станет равно 2
```

### Постоянные значения

Прозвучит парадоксально, но не всякая `const`-переменная — это обязательно константа. Константами называются переменные, значение которых известно ещё до выполнения программы, и это значение не изменяется в ходе выполнения программы.

Обявление переменной с помощью `const` обеспечивает только второе из этих двух условий — неизменяемость. И чтобы в коде отличать константы от неизменяемых переменных, первые именуют в `CONSTANT_CASE` (когда все буквы заглавные, а слова разделяются подчёркиванием).

```
// Настоящая константа
const EARTH_RADIUS = 6371;

// Просто неизменяемая переменная
const randomNumber = Math.random();
```

### Какой способ обявления переменной выбрать

На начальных этапах, пока учитесь, всегда используйте `const`, и только если столкнётесь с необходимостью перезаписать значение переменной — используйте `let`. Это поможет избежать неявных, и часто случайных, перезаписей переменных, которые по неопытности трудно отловить и исправить. А когда наберёtesь опыта, уже решите для себя самостоятельно, какой вариант вам больше по душе, того и придерживайтесь.

## Глава 2.1. Переменные

Переменные позволяют хранить данные и использовать их в любом месте программы, обратившись по имени переменной. Переменную можно представить в виде подписанной коробки. В любую коробку можно что-то поместить, а затем из неё достать. Надпись на коробке (имя переменной) поможет найти нужную, если одинаковых коробок будет много.

Переменные объявляются по схеме:

```
ключевое_слово имя_переменной = значение_переменной
```

Давайте обявим переменную в JavaScript:

```
let nickname = 'Кекс';
```

Готово, мы только что обявили переменную `nickname` со значением `'Кекс'` с помощью ключевого слова `let`.

### Имя

У любой переменной должно быть имя, иначе мы не сможем к ней обратиться. Имя переменной может быть почти любым, но не должно начинаться с цифры, а из спецсимволов разрешены только `_` и `$`.

Правильные имена переменных:

```
let nickname = 'Кекс';
let first_man_in_space = 'Юрий Гагарин';
let $ = 'jQuery';
```

Имена переменных, которые вызовут ошибку:

```
let nick-name = 'Кекс';
let 1_man_in_space = 'Юрий Гагарин';
```

Кроме того, в JavaScript есть [зарезервированные слова](#), которые нельзя использовать для именования переменных. Иными словами, если обявить переменную с любым словом из этого списка в качестве имени, произойдёт ошибка.

Имена переменных чувствительны к регистру: `header`, `Header` и `HEADER` — это разные переменные. Но самое главное, чтобы переменная действительно делала код понятнее, её имя должно описывать то, что в ней хранится.

### Постоянные переменные

Кроме ключевого слова `let`, для обявления переменных в JavaScript используется ключевое слово `const`:

```
const nickname = 'Кекс';
```

Способы обявления переменной с помощью ключевых слов `let` и `const` равнозначны, но у `const` есть одно отличие. Возможно, вы уже догадались об этом по названию, `const` — это сокращение от слова `constant` (константа или постоянная, англ.) Переменную, обяленную с помощью `const`, нельзя перезаписать:

```
const x = 1;
x = 2; // Получим ошибку
```

В то время, как `let`-переменную перезаписать можно:

```
let x = 1;
x = 2; // Ошибки не будет, значение переменной x станет равно 2
```

### Постоянные значения

Прозвучит парадоксально, но не всякая `const`-переменная — это обязательно константа. Константами называются переменные, значение которых известно ещё до выполнения программы, и это значение не изменяется в ходе выполнения программы.

Обявление переменной с помощью `const` обеспечивает только второе из этих двух условий — неизменяемость. И чтобы в коде отличать константы от неизменяемых переменных, первые именуют в `CONSTANT_CASE` (когда все буквы заглавные, а слова разделяются подчёркиванием).

```
// Настоящая константа
const EARTH_RADIUS = 6371;

// Просто неизменяемая переменная
const randomNumber = Math.random();
```

### Какой способ обявления переменной выбрать

На начальных этапах, пока учите, всегда используйте `const`, и только если столкнётесь с необходимостью перезаписать значение переменной — используйте `let`. Это поможет избежать неявных, и часто случайных, перезаписей переменных, которые по неопытности трудно отловить и исправить. А когда наберётесь опыта, уже решите для себя самостоятельно, какой вариант вам больше по душе, того и придерживайтесь.

## Глава 2.2. Именование переменных и функций

Разработка — это процесс. Нельзя написать сценарий и просто забыть о нём. Однажды требования могут измениться, и в код понадобится внести правки. Лучше заранее позаботиться о том, чтобы код был максимально понятным. В этом разделе мы обсудим несколько правил написания понятного кода.

### Самодокументируемый код

В настоящее время IT-сообщество придерживается концепции **самодокументируемого кода**. Это значит, что в самом коде должно содержаться как можно больше информации о том, что он делает. И самый главный принцип самодокументируемого кода — правильное именование переменных.

Удачно подобранные имена переменной может рассказать многое. Неудачно подобранные — заставят лезть в документацию (если она есть), а то и вовсе запутает и заставит думать, что программа делает не то, что она делает, а нечто совершенно иное.

Называя переменную, следует как можно точнее описать её назначение. Однако не следует быть слишком многословным или избыточно точным:

```
// Плохо, название не описывает почти ничего
const s = 'Барсик';

// Плохо, указан тип, но не назначение
const string = 'Барсик';

// Уже лучше
const cat = 'Барсик';

// Совсем хорошо
const catName = 'Барсик'; // <<- Золотая середина

// Пойдёт
const myCatName = 'Барсик';

// Излишняя специфика
const barsikName = 'Барсик';

// Излишне многословно
const theNameOfSomeCat = 'Барсик';
```

### Почему английский?

Как латынь является профессиональным языком биологов и медиков, так и английский де facto стал языком программистов. Можно относиться к этому по-разному, однако факт остаётся фактом. Для именования переменных следует использовать слова на английском языке.

```
// Плохо
const VOZRAST = 18;

// Хорошо
const AGE = 18;
```

Если у вас не очень хорошо с английским, используйте автоматический переводчик. Поначалу это трудно, но со временем ваше знание языка само по себе улучшится.

### Стиль именования

Помимо того, что название должно быть правильно подобрано, оно должно быть правильно оформлено. Одно и то же название можно написать по-разному. В JavaScript принято использовать три стиля оформления:

- **camelCase** («верблюжий стиль» — слитно, каждое слово, кроме первого, начинается с большой буквы);
- **PascalCase** («стиль Паскаля» — как предыдущий, но первое слово тоже с большой буквы);
- **CONSTANT\_CASE** («стиль констант» — все буквы большие, слова разделяются символом подчёркивания).

Правильный стиль оформления несёт важную информацию о назначении переменной. Например, увидев `CONSTANT_CASE`, человек, читающий код, сразу понимает, что значение этой переменной не меняется и известно заранее. Поэтому, если, допустим, этот человек не просто читает код, а ищет в нём ошибку, он сразу знает, что ошибка возникла не из-за того, что эта переменная внезапно была изменена.

Верно и обратное: используя неправильный стиль оформления, можно запутать читателя, дать ему ложные предпосылки, которые приведут к неправильным выводам. Страйтесь избегать этого, если, конечно, ваша цель не промышленная диверсия.

#### CONSTANT\_CASE

Как нетрудно догадаться по названию, используется для именования констант — переменных, значение которых известно заранее и не изменяется в ходе выполнения программы:

```
// Плохо, константа не названа как константа
const earthRadius = 6371;

// Плохо, не константа названа как константа
const RANDOM_NUMBER = Math.random();

// Хорошо
const EARTH_RADIUS = 6371;
```

#### PascalCase

Используется для именования классов, конструкторов и перечислений. Классы и конструкторы — заслуживают отдельного разговора. Сейчас мы разбирать их не будем. А перечисление — это, грубо говоря, «сборник констант», которые объединены общей тематикой:

```
// Неправильно
const RAINBOW = {
  red: '#ff0000',
  orange: '#ffa500',
  yellow: '#ffff00',
  green: '#008000',
  lightBlue: '#4a2aff',
  blue: '#0000ff',
  indigo: '#4b0082',
};

// Правильно
const Rainbow = {
  RED: '#ff0000',
  ORANGE: '#ffa500',
  YELLOW: '#ffff00',
  GREEN: '#008000',
  LIGHTBLUE: '#4a2aff',
  BLUE: '#0000ff',
  INDIGO: '#4b0082',
};
```

О синтаксисе объектов `{}` и что это вообще такое, объекты, мы поговорим позже.

#### camelCase

Во всех остальных случаях используется обычный **camelCase**.

### Части речи

Ещё один важный способ дать подсказки человеку, читающему код — использование правильных частей речи. Для названий переменных, в которых содержатся значения примитивного типа или объекты, используйте существительные в единственном числе:

```
// Неправильно
const remember = 'Купить кошачьей еды';

// Опять неправильно
const tasks = 'Купить кошачьей еды';

// Правильно
const task = 'Купить кошачьей еды';
```

Множественное число тоже используется, когда речь идёт о наборе из нескольких значений. С примерами таких переменных мы познакомимся позже.

Функция отличается от других переменных тем, что её можно вызвать, и она произведёт какие-то действия. Чтобы отразить это, в названии функции обязательно используется глагол:

```
// Неправильно
function plus (a, b) {
  return a + b;
}

// Правильно
function summarize (a, b) {
  return a + b;
}
```

Впрочем, из этого правила есть некоторые исключения, вроде именования обработчиков событий по схеме `on` + событие, например `onClick`. С ними мы познакомимся позже.

#### Коллизии при именовании

В английском языке есть особенность, когда одно и то же слово может быть как глаголом, так и существительным. Такие слова в коде стоит использовать с осторожностью, потому что они сводят на нет различие между переменными и функциями. Например, без контекста трудно сказать, `filter` — это фильтровать (функция) или фильтр (переменная). Самое простое решение этой проблемы, использовать словосочетание. С уверенностью можно сказать, что `filterList` — это отфильтровать список (функция), а `currentFilter` — выбранный фильтр (переменная).

### Краткость

Порой возникает соблазн назвать переменную покороче, чтобы меньше пришлось печатать. Или назвать её одной буквой. Казалось бы, огромная экономия времени и клавиатуры. Однако это плохая идея, о которой впоследствии кто-нибудь жалеет.

Во-первых, сокращения быстрее пишутся, но медленнее читаются. Чтение кода, состоящего из каких-нибудь `smyrz` вместо `summarize`, удовольствие сильно ниже среднего. Во-вторых, в сокращениях легко запутаться. Уже через пять минут после того, как сокращённое название скрылось с экрана, становится трудно вспомнить: там было `smyrz`, или `smrz`, или вообще `sum`? В-третьих, сокращения не всегда можно однозначно восстановить.

Однобуквенные сокращения вводят все вышеперечисленные недостатки в куб. По таким именам нельзя понять, что хранится в переменной, не взглянув на её содержимое. Никогда не используйте сокращения для имён переменных и функций. Почти никогда. Исключение — общепринятые сокращения. Некоторые сокращения использовались настолько часто, что со временем из ошибки стали правилом, например:

- `evt` для объектов `Event` и его производных (`MouseEvent`, `KeyboardEvent` и подобные);
- `i`, `j`, `k`, `l`, `t` для счётчиков циклов и циклических методов;
- `cb` для единственного колбэка в параметрах функции.

И так далее. Почти со всеми сокращениями из «допустимого списка» мы познакомимся в течение курса.

## Глава 2.3. Типы данных в JavaScript

В JavaScript существуют два типа данных: примитивные и объектные. На сленге — простые и сложные. С их помощью описываются все данные, которые нужны для работы программ.

Стандарт ECMAScript определяет семь примитивных типов данных:

- строки `String`. В коде 'обрамляются' "различными" `кавычками`;
- числа `Number`, как целые 1 2 3, так и дробные 1.5 4.213 9.75 (последние правильнее называть «числа с плавающей точкой»);
- большие целые числа `BigInt`, помечаются буквой `n` на конце (9007199254740991n), используются для представления чисел, которые не может выразить `Number`;
- логический (булев) тип `Boolean` с двумя значениями `true` и `false`;
- `undefined`;
- `null`;
- символы `Symbol`.

И объектный тип данных, собственно, объекты (`{ name: 'Кекс' }`).

Чтобы определить тип данных, используется оператор `typeof`, который возвращает название типа на английском:

```
typeof 1; // вернёт "number"
typeof 'Привет, мир!'; // вернёт "string"
```

Про типы данных стоит запомнить, если не сказать зазубрить, несколько особенностей:

1. Хотя `null` — это примитив, выражение `typeof null` вернёт "object".
2. Оператор `typeof` для функции вернёт "function", но на самом деле функции — это те же объекты с одним отличием: их можно вызвать.
3. Числа и другие типы, представленные как строки, это строки:

```
typeof 1;      // вернёт "number"
typeof '1';    // вернёт "string"
typeof true;   // вернёт "boolean"
typeof 'true'; // вернёт "string"
// и т.д.
```

## Глава 2.3.1. Примитивы

В стандарт ECMAScript определены семь примитивных типов данных, разберём каждый отдельно.

### Строки

По определению строка – это последовательность символов произвольной длины. Любой символов: букв, цифр, пробелов, знаков препинания или даже вообще ничего, потому что пустая строка – тоже строка.

Чтобы строки можно было отличить от окружающего кода, они заключаются в кавычки: обычные, двойные или обратные. Выбор кавычек не влияет на значение строки и служит исключительно для удобства:

```
const singleQuote = 'Строка';
const doubleQuote = "Тоже строка";
const backtick = `Строка, хоть и непростая`;
```

«**Однако**, у строки, обрамлённой обратными кавычками, есть одна удобная особенность – интерполяция. Мы поговорим о ней позже.

Ещё один континтуитивный момент: строка может включать в себя символ переноса строки и таким образом фактически состоять из нескольких строк.

```
const multiline = 'Строка раз \n Строка два';
```

```
const multiline = 'Строка раз \n Строка два';
multiline
<- "Строка раз
Строка два"
> |
```

Выполняем код в консоли браузера

#### Перенос строки

Здесь `\n` – это не то, чем кажется. Выведя этот текст в консоль, мы не увидим косую черту и латинскую букву «\». Вместо них возникнет перенос строки. Сочетание символов `\n` – это одна из так называемых управляющих последовательностей. Такие последовательности начинаются с символа обратного слэша `\`, хотя и состоят из нескольких символов, «на выходе» дают один символ.

С помощью управляющих последовательностей можно кодировать символы, которые нельзя вставить просто так. Например, нельзя вставить двойную кавычку посреди строки, заключённой в двойные кавычки.

Код вызовет ошибку:

```
const error = "этот код даже " не запустится";
```

Ошибки не будет:

```
const success = "с этой строкой \" всё в порядке";
```

```
const success = "с этой строкой \" всё в порядке";
success
<- "с этой строкой \" всё в порядке"
> |
```

Выполняем код в консоли браузера

► А зачем это нужно?

### Числа

В отличие от многих других языков программирования, где существует множество числовых типов, в JavaScript есть только два типа для хранения чисел: `Number` для обычных целых и дробных чисел и `BigInt` для сверхбольших.

Можно записывать числа в различных системах счисления:

```
const decimal = 1234; // число 1234 в десятичной системе счисления
const hexadecimal = 0x4d2; // число 1234 в шестнадцатеричной системе счисления
```

В JavaScript это всё равно будет `Number`.

«**Если забыли**, что такое система счисления, освежить знания можно [на Википедии](#).

Можно записывать дробные (буквально «числа с плавающей точкой» от англ. float), положительные и отрицательные числа, даже заменять количество нулей на символ `e`:

```
const pi = 3.14; // число с плавающей точкой
const positive = 5; // положительное число
const negative = -5; // отрицательное число
const scientific = 1e6; // короткая запись 1000000
```

#### Бесконечность и «не число»

Кроме «привычных» нам чисел в JavaScript существуют специальные значения:

- `Infinity` – бесконечность;
- `-Infinity` – минус бесконечность;
- `NaN` – аббревиатура от not a number, буквально «не число».

С бесконечностью и минус бесконечностью всё должно быть интуитивно понятно, а вот `NaN` довольно континтуитивная вещь. В большинстве языков программирования попытка поделить на ноль или совершил какую-то другую запрещённую операцию приведёт к ошибке. В JavaScript ошибки не будет, вместо этого будет возвращено специальное значение – `NaN`.

Кроме того у `NaN` есть две особенности:

1. «не число» по типу является числом и `typeof NaN` вернёт `"number"`.
2. `NaN` – единственное значение в JavaScript, которое не равно вообще ничему, в том числе самому себе.

### Логический (булев) тип

Один из самых простых типов, в котором есть только два значения: `true` (истина) и `false` (ложь). Этот тип назван в честь английского математика Джорджа Буля, который изобрёл алгебру имени себя, где есть только эти два значения.

Несмотря на свою простоту, это самый важный и самый фундаментальный тип в программировании. Именно булевые значения используются в условиях вроде `if...else` – специальных конструкциях языка, для выполнения кода по условию.

### `undefined` и `null`

Ещё одна причина, по которой программисты на других языках недолюбливают JavaScript, целых два значения, означающих «ничего», и для каждого из них отведён свой отдельный тип.

Несмотря на то, что оба они символизируют отсутствие значения, между ними есть некоторая разница. `null` используют там, где хотят явным образом показать отсутствие значения. `undefined` возникает там, где значение не вернулось явно. Например, `undefined` – это значение переменной, которая изначально объявлена без значения:

```
let name; // в момент объявления значение переменной будет undefined
```

```
name = 'Keks'; // мы сменили значение с undefined на 'Keks'
```

Когда что использовать? Если вы сами присваиваете или передаёте значение и хотите показать его отсутствие, используйте `null`. Оставьте `undefined` служебным функциям JavaScript и используйте его только в проверках на то, что значение существует.

### Символ

Последний в нашем списке и самый новый из существующих примитивных типов данных – `Symbol`. Он был добавлен в стандарте ECMAScript 2015. В курсе этот тип не рассматривается и не применяется, поэтому за более подробной информацией [обращайтесь к MDN](#).

## Глава 2.3.2. Объектные типы данных

Значения, не являющиеся примитивными, называются **объектными** или **ссылочными**. К ним относятся объекты, функции, массивы и прочее. В отличие от примитивных значений, объекты могут иметь свойства. Доступ к свойствам можно получить по их именам через точку или через квадратные скобки. Значением свойства может быть что угодно, начиная от примитива и заканчивая ещё одним объектом.

```
const obj = {
  a: 1,
  b: 'какая-то строка'
};

console.log(obj.a); // 1
console.log(obj['b']); // 'какая-то строка'
```

Фундаментальное отличие ссылочных типов от примитивных можно проиллюстрировать следующим кодом:

```
let a = 1;
let b = a;
a = 2;

console.log(a); // 2
console.log(b); // 1
// От манипуляций с переменной a значение переменной b не изменится
```

```
let object = {a: 1};
let anotherObject = object;
object.a = 2;

console.log(object.a); // 2
console.log(anotherObject.a); // 2
// Мы изменили object.a, но изменилось ещё и anotherObject.a
```

И дело не в `let`, с `const` результат будет тот же:

```
const object = {a: 1};
const anotherObject = object;
object.a = 2;

console.log(object.a); // 2
console.log(anotherObject.a); // 2
```

Значения ссылочного типа — это не сами данные, а ссылка на область памяти в компьютере, где хранятся данные. Если мы скопируем эту ссылку, а затем изменим данные, используя исходную ссылку, то данные, на которые указывает ссылка-копия, также изменятся, поскольку это одни и те же данные!

Технически все значения ссылочного типа являются разновидностями объектов. Однако некоторые разновидности стоит рассмотреть отдельно. С точки зрения JavaScript, функция — это специальный вид объекта, который можно вызвать. Массив — это также особый вид объекта, оптимизированный для хранения серий значений по числовым индексам. Тем не менее и функцию, и массив при желании можно использовать как обычный объект. Например, создавать у них произвольные свойства.

## Глава 2.4. Сравнение сложных типов данных

В JavaScript значения могут быть двух типов: **примитивные** и **сложные**. К сложным относятся объекты, массивы и функции. Все они представляют тип `object`.

Также из тренажёров вы знаете, что сложные типы данных передаются по ссылке. Примитивные – по значению. Давайте на примерах разберём, что это значит.

### Передача по значению

```
let a = 5;
const b = a;
console.log('a =', a); // a = 5
console.log('b =', b); // b = 5
```

В данном примере мы создаём переменную `a` со значением 5. Это число, а значит примитив. Далее создаём переменную `b` и передаём ей значение переменной `a`. Получаем две совершенно независимые переменные, хотя их значения равны.

В том, что переменные никак не связаны, мы можем убедиться, изменив одну из них:

```
let a = 5;
const b = a;
a += 3;
console.log('a =', a); // a = 8
console.log('b =', b); // b = 5
```

Значение `b` осталось прежним, значение `a` – изменилось. Потому что в строке `const b = a;` число 5, примитив, передался по значению. Это и есть передача «по значению». Так происходит во всех примитивных типах.

### Передача по ссылке

Проведём аналогичный эксперимент с объектом.

```
const foo = {a: 1};
const bar = foo;
foo.a++;
console.log('foo.a =', foo.a); // foo.a = 2
console.log('bar.a =', bar.a); // bar.a = 2
```

В результате изменились значения обоих объектов, хотя мы изменияли только свойство объекта `foo`.

Всё дело в том, что сложный тип хранит в переменной не само значение (объект), а ссылку на него. И передаёт при присвоении не сам объект, а ссылку на него. То есть и `foo`, и `bar` в нашем примере ссылаются на один и тот же объект, который хранится в памяти компьютера. И если объект подвергся изменению, обе переменные покажут одинаковый результат. Это и есть передача «по ссылке». Так происходит во всех сложных типах.

## Сравнение

Сравнение примитивных типов происходит по значению:

```
const a = 10;
const b = 20;
console.log('a > b ? Ответ:', a > b); // a > b ? Ответ: false
```

Строки сравниваются посимвольно с учётом алфавитного порядка букв, пока не закончится одно из слов:

```
const str1 = 'Длиннийий';
const str2 = 'Длинний';
console.log('str1 > str2 ?', str1 > str2); // str1 > str2 ? true
const str3 = 'А';
const str4 = 'Б';
console.log('str3 > str4 ?', str3 > str4); // str3 > str4 ? false
const str5 = 'Солнце';
const str6 = 'Солнце';
console.log('str5 === str6 ?', str5 === str6); // str5 === str6 ? true
```

Если значения примитивов разных типов, то при сравнении JavaScript приводит их к единому:

```
const a = '111';
const b = 111;
console.log('a == b ?', a == b); // a == b ? true
```

В данном примере значение переменной `a` преобразуется из строки в число `111` и только потом сравнивается с `b`. Поэтому результат `true`.

Чтобы произвести сравнение без приведения типов, используйте оператор строгого равенства `==`:

```
const a = '111';
const b = 111;
console.log('a === b ?', a === b); // a === b ? false
```

Сравнение сложных типов происходит по ссылке, то есть **переменные сложных типов равны только в случае, если ссылаются на один и тот же объект в памяти компьютера**.

Операторы равенства `==` и строгого равенства `===` для сложных типов равнозначны. Рассмотрим примеры.

```
const foo = {a: 1};
const bar = foo;
console.log('foo == bar ?', foo == bar); // foo == bar ? true
```

Несмотря на то, что синтаксис присвоения объектов выглядит точно так же, как и у примитивов, при присвоении объектов `bar = foo` в `bar` передалась ссылка на объект `{a: 1}`, и теперь и `foo`, и `bar` ссылаются на один и тот же объект.

```
const foo = {a: 1};
const bar = {a: 1};
console.log('foo == bar ?', foo == bar); // foo == bar ? false
```

Здесь в переменных `bar` и `foo` лежат ссылки на объекты, у которых одинаковое содержимое, но для JavaScript это два разных объекта в памяти компьютера, а значит ссылки на них разные, поэтому эти объекты не равны. Рассмотрим ещё один пример:

```
const foo = {a: 1};
const bar = foo;
bar.a += 5;
console.log('foo.a < bar.a ?', foo.a < bar.a); // foo.a < bar.a ? false
console.log('foo.a == bar.a ?', foo.a == bar.a); // foo.a == bar.a ? true
```

При изменении свойства `a` – `bar.a += 5` – свойство изменяется у объекта в памяти компьютера, на который ссылаются и переменная `foo`, и переменная `bar`. Поэтому значение свойства изменится для обеих переменных, и `foo.a` будет равно `bar.a`.

### Как долго объект хранится в памяти

Жизненный цикл объекта в JavaScript продолжается от создания объекта до потери последней ссылки на него. То есть объект существует до тех пор, пока существует хотя бы одна переменная, которая на него ссылается. Как только объект теряет последнюю ссылку на себя, он подлежит утилизации.

Это связано с тем, что при жизни объекты занимают место в памяти. И если объект теряет все ссылки на себя, значит он не используется, и можно освободить память, которую занимает этот «ненужный» объект. Функцию утилизации «ненужных» объектов выполняет так называемый «сборщик мусора». В JavaScript процесс уборки спрятан под капотом и никак не управляется разработчиком.

Рассмотрим пример:

```
let foo = {a: 1};
let bar = foo;
foo = null;
bar = {b: 2};
bar = 25;
```

В переменную `foo` передали ссылку на объект `{a: 1}`. Далее в результате присвоения `bar = foo` в переменную `bar` передали ссылку на тот же объект.

Сейчас `foo` и `bar` ссылаются на один и тот же объект, другими словами, на объект `{a: 1}` ссылаются две переменные.

Если присвоить переменной `foo` значение `null`, то ссылка на объект удалится, и теперь на объект

ссылается только одна переменная `bar`.

Далее передаётся в переменную `bar` ссылку на другой объект – `{b: 2}`, соответственно, ссылка на объект

`{a: 1}` в переменной `bar` перезаписывается, и теперь на объект `{a: 1}` больше не существует

ни одной ссылки. Значит объект `{a: 1}` подлежит утилизации, и в какой-то момент придёт «сборщик мусора» и удалит его.

Если переменной `bar` присвоить не другой объект, а, например, примитив `25`, ссылка на объект `{b: 2}` также будет удалена из переменной `bar`. И теперь на объект `{b: 2}` нет ни одной ссылки, и он также подлежит утилизации.

Ситуация, когда объект не используется, но на него существуют ссылки, и он не может быть из-за этого утилизирован и продолжает занимать память, называется «утечкой памяти». Не допускать «утечек памяти» необходимо во время проектирования и разработки приложения.

## Глава 2.4.1. Динамическое приведение типов

У всех операторов в JavaScript есть нечто общее. Они имеют смысл лишь для определённых типов данных. Например, мы умеем складывать числа, но непонятно, как складывать, скажем, `null` и `undefined`. Мы можем применить оператор «логическое ИЛИ» к значениям типа булево, но что значит «Вася ИЛИ Петя»?

Что происходит, когда программист пытается применить оператор к данным неподходящего типа? Зависит от того, на каком языке он программирует. Одни языки моментально выдадут ему ошибку. Другие попытаются найти какой-то смысл в том, что он написал. Они попробуют преобразовать значения в такой тип, для которого этот оператор будет иметь смысл. Такое преобразование называется динамическим приведением типов.

Допустим, мы пытаемся разделить строку `'4'` на строку `'2'`. Деление имеет смысл только для чисел, поэтому JavaScript пытается превратить строки в числа. Он без труда догадывается, что строка `'4'` — символизирует число `4`, а строка `'2'` — число `2`. Поэтому в результате получится `4 / 2 = 2`.

Если мы попытаемся разделить строку `'Вася'` на строку `'Петя'`, JavaScript будет действовать аналогично. Разница в том, что ни `'Вася'`, ни `'Петя'` к числу разумным образом не приводится. Поэтому JavaScript превратит каждую из этих строк в специальное значение `NaN`. А «не число», делёное на «не число», в результате опять даёт «не число». Аналогично работают и другие типы операторов. Если логический оператор получает на вход строку, он пытается превратить её в `true` или `false`. И так далее, и тому подобное.

### Основные правила динамического приведения типов

Рассмотрим лишь частные случаи. Арифметические операторы всегда приводят свои operandы к числовому типу. Исключение — оператор `+`, из-за своего двойного назначения. Если хотя бы один из operandов — строка, то плюс считает, что он оператор конкатенации, и пытается привести другой operand тоже к строке.

```
'5' - '3'; // Получим число 2
'5' + '3'; // Получим строку '53'
```

Если нужно привести нечисловое значение к числовому типу, можно воспользоваться этой особенностью и применить к нему унарный оператор `+`. Применённый к числу он не делает ничего, однако применённый к чему-то другому, он, благодаря механизму динамического приведения типов, превратит это «что-то» в число:

```
(+'5') + (+'3') // Получим число 8
```

Как говорилось выше, плюс работает как оператор конкатенации, если хотя бы один operand — строка. Это можно использовать для приведения значений к строковому типу:

```
false + ''; // Получим строку 'false'
```

Логические операторы приводят свои operandы к булеву типу. Здесь, однако, важно понимать, что операторы «логического И» `&&`, и «логического ИЛИ» `||` — это не совсем то же самое, что логические операции «И» и «ИЛИ» из курса школьной информатики. Для примера распишем подробнее, как действует оператор `||`. Он берёт свой первый operand и приводит его к булеву типу. Если получится `true`, то оператор возвращает свой первый operand в исходном виде. Если нет — свой второй operand, опять же в исходном виде. Приведение к логическому типу используется только для выяснения, какой operand вернуть в качестве результата, но не применяется к самому результату:

```
'Вася' || 'Петя'; // 'Вася'
/*
  'Вася' приводится к true, потому что строка не пустая,
  и поэтому оператор || вернёт первый operand — 'Вася'.
  До Пети дело даже не доходит
*/
'' || 'Петя'; // 'Петя'
/*
  А здесь пустая строка приводится к false,
  и поэтому оператор || вернёт второй operand — 'Петя'.
*/
```

В отличие от логических операторов, оператор отрицания `!` возвращает значение булева типа. Чтобы привести произвольное значение к булеву типу, можно применить оператор отрицания дважды.

```
!!'Вася' // true
```

### Резюме

Если вам показалось, что динамическое приведение типов, это удобно... увы, это не так. Страйтесь как можно меньше использовать динамическое приведение типов.

Да, вы не ослышались! Это очень удобный механизм, позволяющий записывать коротко то, что в языке без него вышло бы многословно. Однако нужно понимать, что когда этот механизм проектировался, никто и не предполагал, какую роль займёт язык JavaScript десятки лет спустя. Изначально язык предназначался для написания простейших скриптов (вроде обработчика нажатия кнопки) людьми, далёкими от программирования. И он был спроектирован так, чтобы как можно меньше отпугивать их сложными концепциями. Сейчас, когда на JS пишутся полноценные приложения, былая простота зачастую оборачивается головной болью.

Выражения, использующие динамическое приведение типов, проще писать, но сложнее читать и искать в них ошибки. В сложных случаях лучше привести аргументы к нужным типам заранее. Динамическое приведение типов стоит использовать лишь в самых простых выражениях.

## Глава 2.5. Комментарии в коде

Любой язык программирования позволяет оставлять в коде комментарии. JavaScript не является исключением. **Комментарий** – вспомогательный текст, поясняющий работу исходного кода программы. Этот текст никак не влияет на интерпретирование и выполнение исходного кода программы.

Комментарии пишутся прямо в коде. Чтобы интерпретатор мог отличить их от основного кода, текст комментариев заключается между специальными символами. В каждом языке программирования приняты свои символы для обрамления текста комментариев. В JavaScript поддерживается два вида комментариев: односрочные и многострочные.

### Односрочные комментарии

Для односрочных комментариев применяется комбинация из двух символов слэш – `//`. Рассмотрим пример:

```
// Это односрочный комментарий
// Здесь может быть любой текст
// Его задача пояснить код. Например:
// Функция для приветствия
function greet () {
    return 'Привет, фронтендеры!';
}
```

Как мы уже отметили выше, комментарий – это просто вспомогательный текст. В примере выше представлено несколько односрочных комментариев. Текст пишется после двух символов `//`. Хорошим тоном делать после них пробел, а потом начинать писать текст комментария. Если текст не влезает в одну строку, то можно продолжить на следующей, но придется опять воспользоваться символами `//`.

### Многострочные комментарии

В дополнение к односрочным комментариям, JavaScript поддерживает многострочные. Их удобно применять, когда комментарий содержит несколько строк текста. Многострочные комментарии обрамляются символами `/* Текст комментария */`. Рассмотрим на примере.

```
/*
Это многострочный комментарий
Здесь может быть любой текст
Его задача пояснить код. Например:
Функция для приветствия.
*/
function greet () {
    return 'Привет, фронтендеры!';
}
```

При использовании многострочных комментариев важно не забыть закрыть блок с комментарием (`/* */`). Применение односрочных или многострочных комментариев зависит от объема пояснительного текста.

Обходимся одной строкой – односрочные, если нужно больше – многострочные.

Комментарии могут быть в любом месте исходного кода. На представленных примерах они описаны перед функциями, но в действительности могут быть и в теле функций. Одним словом – в любом месте исходного кода.

### Плохие комментарии

Может показаться, что хороший код должен быть прекрасно прокомментирован. На самом деле нет. Не стоит увлекаться написанием комментариев. Да, они никак не влияют на выполнение кода, но ими легко захламить код. Стоит увлечься, и рано или поздно с обильно прокомментированным кодом станет трудно работать.

Поэтому общий совет – не стоит увлекаться комментированием. Лучше сфокусироваться на читабельности кода. Если подобрать «говорящие» имена для переменных и функций, то такой код вряд ли потребует дополнительных пояснений. Для наглядности рассмотрим пример плохих комментариев:

```
// Функция для подсчета суммы a и b
const s = function (a, b) {
    // Считаем сумму a + b
    const c = a + b;

    // Возвращаем результат вычислений
    return c;
}
```

В примере приведена простая функция для суммирования параметров `a` и `b`. На первый взгляд комментарии в этом коде решают важную задачу: объясняют предназначение функции и подробно расписывают её работу. Однако на самом деле, комментарии маскируют проблему. Проблему плохого именования.

Имя функции состоит из одного символа. Если не заглянуть в тело функции, то её предназначение будет непонятно. Комментарии пытаются решить эту проблему за счет пояснительного текста, но по факту маскируют проблему. Лучше изменить наименование функции и сделать его более «говорящим». Тогда необходимость в комментарии отпадет:

```
const summarize = function (a, b) {
    return a + b;
}
```

### Хорошие комментарии

А какие же тогда комментарии принято считать хорошими? Ответ на этот вопрос прост – полезные. Запомните, комментарии пишутся для людей. Для автора кода и других разработчиков, которым рано или поздно придется работать с кодом. При написании комментария следует отталкиваться именно от пользы.

Со многими вещами проще разобраться, прочитав код. Если он написан хорошо и понятно, то необходимость комментирования в принципе отпадает. Однако, это ни в коем случае не говорит о бесполезности комментирования.

Первый пример полезных комментариев: причина выбора именно этого решения. Любую задачу можно решить несколькими способами. Комментарий может помочь объяснить выбор конкретного решения.

Такие комментарии пригодятся как самому разработчику, так и другим участникам команды. Вполне возможно, что до этого решения применялись другие, но данное решение позволило избежать неочевидных проблем.

Со временем вы сами можете вернуться к коду, прочитать этот комментарий, и в голову может прийти более удачное решение этой задачи. Комментарий поможет вспомнить, почему был выбран именно этот вариант.

Другой пример полезных комментариев – описание тонкостей. Даже если придерживаться правил хорошего именования, всегда есть вероятность хитрого кода – кода, прочитав который, будет непонятно, как он работает. В таких случаях комментарии тоже полезны. Однако стоит помнить: понятный код всегда лучше хитрого. Код в первую очередь для людей, а не для машин.

Есть еще один полезный вид комментариев – JSDoc. Современные редакторы/IDE поддерживают формат описания комментариев в формате [JSDoc](#). Этот формат вводит правила на описание (комментирование) функций/классов. Например, он позволяет задокументировать параметры (какие типы принимает функция), результат выполнения функции и так далее.

Стоп! А зачем это, если можно использовать правила хорошего именования? JSDoc – это больше, чем просто комментирование. Комментарии, написанные в таком формате, умеют разбирать редакторы кода. Обычно такая функциональность либо встроена, либо решается за счет установки расширения.

Если редактор обнаружит описание функции в JSDoc, то при обращении к этой функции, он выведет подсказку. Перечислит параметры, укажет их тип и даже добавит описание функции. Подобные подсказки вы наверняка видели во время применения встроенных функций.

Другая польза от JSDoc – возможность генерировать документацию. Специальные инструменты могут перебрать все модули проекта, вытащить из них комментарии в JSDoc и собрать красивую документацию.

### Горячие клавиши в редакторах

Практически все популярные редакторы кода поддерживают горячие клавиши для быстрого комментирования. Это бывает полезно, когда требуется закомментировать участок кода. Например, хочется проверить работу программы без нескольких строчек кода (возможно в них ошибка). Чтобы временно не удалять эти строчки, их можно закомментировать. Тогда они превратятся в «текст» и интерпретатор не выполнит их.

Эту задачу можно сделать руками – обрамить код либо в односрочный, либо в многострочный комментарий. Когда речь идет об одной строке – это сделать не сложно. Однако, если требуется закомментировать сразу блок кода, то удобнее воспользоваться горячими клавишами в редакторе кода.

В VSCode для этого предусмотрена комбинация горячих клавиш – `Ctrl + /` (или `CMD + /` если вы работаете в macOS). Попробуйте выделить участок кода и нажать эту комбинацию клавиш. Выбранный участок превратится в комментарий. Чтобы раскомментировать код воспользуйтесь этим же сочетанием клавиш (помните, код должен быть выделен).

### Резюме

Комментарии в коде полезны далеко не всегда. Если приходится писать длинные комментарии о работе кода, то скорее всего с кодом что-то не так. Возможно, такой код следует пересмотреть. Оставляйте комментарии для действительно сложных участков кода. Когда требуется понять, почему выбрано именно это решение, а не другое.

Комментарии полезны для описания выбранной архитектуры/решения и другой информации, которая поможет вам и другим разработчикам. Узнайте больше о формате JSDoc. Он позволит сделать работу с кодом более комфортной.

## Глава 2.6. Точка с запятой

Во многих языках программирования символ «точка с запятой» ; является обязательным. Его ставят после каждого выражения (операции), чтобы интерпретатор смог отделить одну инструкцию от другой. Встречаются и обратные ситуации: языки, в которых этот символ ставить необязательно вовсе.

JavaScript в отношении точки с запятой стоит где-то посередине. Спецификация предусматривает [автоматическую вставку точки с запятой](#), поэтому в большинстве случаев её можно не ставить, если инструкции отделяются переходом на новую строку. Переход на новую строку — сигнал для интерпретатора поставить неявную точку с запятой.

Однако есть ситуации, когда явный перенос строки JavaScript не сможет правильно интерпретировать и автоматически проставить точку с запятой. Результатом станет ошибка. Рассмотрим несколько примеров таких ситуаций:

```
const a = b + c
(d + e).toString()
```

Пусть каждая переменная содержит числовое значение. Попытка выполнить этот код приведёт к ошибке: `c` `is not a function`. Обратите внимание, перенос строки есть, но JavaScript не смог автоматически разделить инструкции.

Он посчитал, что на следующей строке происходит передача аргументов для функции `c`, но по задумке это обычная переменная с числовым значением. Конечно, одной ситуацией дело не ограничивается:

```
const text = 'Ещё одна курьёзная ситуация'
console.log(text)
[text, text].forEach(console.log)
```

Не обращайте внимание на последнюю строку кода в примере. В ней мы пытаемся дважды вывести содержимое переменной `text`. Как это работает — рассмотрим в одном из следующих разделов.

Как вы уже могли догадаться, при выполнении этого кода опять возникнет ошибка. Мы понадеялись на автоматическую вставку точки с запятой, но из-за третьей операции JavaScript принял неправильное решение. Третья инструкция стала частью второй из-за отсутствующей точки с запятой:

```
console.log(text)[text, text].forEach(console.log)
```

### Возможны и другие ситуации

Можно привести и ещё несколько примеров с демонстрацией неправильного разделения инструкций. В этом нет смысла. Спецификация [подробно описывает](#) правила автоматической простановки символа ;. Не нужно пытаться их все заучить наизусть. В большинстве ситуаций проще ставить точку с запятой явным образом.

### Резюме

Ставить точки с запятой или нет? Мнения сообщества JavaScript-разработчиков здесь разнятся. Есть приверженцы автоматической вставки, а есть те, кто предпочитает ставить самостоятельно. Мы рекомендуем использовать символ ; явно, то есть проставлять самостоятельно, разделяя инструкции.

## Глава 2.7. Функции

В любом языке программирования есть возможность разбить код на небольшие подпрограммы — самостоятельные фрагменты кода. Подпрограммы упрощают написание кода и позволяют переиспользовать его.

Упрощение достигается за счёт разделения кода на отдельные составляющие. Лучше всего это представить в виде фрагментов кода — каждый такой фрагмент решает маленькую задачу. Таким образом, разработчику не нужно держать в голове весь код программы. Он может фокусироваться на определённых фрагментах кода и не погружаться в детали других.

Для создания таких фрагментов кода, в языках программирования предусмотрены функции. **Функция** — это механизм, позволяющий разбивать код приложения на подпрограммы (фрагменты кода), тем самым давая возможность многократно переиспользовать его в разных частях программы.

Другая задача функций — упростить поддержку кода. Например, выделить сложный код в отдельный фрагмент, то есть функцию. Таким образом, код приложения разгружается, а функцию вы сможете вызвать в любом месте приложения.

Из определения выше, может сложиться впечатление, что функции применять не просто. На самом деле это не так. С точки зрения кода, функция не что иное как именованный фрагмент кода, обёрнутый оператором или специальными символами. Для выполнения кода функции применяется идентификатор — то самое имя.

### Объявление функций

Объявить функцию в JavaScript можно несколькими способами: декларативно (function declaration) и в виде функционального выражения (function expression). Спецификация ECMAScript 2015 добавила дополнительный синтаксис для описания функций — стрелочные функции. Про них мы поговорим в отдельном разделе, а сейчас рассмотрим первые два способа.

#### Декларативное объявление

При декларативном способе объявления функция определяется в основном потоке кода. Для объявления функции используется ключевое слово `function`. После него следует имя функции. Имя функции или идентификатор, позволят взаимодействовать с ней.

Как и в случае с переменными, имя функции не может содержать пробелов, начинаться с числа и включать спецсимволы, кроме `_` и `$`.

В качестве имени функций следует выбирать осмысленные слова. Не стоит создавать функции, имена которых состоят из одного символа. При взгляде на имя функции, должно быть сразу понятно её предназначение.

После имени функции в круглых скобках описываются параметры. **Параметры функции** — это входные данные. Проще говоря, значения, которые могут быть использованы внутри функций. Параметры передаются в функцию аргументами при вызове, об этом дальше. Для каждого такого значения в теле функции будет доступна переменная.

```
function имя_функции ([параметры_функции]) {
  [тело_функции]
}
```

Функция может не принимать никаких аргументов, быть самостоятельной, а значит параметры у неё будут отсутствовать. В этом случае следует указать пустые круглые скобки:

```
function имя_функции () {
  [тело_функции]
}
```

Почему функция не принимает «аргументы», а отсутствуют «параметры»? Дело в том, что параметрами называются значения, которые мы задаём в момент объявления функции. Их же мы используем и в теле функции. А аргументами называют значения, которые мы передаём в функцию при её вызове.

После круглых скобок открываются фигурные и в них описывается тело функции — тот самый фрагмент кода. Следуя структуре, попробуем описать первую функцию. Напишем функцию, которая умеет складывать два числа. Назовём эту функцию `summarize`:

```
function summarize (firstNumber, secondNumber) {
  return firstNumber + secondNumber;
}
```

У функции `summarize` объявлено два параметра: `firstNumber` и `secondNumber`. А значит при вызове функции мы можем аргументами передать числа, которые должны быть суммированы:

```
summarize(1, 2);
```

Внутри функции мы обращаемся к ним как к обычным переменным.

Следом за параметрами идёт описание тела функции. Тело функции обрамляется фигурными скобками. В теле описываются действия (код), которые должна выполнять функция.

В примере таким действием является сложение, но, само собой, действий может быть значительно больше. Всё зависит от задачи, которую решает функция.

Функция может возвращать результат своей работы. Применительно к примеру выше, результатом является сумма чисел. Для возврата значения из функции в языке предусмотрен оператор `return`. Выполнив оператор `return`, функция прекратит работу и вернёт значение.

Оператор `return` необязателен. Если в теле функции не указать возвращаемый результат с помощью оператора `return`, то JavaScript по умолчанию будет считать, что функция вернула `undefined`. Поэтому оба примера ниже равнозначны:

```
function summarize (firstNumber, secondNumber) {
  firstNumber + secondNumber;
  return undefined;
}
```

```
function summarize (firstNumber, secondNumber) {
  firstNumber + secondNumber;
}
```

### Функциональное выражение

При объявлении функции в виде функционального выражения результат определения функции записывается в переменную, а имя функции можно не указывать.

```
const имя_переменной = function ([параметры_функции]) {
  [тело_функции]
}
```

В остальном синтаксис идентичен. Рассмотрим на примере:

```
const deduct = function (firstNumber, secondNumber) {
  return firstNumber - secondNumber;
}
```

Мы определяем переменную `deduct` с анонимной функцией в качестве её значения.

Но как вызвать анонимную функцию, то есть функцию без имени? По имени переменной, значением которой она является:

```
deduct(2, 1);
```

Для простоты общения дальше по тексту мы будем использовать словосочетание «имя функции» вместо «имя переменной» для функций, объявленных как функциональное выражение.

### Вызов функции

Чтобы выполнить код внутри функции, её необходимо выполнить — вызвать. Для этого достаточно поставить круглые скобки после имени функции, открывающую и закрывающую, и функция будет вызвана:

```
summarize();
```

Если функция принимает аргументы, то в этих скобках следует передать значения для каждого из них:

```
summarize(1, 2);
```

### Результат выполнения функции

Если функция что-то возвращает (в её теле объявлен `return`), то результат её выполнения следует куда-то сохранить, чтобы потом им воспользоваться. Для этого достаточно записать вызов функции в переменную:

```
const total = summarize(1, 2);
const diminution = deduct(2, 1);
```

### Резюме

Функции позволяют разбить код на небольшие «подпрограммы» и многократно переиспользовать их из разных мест приложения. Объявить функции в JavaScript можно несколькими способами: в виде функционального выражения и декларативно. У обоих подходов есть свои преимущества и недостатки, однако на курсе они нивелируются, поэтому выбирайте любой из подходов. Главное придерживаться его во всём проекте!

## Глава 2.7.1. rest-параметры функций

Чаще всего при написании или работе с функциями в JavaScript у нас есть конкретное число аргументов, которые эти функции принимают. Однако существуют функции, которые работают с любым количеством аргументов. Например, функция `Math.max()` для определения максимального значения среди переданных аргументов. Ей неважно, среди какого количества элементов определять максимальный:

```
Math.max(1); // вернёт: 1
Math.max(1, 2); // вернёт: 2
Math.max(3, 2, 1); // вернёт: 3
Math.max(Infinity, 1, 2, 3, 4, 5, 100, 1000); // вернёт: Infinity
```

Но как реализовать такую функцию? Заложить сто параметров? Тысячу? А потом? Надеяться, что разработчик при вызове не передаст больше? Конечно, нет! Для случаев вроде этого, когда количество параметров заранее неизвестно, и существуют **rest-параметры**.

« Понять **rest-параметры** может быть немного проще, если перевести слово **rest** на русский. **Rest** — «оставшиеся» или «остальные», то есть получится «оставшиеся параметры» или «остальные параметры». Буквально — это возможность собрать все параметры функции вместе, сколько бы при вызове не было передано аргументов.

Для описания **rest-параметров** используется специальный оператор, выглядит как многоточие `...`, следом за которым без пробелов идёт название переменной, в которую нужно записать все параметры функции. Это значит, что все аргументы, которые будут переданы при вызове, будут собраны в переменную, и обратиться к ней можно будет по заданному вами имени.

```
function имя_функции (...rest_параметры) {}
```

Разберём пример:

```
function getMaxValue (...values) {
    // В теле функции мы можем обращаться к переменной values,
    // как к обычному параметру, только в этой переменной будут
    // собраны все переданные аргументы

    // Дальше дело техники. Пока не прочитали параметры,
    // берём за максимальное значение минимально возможное число —
    // минус бесконечность
    let max = -Infinity;

    // Затем в цикле, пока не кончатся параметры...
    for (let i = 0; i < values.length; i++) {
        // Проверяем каждый параметр: не больше ли он максимального...
        if (values[i] > max) {
            // Если больше, то считаем за максимальный его
            max = values[i];
        }
    }

    // После цикла возвращаем из функции самый максимальный параметр
    return max;
}

getMaxValue(3, 2, 1); // Для вызова вроде этого переменная values будет содержать [3, 2, 1]
```

Если вы не до конца понимаете код функции (например, что значит запись `values[i]`) — ничего страшного. Просто вернитесь к этому примеру после раздела о массивах.

Не обязательно превращать все параметры функции в «оставшиеся». Можно определить нужное количество параметров отдельными переменными, а остальные как **rest-параметры**:

```
function имя_функции (первый_параметр, второй_параметр, ...rest_параметры) {}
```

Например:

```
function getMaxValue (a, b, ...values) {
    // Первые два аргумента при вызове станут
    // параметрами a и b соответственно
    // Все другие аргументы попадут в переменную values

    // ...
}

getMaxValue(3, 2, 1); // Для вызова вроде этого переменная values будет содержать [1], а 3 и 2
// станут a и b соответственно
```

Переменную с оставшимися параметрами часто так и называют `rest`, однако на курсе такое имя нарушит критерий наименование переменных.

Однако, есть один нюанс: **rest-параметры** всегда должны идти последними и могут быть только одни. Впрочем, это следует даже из названия «остальные» или «оставшиеся». Поэтому объявить функцию с **rest-параметрами** посередине не выйдет:

```
function getMaxValue (a, b, ...values, c) {
    // ...
}
```

JavaScript сразу покажет ошибку `SyntaxError: parameter after rest parameter`.

### Оператор **rest**

В интернете вы можете встретить понятие «оператор **rest**» или «**rest-оператор**». Дело в том, что подход с **rest-параметрами** оказался настолько удобным, что его стали использовать не только для параметров функции, но и в других синтаксических конструкциях JavaScript. О таком использовании **rest-параметров** мы поговорим позже.

## Глава 2.7.1. rest-параметры функций

Чаще всего при написании или работе с функциями в JavaScript у нас есть конкретное число аргументов, которые эти функции принимают. Однако существуют функции, которые работают с любым количеством аргументов. Например, функция `Math.max()` для определения максимального значения среди переданных аргументов. Ей неважно, среди какого количества элементов определять максимальный:

```
Math.max(1); // вернёт: 1
Math.max(1, 2); // вернёт: 2
Math.max(3, 2, 1); // вернёт: 3
Math.max(Infinity, 1, 2, 3, 4, 5, 100, 1000); // вернёт: Infinity
```

Но как реализовать такую функцию? Заложить сто параметров? Тысячу? А потом? Надеяться, что разработчик при вызове не передаст больше? Конечно, нет! Для случаев вроде этого, когда количество параметров заранее неизвестно, и существуют **rest-параметры**.

« Понять **rest-параметры** может быть немного проще, если перевести слово **rest** на русский. **Rest** — «оставшиеся» или «остальные», то есть получится «оставшиеся параметры» или «остальные параметры». Буквально — это возможность собрать все параметры функции вместе, сколько бы при вызове не было передано аргументов.

Для описания **rest-параметров** используется специальный оператор, выглядит как многоточие `...`, следом за которым без пробелов идёт название переменной, в которую нужно записать все параметры функции. Это значит, что все аргументы, которые будут переданы при вызове, будут собраны в переменную, и обратиться к ней можно будет по заданному вами имени.

```
function имя_функции (...rest_параметры) {}
```

Разберём пример:

```
function getMaxValue (...values) {
    // В теле функции мы можем обращаться к переменной values,
    // как к обычному параметру, только в этой переменной будут
    // собраны все переданные аргументы

    // Дальше дело техники. Пока не прочитали параметры,
    // берём за максимальное значение минимально возможное число —
    // минус бесконечность
    let max = -Infinity;

    // Затем в цикле, пока не кончатся параметры...
    for (let i = 0; i < values.length; i++) {
        // Проверяем каждый параметр: не больше ли он максимального...
        if (values[i] > max) {
            // Если больше, то считаем за максимальный его
            max = values[i];
        }
    }

    // После цикла возвращаем из функции самый максимальный параметр
    return max;
}

getMaxValue(3, 2, 1); // Для вызова вроде этого переменная values будет содержать [3, 2, 1]
```

Если вы не до конца понимаете код функции (например, что значит запись `values[i]`) — ничего страшного. Просто вернитесь к этому примеру после раздела о массивах.

Не обязательно превращать все параметры функции в «оставшиеся». Можно определить нужное количество параметров отдельными переменными, а остальные как **rest-параметры**:

```
function имя_функции (первый_параметр, второй_параметр, ...rest_параметры) {}
```

Например:

```
function getMaxValue (a, b, ...values) {
    // Первые два аргумента при вызове станут
    // параметрами a и b соответственно
    // Все другие аргументы попадут в переменную values

    // ...
}

getMaxValue(3, 2, 1); // Для вызова вроде этого переменная values будет содержать [1], а 3 и 2
сттанут a и b соответственно
```

Переменную с оставшимися параметрами часто так и называют `rest`, однако на курсе такое имя нарушит критерий наименование переменных.

Однако, есть один нюанс: **rest-параметры** всегда должны идти последними и могут быть только одни. Впрочем, это следует даже из названия «остальные» или «оставшиеся». Поэтому объявить функцию с **rest-параметрами** посередине не выйдет:

```
function getMaxValue (a, b, ...values, c) {
    // ...
}
```

JavaScript сразу покажет ошибку `SyntaxError: parameter after rest parameter`.

### Оператор **rest**

В интернете вы можете встретить понятие «оператор **rest**» или «**rest-оператор**». Дело в том, что подход с **rest-параметрами** оказался настолько удобным, что его стали использовать не только для параметров функции, но и в других синтаксических конструкциях JavaScript. О таком использовании **rest-параметров** мы поговорим позже.

## Глава 2.8. Стрелочные функции

В JavaScript кроме функционального выражения и декларативного объявления функций существует более лаконичный синтаксис. Он особенно удобен при описании функций, выполняющих одно действие. Функции, определённые с помощью такого сокращённого синтаксиса, принято называть стрелочными функциями (от англ. *arrow function*).

### Синтаксис

Особенность стрелочных функций с точки зрения синтаксиса — отсутствие слова `function`. Вместо него используется «стрелка». Соединив вместе знак равенства (`=`) и больше (`>`), мы получим что-то похожее на «стрелку» (`=>`).

```
const имя_переменной = ([параметры_функции]) => {
  [тело_функции]
}
```

Определение обычной функции начинается с ключевого слова `function`, а стрелочной — с параметров. Открываются круглые скобки и перечисляются все параметры для функции. Как и у обычных функций, число параметров не ограничено.

После списка параметров идёт та самая стрелка, о которой упоминалось выше — `=>`. Благодаря этой комбинации символов, интерпретатор сможет понять, что это функция и её следует обрабатывать соответствующим образом.

За «стрелкой» следует описание тела функции. Для этого используются фигурные скобки. Здесь всё как у обычных функций: в теле допускается несколько операций, и чтобы вернуть из функции значение, применяется старый добрый оператор `return`. Рассмотрим на примере определение стрелочной функции для умножения чисел:

```
const multiply = (a, b) => {
  return a * b;
}

multiply(2, 2); // 4
```

Функция `multiply` определена в виде выражения. Только вместо функционального выражения применяется выражение стрелочной функции (arrow function expression). Важно запомнить: стрелочная функция всегда анонимна. Мы не можем задать для неё имя и объявить декларативно, как это делали с `function`:

```
function multiply (a, b) {
  return a * b;
}
```

Поэтому, если требуется описать стрелочную функцию, к которой планируется обращение в будущем, необходимо сохранить на неё ссылку в переменную. Одним словом, воспользоваться выражением стрелочной функции. В примере выше ссылка на стрелочную функцию для умножения сохраняется в константе `multiply`.

### Ещё более короткий синтаксис

Стрелочные функции можно описать ещё лаконичней. Всё зависит от самой функции. Если функция состоит из одного выражения (действия), то фигурные скобки и оператор `return` необязательны. Такая функция автоматически вернёт результат выполнения единственного действия.

```
const имя_переменной = ([параметры_функции]) => действие
```

Рассмотрим на примере функции умножения:

```
const multiply = (a, b) => a * b;
multiply(2, 2); // 4
```

Функция `multiply` состоит из одного выражения (`a * b`), поэтому применение `return` и фигурных скобок для описания тела функции не требуется. Функция вернёт результат вычисления `a * b` автоматически.

### Отбрасываем скобки

На этом возможность «сэкономить» на символах при определении стрелочных функции не заканчивается. В случаях, когда стрелочная функция принимает лишь один параметр, круглые скобки можно не писать.

```
const имя_переменной = параметр_функции => действие
```

Рассмотрим на примере:

```
const addTwo = count => count + 2;
addTwo(2); // 4
```

Определение функции `addTwo` выглядит ещё короче за счёт отказа от скобок. Однако мы рекомендуем не применять такой способ, а всегда описывать параметры стрелочной функции в скобках. Это удобно по некоторым причинам: при чтении кода глазу проще отделить тело функции от параметров. Причина субъективная, но многие разработчики сходятся в этом мнении. Другая причина заключается в упрощении рефакторинга. Если потребуется добавить второй параметр, то придётся возвращать скобки. Мелочь, но фактически дополнительное неудобство.

### Резюме

Стоит ли применять стрелочные функции повсеместно? Да, если не требуются возможности, присущие классическим функциям (вроде контекста `this`), и нужен более короткий синтаксис, особенно если функция состоит из одного действия.

## Глава 2.9. Оператор switch

Почти ни одна программа не обходится без применения условных операторов (`if...else`). Всегда есть условия, влияющие на логику выполнения программы. Однако, условный оператор не всегда отличный выбор для решения такой задачи.

Представим, что вы пишете программу «Персональный стилист», задача которой подбирать одежду в зависимости от погоды. Самое прямолинейное решение – использовать конструкцию `if...else if`, которая работает один в один (`if...else`), только поочередно проверяет не одно, а несколько условий, буквально: если не это, то другое; если не другое, то третье – и так далее:

```
// Опишем функцию выбора одежды
function getClothing (weather) {
  if (weather === 'Солнечно') {
    return 'Майку';
  } else if (weather === 'Ветрено') {
    return 'Куртку';
  } else if (weather === 'Дождливо') {
    return 'Дождевик';
  }

  return 'Непонятно!';
}

// Что надеть, если...
getClothing('Солнечно'); // 'Майку'
getClothing('Ветрено'); // 'Куртку'
getClothing('Дождливо'); // 'Дождевик'
getClothing('Неизвестная погода'); // 'Непонятно!'
```

### Минусы решения

#### Страдает читаемость

При большом количестве вариантов погоды эта конструкция разрастётся и будет выглядеть громоздко и многословно.

#### Сложно расширять

Допустим, нужно добавить дополнительные варианты погоды к текущему решению:

```
function getClothing (weather) {
  if (weather === 'Солнечно') {
    return 'Майку';
  } else if (weather === 'Ветрено') {
    return 'Куртку';
  } else if (weather === 'Дождливо') {
    return 'Дождевик';
  } else if (weather === 'Морозно') {
    return 'Пуховик';
  } else if (weather === 'Пасмурно') {
    return 'Плащ';
  }

  return 'Непонятно!';
}
```

Для каждого нового элемента массива придётся дописывать дополнительный блок `else if`, что само по себе неудобно, учитывая количество скобок, которые нужно не забыть закрыть.

#### Дублируется код

Это в свою очередь повышает вероятность ошибки. В коде постоянно повторяется трёхстрочная конструкция `else if (...) {}` и сравнение с параметром `weather`. Повторяющийся код читается не так внимательно. Следовательно, возрастают шансы на опечатки и ошибки. Хороший программист следует принципу DRY (Don't repeat yourself, не повторяйся) и избегает лишних повторений.

## Оператор switch

Для подобных задач в JavaScript предусмотрен специальный оператор – `switch` (англ. «переключатель»).

Приведём рефакторинг функции `getClothing`. Заменим лесенку из `else if` на оператор `switch`:

```
function getClothing (weather) {
  switch (weather) {
    case 'Солнечно':
      return 'Майку';
    case 'Ветрено':
      return 'Куртку';
    case 'Дождливо':
      return 'Дождевик';
    case 'Морозно':
      return 'Пуховик';
    case 'Пасмурно':
      return 'Плащ';
    default:
      return 'Непонятно!';
  }
}
```

`switch` принимает в скобках выражение, результат которого будет проходить строгое сравнение со значениями в `case` (англ. «случай»). При совпадении выполнится блок кода соответствующего `case`. Блок кода указывается через двоеточие после указания `case`.

Если тождественный случай отсутствует среди всех `case`, то сработает блок `default`. Этот блок используется для обработки непредусмотренных входных значений.

Чтобы прервать выполнение `switch`, используется `return` (только если `switch` внутри функции!) или ключевое слово `break`.

### Другой пример

Есть функция, задача которой определить, является ли число чётным или нет, но тут закралась ошибка, и функция считает все числа от 1 до 4 чётными. Так не пойдёт, давайте разбираться.

```
function isEven (number) {
  let result;
  switch (number) {
    case 0:
      result = 'Чётное';
      break;
    case 1:
      result = 'Нечётное';
    case 2:
      result = 'Чётное';
    case 3:
      result = 'Нечётное';
    case 4:
      result = 'Чётное';
      break;
    default:
      result = 'Я умею считать только до 4 :(';
  }
  return result;
}

isEven(1); // 'Чётное'.
```

Если не указать `return` или `break` внутри `case`, то переключатель пойдёт дальше, и функция вернёт неожиданный результат. В примере мы «проваливаемся» сквозь все кейсы, начиная с `case 1` и до `case 4`, и получаем результат `'Чётное'`, потому что `break` только в `case 4`.

#### Это не ошибка

С помощью этой механики можно объединять одинаковые случаи. В нашем примере, где функция умеет проверять только до 4, ответа может быть всего два. Объединим эти случаи и заменим `break` на `return` для краткости:

```
const isEven = function (number) {
  switch (number) {
    case 0:
    case 2:
    case 4:
      return 'Чётное';
    case 1:
    case 3:
      return 'Нечётное';
    default:
      return 'Я умею считать только до 4 :(';
  }
}

isEven(1); // 'Нечётное'
```

### Запомните

- Если `case` не объединяются, то должны содержать `break` или `return`.
- Выражение, передаваемое в `switch`, представляется в любом виде (примитив, переменная, вычисляемое значение или вызов функции).
- Порядок случаев не важен.
- Блок `default` не является обязательным, но его удобно использовать на случай ошибок или значений по умолчанию.
- При наличии одинаковых случаев будет выбран первый попавшийся:

```
switch (weather) {
  case 'Солнечно': // Будет выбран этот вариант
  return 'Майку';
  case 'Солнечно':
  return 'Кенгу';
}
```

### Резюме

Оператор `switch` улучшает читаемость кода, уменьшает вероятность ошибки и является отличным решением при работе с фиксированными наборами значений.

## Глава 2.10. Тернарный оператор

Все операторы различаются по количеству operandов, с которыми они взаимодействуют. Например, существует оператор минус `-`, который меняет знак числа на противоположный. Если такой оператор применяется к одному числу, то есть у него один operand, оператор называется **унарным**.

```
const negativeNumber = -7;
```

Кроме унарных операторов существуют операторы с двумя operandами — **бинарные**. Например, бинарный плюс `+` складывает два операнда:

```
const sum = 7 + 3;
```

**« Обратите внимание,** что и плюс, и минус могут выступать как в роли унарных операторов, так и в роли бинарных.

И, наконец, **тернарный оператор**. Тернарный (или условный) оператор существует во многих языках программирования — например, в C++, Java, Python, PHP и других. Кстати, в JavaScript это единственный оператор с тремя operandами:

```
условие ? выражение_1 : выражение_2
```

Первый operand — это **условие**. Если оно истинно (равно `true`), оператор вернёт значение **выражение1**. В ином случае оператор вернёт значение **выражение2**.

Что-то напоминает, да? По механике работы тернарный оператор похож на условную конструкцию `if` с альтернативной веткой `else`, но его синтаксис позволяет писать меньше строк кода. Сравним:

```
const booksCount = 19; // Количество книг, прочтённых за год
let result; // Сюда запишем результат сравнения booksCount с эталонным значением

// Сравним с помощью условной конструкции if
if (booksCount > 15) {
  result = 'План на год выполнен!';
} else {
  result = 'Читать и ещё раз читать';
}

// А теперь с помощью тернарного оператора
result = (booksCount > 15) ? 'План на год выполнен!' : 'Читать и ещё раз читать';
```

По сути оба фрагмента кода выполняют одно и то же действие — проверяют условие, а затем присваивают переменной первое или второе выражение в зависимости от истинности этого условия. Разница лишь в форме записи.

### Варианты использования

Значение, возвращаемое тернарным оператором, можно записать в переменную — этот вариант мы уже рассмотрели в примере выше. Кроме этого, его можно использовать в функциях при возвращении значения с помощью `return`:

```
function getResult (booksCount) {
  return (booksCount > 15) ? 'План на год выполнен!' : 'Читать и ещё раз читать';
}
getResult(19); // Функция вернёт значение 'План на год выполнен!'
```

Также возможно использование множественных тернарных операций. В этом случае несколько операторов `?` будут идти подряд:

```
const booksCount = 19;
let result = (booksCount > 15)
  ? 'План на год выполнен!'
  : (booksCount > 10)
    ? 'Уже неплохо!'
    : 'Читать и ещё раз читать';
```

Здесь каждое условие проверяется последовательно. Если первое условие истинно, переменной `result` присвоится значение `'План на год выполнен!'`. В ином случае код выполняется дальше, и проверяется второе условие (`booksCount > 10`), в зависимости от его истинности переменной `result` присвоится либо значение `'Уже неплохо!'`, либо `'Читать и ещё раз читать'`.

### Что выбрать: тернарный оператор или if?

При выборе за основной показатель нужно взять читабельность кода. Чем код понятнее, нагляднее, тем удобнее его рефакторить (так называется улучшение кода) и поддерживать. Тернарный оператор может как сделать код проще, так и необоснованно его усложнить. Это зависит от ситуации.

Посмотрим ещё раз на самый первый вариант, уже разобранный выше. Здесь переменной присваивается значение в зависимости от условия, и это пример грамотного использования тернарного оператора:

```
const booksCount = 19;
let result = (booksCount > 15) ? 'План на год выполнен!' : 'Читать и ещё раз читать';
```

В таком случае он позволяет избавиться от громоздкой условной конструкции и сделать код проще и короче.

Но есть варианты, когда использование оператора усложняет код. В большинстве случаев это относится к множественным тернарным операциям, о которых речь шла выше. Тем не менее не стоит отказываться от тернарного оператора. Он может помочь сделать код понятным и лаконичным. Главное — знать, в каких конкретно ситуациях его полезно использовать, и не злоупотреблять.

## Глава 2.11. Строгий режим

JavaScript активно развивается и меняется. Последние версии спецификаций расширяют возможности языка, сохраняя обратную совместимость. Увы, так было не всегда. История JavaScript повидала многое: взлёты и падения в виде неудачных решений. Чтобы как-то оградить разработчиков от возможностей и подходов, которыми не стоит пользоваться, в пятую версию спецификации был добавлен строгий режим для выполнения кода.

По умолчанию этот режим выключен (исключение ECMAScript-модули). Это сделано намерено для сохранения совместимости с устаревшим кодом. Как включить строгий режим узнаем ниже.

### Строгий режим

Главная задача строгого режима (strict mode) – уберечь разработчика от ошибок. Сделать невозможными некоторые безумные вещи, на которые раньше интерпретатор закрывал глаза. Это приводило к ошибкам на ровном месте. Немного позже разберём это утверждение на реальных примерах кода, а пока немного предварительных итогов:

- Строгий режим повышает требования к коду. Нет, он не убережёт от написания плохого кода, но от некоторых ошибок защитит. Вместо того, чтобы продолжить выполнять заведомо опасную и бессмыслицу операцию, консоль выведет ошибку.
- Включение строгого режима исправляет ошибки, которые мешают движку оптимизировать выполнение кода. В определенных случаях, код в строгом режиме может выполняться быстрей, чем аналогичный без включения строгого режима.
- Строгий режим запрещает использовать потенциально некорректные операции, устаревший синтаксис или зарезервированные слова для будущих версий языка.

### Как включить строгий режим

Для включения строгого режима применяется директива `"use strict"` или `'use strict'`. Да, здесь нет никакой ошибки – это самая обычная строка. Строгий режим может быть включён как для всего кода, так и для отдельной функции.

Если разместить директиву `"use strict"` в самом начале сценария (в первой строке), то её действие распространится на весь код сценария. Например:

```
'use strict';
// Любой код далее будет работать в строгом режиме
```

Строгий режим можно активировать для отдельной функции. Для этого директиву `"use strict"` следует определить в начале функции:

```
function inStrictMode() {
  'use strict';
  // Код функции работает в строгом режиме
}

// Код вне функции inStrictMode работает обычном режиме
function noStrictMode() {
  // Код внутри других функций, объявленных вне,
  // также работает обычном режиме
}

inStrictMode();
noStrictMode();
```

### Отключить невозможно

Отменить строгий режим невозможно. Спецификация не предусматривает для этого отдельной директивы. Если вы перешли в строгий режим, то вернуться в обычный не получится.

### Строгий режим по умолчанию

Стоит ли использовать строгий режим повсеместно? Да. Нет ситуаций, которые оправдывают применение нестрогого режима. Более того, начиная с ECMAScript 2015, строгий режим включён по умолчанию для ECMAScript модулей. Если в проекте используются модули, то включать вручную строгий режим не нужно. Он включён по умолчанию.

### Строгий режим включён

Мы знаем, какие задачи решает строгий режим. Теперь посмотрим на эту возможность с практической точки зрения. Разберём несколько примеров, которые продемонстрируют поведение интерпретатора при активированном строгом режиме.

#### Случайные переменные

Для объявления новой переменной применяются ключевые слова `let` и `const`. Есть ещё `var`, но в современном коде он не используется. Работая в нестрогом режиме, легко случайно объявить глобальную переменную. Стоит написать идентификатор переменной и присвоить ей значение, как автоматически будет создано новое свойство в `window`. Например:

```
someVar = 10;
anotherVar = true;
somVar = 11;
```

Мы не указывали никаких ключевых слов для определения переменной, но в нестрогом режиме ошибка не произойдёт. Все переменные будут созданы глобально, то есть у объекта `window` появятся однотипные свойства.

Особое внимание стоит обратить на последнюю строку (`somVar`). В этой строке мы хотели переопределить переменную `someVar`, но допустили ошибку – пропустили букву в названии переменной. В нестрогом режиме интерпретатор это пропустит и вместо ошибки (переменная не существует) создаст ещё одну глобальную переменную.

Попробуем добавить директиву `"use strict"` в наш код:

```
'use strict';
someVar = 10;
// ReferenceError: Can't find variable: someVar
anotherVar = true;
somVar = 11;
```

В строгом режиме объявление переменных без ключевого слова вызовет ошибку и выполнение кода остановится.

А если мы определим переменные правильно, допущенная нами ошибка, сразу же станет видна и мы сможем её легко исправить:

```
'use strict';
let someVar = 10;
let anotherVar = true;
somVar = 11;
// ReferenceError: Can't find variable: somVar
// Становится очевидно, что мы допустили ошибку в названии переменной
```

### Некорректные операции

Строгий режим предотвращает некоторые потенциально некорректные операции. Например, в нестрогом режиме ничего не мешает объявлять функцию с несколькими одинаковыми именами параметров:

```
function foo(a, b, c, a) {}
```

Для функции `foo` определено два параметра с именем `a`. Воспользоваться последним не получится, но в нестрогом режиме функция будет создана. Ошибка не возникнет. В строгом напротив, при попытке создать такую функцию произойдёт ошибка: `Duplicate parameter name not allowed in this context`.

Аналогичная ситуация с именами ключей в объектах. Строгий режим не допускает, чтобы имена ключей повторялись.

Другой пример некорректных операций – бессмыслицеское присваивание значений. Есть ли смысл присвоить значение «переменным» с именами `undefined`, `NaN`? Нет. Однако, в нестрогом режиме этот код не приведёт к ошибке:

```
let undefined = 5;
let NaN = 10;

console.log(undefined); // undefined
// Выведет undefined, а не 5, потому что undefined так перезаписать нельзя
```

Строгий режим пресечёт подобные ошибки, выбросив ошибку: `Cannot assign to read only property 'undefined' of object`.

На этом примеры некорректных операций не заканчиваются. Например, в нестрогом режиме допускается возможность удалить у объекта неудаляемое свойство. Фактически оно не будет удалено, но ошибка не произойдёт:

```
// вернёт false. Ошибки не будет
delete Object.prototype;
```

В строгом режиме выполнение такого кода приведёт к ошибке `Cannot delete property 'prototype' of function Object()`.

#### Зарезервированные слова

Строгий режим запрещает использовать зарезервированные слова в качестве имен переменных. Не все перечисленные ключевые слова задействованы в текущей версии стандарта, но их планируется использовать в будущем. Строгий режим уберегёт от их случайного применения: `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` и `yield`. При попытке объявить одну из перечисленных переменных в строгом режиме возникнет ошибка `Unexpected strict mode reserved word`.

#### Функции внутри условных операторов

Строгий режим запрещает декларативно объявлять функции внутри условных конструкций. В примере ниже функция `fn` не будет создана:

```
'use strict';
if (true) {
  function fn() {}
  fn();
}
```

#### arguments в качестве параметра

Внутри функций доступна переменная `arguments`. Это коллекция, которая содержит список всех аргументов, переданных функции. В строгом режиме у функции не может быть одноимённого параметра. При попытке объявить такую функцию возникнет ошибка `Unexpected eval or arguments in strict mode`.

```
'use strict';
function foo(arguments) {} // Unexpected eval or arguments in strict mode
```

### Семантические различия

Значением `this` (контекста) в нестрогом режиме при вызове функции был глобальный объект `window`. В строгом режиме значение `this` в таких случаях `undefined`:

```
function foo() {
  console.log('Контекст:', this);
}

const foo2 = function () {
  'use strict';
  console.log('Контекст:', this);
}

foo() // this равен window
foo2() // this равен undefined
```

### Резюме

Это лишь малая часть изменений, которые привносит строгий режим. С полным списком с подробными комментариями можно ознакомиться в [справочнике разработчика MDN](#).

## Глава 2.12. Контекст функций и проблема потери окружения

Контекст функции — некоторая противоположность областям видимости. Ключевая разница в том, что область видимости самой функции и доступные ей родительские области видимости определяются в момент объявления:

```
// Глобальная область видимости
const greeting = 'Привет!';

function say () {
    // Локальная область видимости функции say
    console.log(greeting);
}
```

Функции `say` доступна переменная `greeting`, где бы мы `say` не вызывали. Можно сказать, переменные родительской области видимости приклеились к функции `say`. В большинстве случаев это удобно, но бывают моменты, когда этого мало.

Давайте предположим, что мы хотим переписать функцию `say` в универсальный разговорный модуль, чтобы можно было этот модуль передать любому персонажу, и он бы смог разговаривать. Персонажи у нас будут описаны объектами, например:

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
};

const dog = {
    nickname: 'Полиграф Шариков',
    greeting: 'Абираул',
};

const fox = {
    nickname: 'Алиса',
    greeting: 'What does the fox say?',
};
```

Как нам научить Кекса приветствовать хозяина? Первый, наверное, самый очевидный вариант, воспользоваться тем, что мы уже знаем — областями видимости:

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
};

// ...

function say () {
    console.log(cat.nickname + ' говорит: ' + cat.greeting);
}

say(); // Кекс говорит: Мяу
```

Но мы уже обсуждали выше, что в таком случае объект `cat` приклеится к функции `say`, и тогда дать голос другим персонажам, собаке или лисе, не получится.

Другой вариант — передавать объект аргументом при вызове `say`:

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
};

const dog = {
    nickname: 'Полиграф Шариков',
    greeting: 'Абираул',
};

const fox = {
    nickname: 'Алиса',
    greeting: 'What does the fox say?',
};

function say (character) {
    console.log(character.nickname + ' говорит: ' + character.greeting);
}

say(cat); // Кекс говорит: Мяу
say(dog); // Полиграф Шариков говорит: Абираул
say(fox); // Алиса говорит: What does the fox say?
```

Кажется, что это уже решение... Но это лишь его часть. Потому что в таком исполнении для того, чтобы персонаж что-то сказал, нужно быть уверенным, что и объект, который описывает персонажа, и функция `say` есть в наличии в месте вызова.

А хорошо бы, чтобы функция в момент вызова сама понимала, для какого объекта её вызывают:

```
cat.say(); // Кекс говорит: Мяу
dog.say(); // Полиграф Шариков говорит: Абираул
fox.say(); // Алиса говорит: What does the fox say?
```

Для этого и нужен контекст функции. В синтаксисе JavaScript контекст функции «скрывается» за ключевым словом `this`:

```
function say () {
    console.log(this.nickname + ' говорит: ' + this.greeting);
}
```

Дальше нужно только представить функцию `say` как метод объекта:

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
    + say,
};

const dog = {
    nickname: 'Полиграф Шариков',
    greeting: 'Абираул',
    + say,
};

const fox = {
    nickname: 'Алиса',
    greeting: 'What does the fox say?',
    + say,
};
```

Может возникнуть вопрос: «А какое значение у `this`?» В этом смысле, что значение отсутствует и определяется только в момент вызова функции:

```
cat.say(); // В данном случае контектом будет cat
dog.say(); // А здесь уже dog
fox.say(); // Тут – fox
```

Где функция объявлена для контекста тоже не важно. Таким образом, контекст функции может меняться от вызова к вызову. Мы даже сами можем его изменять при необходимости, для этого в JavaScript предусмотрены методы `apply` и `call`, но нам они пока не понадобятся.

+ Полный пример кода

Итак, контекст функции `this` — это ссылка на объект, на котором была вызвана функция. Контекст определяется строго в момент вызова функции. При необходимости контекст при вызове можно переопределить помощью методов `apply` и `call`.

### Зачем это всё

Контекст позволяет описать функцию-метод множества однотипных объектов один раз, в одном месте, а затем этот метод переносить, что упрощает дальнейшую поддержку кода.

### Стрелочные функции — исключение

Особенность стрелочных функций, что их контекст ведёт себя скорее как область видимости, потому что контекст стрелочной функции зависит от того, где функция объявлена:

```
// Стрелочная функция объявлена просто в файле,
// значит её контектом станет глобальная область видимости,
// то есть window
const say = () => {
    console.log(this.nickname + ' говорит: ' + this.greeting);
}
```

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
    say,
};

cat.say(); // undefined говорит: undefined
```

► Почему `undefined`?

Кстати, если вам кажется, что стоит переместить функцию прямо в объект, и всё заработает как надо, нет:

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
    say: () => {
        console.log(this.nickname + ' говорит: ' + this.greeting);
    },
};

cat.say(); // undefined говорит: undefined
```

Вспомните, область видимости образуют блоки кода `{}`, а в объекте фигурные скобки `{}` не блок кода, а часть синтаксиса объекта.

Кстати, раз контекста в момент вызова стрелочная функция не образует, то и изменить его с помощью `apply` и `call` нельзя.

### Потеря окружения

Может показаться, что из-за отсутствия собственного контекста, стрелочные функции имеют ограниченное применение. Это не так. Особенность с контекстом реализована намеренно, так как позволяет избавиться от потери окружения — ситуации, когда мы не можем вызвать функцию с нужным нам контекстом.

Рассмотрим на примере... Представим, что нам нужно кроме приветствия перечислить список вкусняшек `goodies`, которые любит Кекс. Список оформлен в виде массива:

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
    goodies: [],
};
```

Учём это в нашей функции `say` и попросим кота подать голос:

```
function say () {
    console.log(this.nickname + ' говорит: ' + this.greeting);

    this.goodies.forEach(function (goodie) {
        console.log(this.nickname + ' любит: ' + goodie);
    });
}
```

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
    goodies: [
        'Свежую рыбку',
        'Шурпки хозяйских кроссовок',
    ],
    say,
};

cat.say();
```

```
// Кекс говорит: Мяу.
// undefined любит: Свежую рыбку
// undefined любит: Шурпки хозяйских кроссовок
```

Откуда же сплыть взлох? Ответ вы знаете: контекст определяется строго в момент вызова функции. А кто вызывает колбэк метода `forEach`? Сам JavaScript! И с каким контекстом он вызывает наш колбэк можно только догадываться. В таких случаях на помощь спешит стрелочная функция, контекст которой закрепляется в момент объявления, а не вызова:

```
function say () {
    console.log(this.nickname + ' говорит: ' + this.greeting);

    - this.goodies.forEach(function (goodie) {
        + this.goodies.forEach((goodie) => {
            console.log(this.nickname + ' любит: ' + goodie);
        });
    });
}
```

```
const cat = {
    nickname: 'Кекс',
    greeting: 'Мяу',
    goodies: [
        'Свежую рыбку',
        'Шурпки хозяйских кроссовок',
    ],
    say,
};

cat.say();
```

```
// Кекс говорит: Мяу.
// Кекс любит: Свежую рыбку
// Кекс любит: Шурпки хозяйских кроссовок
```

Теперь контекст колбэка будет равен контексту функции, в которой он объявлен — это наша `say` — а значит колбэку будет доступно свойство `this.name` объекта. Проблема потери окружения решена.

## Глава 2.13. Функции-конструкторы

Не пугайтесь этого названия, оно звучит страшней, чем есть на самом деле. Создать функцию-конструктор в JavaScript не сложней, чем обычную функцию.

А вот зачем нужны функции-конструкторы, и как ими пользоваться, мы подробно разберём в этом материале.

### Почти конструктор

Давайте немножко забудем про слово «конструктор», а вместо этого вспомним [демонстрацию](#), где мы учились генерировать данные для разработки, а точнее её часть:

```
const createWizard = () => {
  return {
    name: getRandomArrayElement(NAMES) + ' ' + getRandomArrayElement(SURNAMES),
    coatColor: getRandomArrayElement(COAT_COLORS),
    eyesColor: getRandomArrayElement(EYES_COLORS),
  };
};
```

Внимательно посмотрите на приведённый код и попробуйте ответить на вопрос: «Что в этом коде можно улучшить?»

Спрашивается: «А что здесь можно улучшать?» Объект предельно простой и не содержит лишних свойств. С этой точки зрения действительно всё хорошо. Но если докопаться до сути, то что делает функция `createWizard`? Она конструирует объект автоматически. Созданием одной простой функции мы решили несколько проблем: избавились от дублирования кода и сократили возможность допустить ошибку при описании свойств объекта.

Да, мы уже решили несколько важных проблем. Да, наша функция выполняет поставленные задачи, но... у неё есть несколько проблем.

Например, мы не можем знать, какой именно объект возвращает наша функция. Да, какой-то объект она возвращает, но где гарантия, что этот объект описывает волшебника? Об этом мы можем узнать только по косвенным признакам. Например, если у объекта есть ключ `coatColor`, то, возможно, перед нами действительно объект с данными волшебника. На первый взгляд всё верно, но ведь это не гарантия, что перед нами нужный объект.

Чтобы лучше понять, почему это проблема, рассмотрим пример, с которым вы наверняка столкнётесь в работе фронтенд-разработчиком:

```
const listOfNumbers = [1, 2, 3, 4];
console.dir(listOfNumbers);
```

Код приведённого листинга крайне прост: мы объявили массив и вывели его содержимое в консоль. Как нам отличить массив от чего-то другого? Мы можем проверить наличие свойства `length` (длина массива) или попробовать получить какой-то элемент по индексу:

```
console.log(listOfNumbers.length); // 4
console.log(listOfNumbers[0]); // 1
```

Также мы можем пойти дальше и проверить наличие метода `forEach`, который есть у всех массивов.

Теперь давайте взглянем на пример, который разобьёт в пух и прах наши рассуждения. Например:

```
const listOfDivs = document.querySelectorAll('div');
console.log(listOfDivs.forEach); // function...
```

В этом коде мы получаем из произвольного документа коллекцию `div` элементов. Теперь давайте посмотрим на результат и порассуждаем. На вид результат похож на массив. Свойство `length` есть. Метод `forEach` есть. Обратиться по индексу к элементу тоже возможно. Набор признаков совпадает, но ведь мы понимаем, что на самом деле это не массив. Не верите? Попробуйте у такого массива вызвать метод `map`:

```
listOfDivs.map(function(element) {
  console.log(element);
});
```

Такой код создаст нам ошибку `TypeError`. У массивоподобного объекта `listOfDivs` нет метода `map`. Получается, что, полагаясь на изучение объекта по формальным признакам (наличие определённых свойств, методов и т. д.), мы рискуем нарваться на ошибку в самом неожиданном месте.

Идея определения типа объекта по формальным признакам стара как мир. Этот подход называется «[универсальная типизация](#)». Идея подхода проста: если это выглядит как утка, плывет как утка и крякает как утка, то, возможно, это действительно утка. Чуть выше мы применили этот подход в действии, пытаясь определить тип объекта по формальным признакам (длина, наличие определённых свойств и т. д.).

Проверка с помощью «универсальной типизации» реализуется просто, но проблем у неё хватает. Все проблемы, которые мы рассмотрели на примере массива и массивоподобного объекта, также могут произойти и с функцией `createWizard`. Единственный способ убедиться, что функция будет возвращать то, что мы ожидаем — посмотреть её код.

### Функции-конструкторы

В языке JavaScript есть готовое решение для описанной проблемы. Пришло время познакомиться с функциями-конструкторами. Функции-конструкторы позволяют получить объект и быть точно уверенными, что этот объект был создан с помощью определённой функции-конструктора. Можно сказать, что у нас появляется возможность определять тип объекта. Перед тем, как начать рассматривать особенности применения функций-конструкторов, давайте перепишем наш пример. Напомню, мы хотим создать функцию, которая вернёт объект с данными волшебника.

```
const Wizard = function (name, coatColor, eyesColor) {
  this.name = name;
  this.coatColor = coatColor;
  this.eyesColor = eyesColor;
};

const someWizard = new Wizard(
  `${getRandomArrayElement(NAMES)} ${getRandomArrayElement(SURNAMES)}`,
  getRandomArrayElement(COAT_COLORS),
  getRandomArrayElement(EYES_COLORS),
);
```

Посмотрим, что здесь происходит. В самом начале мы объявляем новую функцию-конструктор (`Wizard`). Обратите внимание, имя функции мы записываем с прописной буквы и не используем в имени глагол (об этом позже).

В теле функции `Wizard` мы не создаём новый объект, как в прошлый раз, а добавляем ключи к `this`. Да-да, к тому самому контексту. Потому что в функции-конструкторе контекст `this` содержит ссылку на новый объект, который создаст и вернёт функция-конструктор. Поэтому, если нам нужно добавить какое-то свойство в возвращаемый объект, то мы так и пишем: `this.somethingProperty = 1`.

Чем ещё выделяется эта функция? Тут нет `return`. Зато есть `this`. Дело в том, что новый объект создаётся и возвращается автоматически при использовании `new`. Функции-конструкторы всегда должны вызываться с помощью оператора `new`. Если этого не сделать, то результат вас удивит. Попробуйте протестировать в консоли браузера.

### Объект, я тебя знаю

Мы переписали код, воспользовавшись в этот раз функцией-конструктором. Получилось неплохо, но пока мы не увидели главного преимущества и не можем ответить на вопрос: «Как убедиться, что полученный объект действительно создан с помощью определённой функции-конструктора?» Действительно, на выходе мы получили объект, но в прошлый раз мы тоже получали объект.

Как нам убедиться, что объект действительно создан с помощью конструктора `Wizard`? Для этого в языке JavaScript есть отдельный оператор `instanceof`. Пользоваться им просто:

```
проверяемый_объект instanceof функция_конструктор
```

Результатом вычисления этого выражения будет булево значение: `true` — значит объект был создан с помощью указанной функции-конструктора, `false` — нет. Как видите, ничего сложного в этом нет. Давайте посмотрим несколько примеров:

```
console.log(someWizard instanceof Wizard); // true
console.log({} instanceof Wizard); // false
```

Вот так просто и без лишних сложностей мы можем ответить на вопрос: создан ли объект с помощью функции-конструктора или, как принято говорить, является ли объект экземпляром определённого объекта.

### Вместо заключения

— В качестве имени для функции-конструктора всегда используется существительное. Никаких глаголов быть не должно. Например: `Car`, `Box`, `Wizard` и т. д.

— Имена таких функций принято писать с прописной буквы. Это договорённость и общая рекомендация, позволяющая легко и быстро отличать функции-конструкторы от других функций.

— Функции-конструкторы всегда вызываются с помощью оператора `new`.

— Доступ к объекту, который возвращает функция-конструктор находится в `this`. Именно через `this` вы должны добавлять все новые свойства.

— Вам необязательно писать `return` в теле конструктора. Объект, создаваемый внутри функции-конструктора (`this`), возвращается автоматически.

— Проверить, родство объекта с определённой функцией-конструктором нам поможет оператор `instanceof`.

« В стандарте ECMAScript 2015 появился другой, более лаконичный синтаксис для создания объектов по образу функций-конструкторов — классы. С ними мы подробно знакомим на курсе «JavaScript. Архитектура клиентских приложений»

## Глава 2.14. Введение в прототипы

### Что такое прототип

Прототип – объект, привязанный к функции-конструктору, содержащий общие методы и свойства для всех объектов, созданных с помощью этого конструктора. Определение звучит сложновато, поэтому давайте попробуем разобрать его по частям.

Итак, начнём с главного. Прототип – это объект. Так проще? Хорошо, тогда будем двигаться в этом направлении. Вспомним про функции-конструкторы. Каждый раз, когда вы создаёте функцию-конструктор, у вас создаётся специальный объект. Этот объект необходим для того, чтобы описывать в нём повторяющиеся от объекта к объекту свойства и методы. Давайте посмотрим на примере.

```
function Wizard (name, skill) {
  this.name = name;
  this.skill = skill;
  this.fire = function () {
    const baseFireballSize = 10;
    const fireballSize = baseFireballSize * this.skill;
    console.log(`Огненный шар размером ${fireballSize}`);
  };
}

const gandalfWizard = new Wizard('Гендальф', 5);
const sauronWizard = new Wizard('Саурон', 10);

gandalfWizard.fire(); // Огненный шар размером 50
sauronWizard.fire(); // Огненный шар размером 100
```

В этом примере мы описали функцию-конструктор `Wizard`. С её помощью мы будем создавать волшебников. У каждого волшебника есть имя (`name`) и уровень мастерства (`skill`). Чем выше уровень мастерства, тем более мощные огненные шары умеет создавать волшебник. Огненный шар волшебник создаёт с помощью метода (функции) `fire`.

Мы создали двух волшебников: Гендальфа и Саурана. У Саурана опыта больше, поэтому он умеет делать очень большие огненные шары. Код работает превосходно, и мы можем создать бесчисленную армию волшебников разного калибра.

Давайте подумаем вот о чём. Абсолютно у всех волшебников есть метод (функция) `fire`. Этот метод мы описали в теле функции-конструктора. Для каждого объекта, который мы создадим с помощью этой функции, будет создана своя уникальная функция `fire`. Получается, если мы сделаем пять волшебников, то функция `fire` будет создана пять раз. Одна и та же функция будет доступна в пяти разных копиях.

Вы, наверняка, догадались, что это не очень хорошо. При создании функции движку JavaScript требуется выделять какие-то ресурсы (например, память). Получается, чем больше у нас будет одинаковых функций, тем больше ресурсов придётся выделять впустую.

Давайте убедимся, что функция `fire` первого волшебника – это отдельная копия функции. Как это сделать? Да очень просто, мы можем их сравнить:

```
console.log(gandalfWizard.fire === sauronWizard.fire); // false
```

Результатом сравнения будет `false`, т. е. несмотря на идентичность поведения, это две абсолютно разные функции. Следовательно, если мы сделаем 1000 волшебников, то JavaScript придётся создать 1000 копий функции `fire`.

### Прототипы приходят на помощь

Решитьзвученную выше задачу не сложно, и помогут нам в этом прототипы. Напомним, прототип – это обычный объект. Он создаётся для каждой функции-конструктора. Это происходит автоматически, дополнительных действий от разработчика не требуется. Вы можете добавлять в этот объект методы (функции), свойства. Они будут доступны всем объектам, созданным с помощью функции-конструктора.

На практике это выглядит следующим образом. Давайте посмотрим на листинг ниже, в котором функция-конструктор `Wizard` немного отрефакторена. Мы вынесли метод `fire` в прототип.

```
function Wizard (name, skill) {
  this.name = name;
  this.skill = skill;
}

Wizard.prototype.fire = function () {
  const baseFireballSize = 10;
  const fireballSize = baseFireballSize * this.skill;
  console.log(`Огненный шар размером ${fireballSize}`);
}

// Остальной код остался без изменений
```

Самое интересное начинается в строке, где мы добавляем в объект `prototype` метод `fire`. Это самый обычный объект, следовательно, для добавления новых свойств или методов мы можем применять уже знакомые подходы. Мы вынесли в прототип метод `fire`. Теперь он будет существовать в единичном экземпляре, и все волшебники смогут им пользоваться. Мы можем в этом легко убедиться, повторно сравнив метод `fire` у волшебников. В этот раз результат будет `true`:

```
console.log(gandalfWizard.fire === sauronWizard.fire); // true
```

Сколько бы мы не создали волшебников, функция `fire` будет существовать в единичном экземпляре.

### Как это работает

Хорошо, проблему с `fire` мы решили, но возникает резонный вопрос: «А как это работает?». Как JavaScript понимает, что у волшебников есть метод `fire`? В конструкторе описаны только свойства `name` и `skill`, а метод `fire` описан совершенно в другом месте. Магия?

Нет, магии, к счастью, не существует. Когда создаётся новый объект (с помощью функции-конструктора), то помимо перечисленных в конструкторе свойств этому объекту добавляется свойство `__proto__`. В этом свойстве содержится ссылка на свойство `prototype` функции-конструктора, с помощью которой был создан объект, т. е. применительно к нашему случаю ссылка на `Wizard.prototype`. Вы можете это проверить самостоятельно:

```
console.log(gandalfWizard.__proto__ === Wizard.prototype); // true
```

Дальше алгоритм такой: при обращении к свойству или методу JavaScript пытается найти его среди собственных свойств объекта. Если поиск не увенчался успехом, то предпринимается попытка найти его в прототипе, а доступ к прототипу есть в свойстве `__proto__`.

« К счастью для разработчиков, начиная со стандарта ECMAScript 2015, в синтаксисе JavaScript была добавлена более удобная конструкция для реализации той же функциональности (и не только той же!) – классы. С ними мы подробно знакомимся на курсе «JavaScript. Архитектура клиентских приложений».

## Глава 3.1. Массивы и их методы

Массивы в JavaScript обладают набором полезных методов, позволяющих более гибко манипулировать их содержимым. С помощью этих методов вы можете перебирать значения массива (без использования циклов), трансформировать массив и делать другие полезные вещи. Методов у массивов больше двух десятков, мы разберём наиболее полезные и часто применяемые методы. Информацию о всех методах массивов вы всегда можете найти в [справочнике разработчика MDN](#).

Методом называют обычную функцию, которая является частью большой сущности, объекта. Например, знакомый вам из тренажёра метод `keks.plot()`. Многие методы массивов, если не сказать большая их часть, в качестве аргумента принимают другую функцию, которая будет вызвана не в момент передачи, а потом. Эта функция, предназначенная для отложенного выполнения, называется колбэком (от англ. `callback`, перезвонить) или по-русски «функцией обратного вызова».

```
[1, 2, 3].forEach(function () {});  
// ^^^^^^^^^^^^^^ вот он, колбэк
```

Действительно, принцип работы колбэков схож с заказом обратного телефонного звонка. Представьте, что вы звоните заказать пиццу, но срабатывает автоответчик, где приятный голос просит оставаться на линии, пока не освободится оператор, или предлагает заказать обратный звонок. Когда оператор освободится — он перезвонит и примет заказ. В таком случае вместо ожидания ответа оператора мы можем заниматься своими делами, как только нам перезвонят (произойдёт колбэк), мы сможем выполнить задуманное — заказать пиццу.

Так и с колбэками, которые мы передаём в методы массивов. Объявляем мы их сразу:

```
[1, 2, 3].forEach(function () {});
```

А вызов колбэка `function () {}` произойдёт где-то во внутренностях метода `.forEach()`, описанного в движке JavaScript.

## Глава 3.1. Массивы и их методы

Массивы в JavaScript обладают набором полезных методов, позволяющих более гибко манипулировать их содержимым. С помощью этих методов вы можете перебирать значения массива (без использования циклов), трансформировать массив и делать другие полезные вещи. Методов у массивов больше двух десятков, мы разберём наиболее полезные и часто применяемые методы. Информацию о всех методах массивов вы всегда можете найти в [справочнике разработчика MDN](#).

Методом называют обычную функцию, которая является частью большой сущности, объекта. Например, знакомый вам из тренажёра метод `keks.plot()`. Многие методы массивов, если не сказать большая их часть, в качестве аргумента принимают другую функцию, которая будет вызвана не в момент передачи, а потом. Эта функция, предназначенная для отложенного выполнения, называется колбэком (от англ. `callback`, перезвонить) или по-русски «функцией обратного вызова».

```
[1, 2, 3].forEach(function () {});  
// ^^^^^^^^^^^^^^ вот он, колбэк
```

Действительно, принцип работы колбэков схож с заказом обратного телефонного звонка. Представьте, что вы звоните заказать пиццу, но срабатывает автоответчик, где приятный голос просит оставаться на линии, пока не освободится оператор, или предлагает заказать обратный звонок. Когда оператор освободится — он перезвонит и примет заказ. В таком случае вместо ожидания ответа оператора мы можем заниматься своими делами, как только нам перезвонят (произойдёт колбэк), мы сможем выполнить задуманное — заказать пиццу.

Так и с колбэками, которые мы передаём в методы массивов. Объявляем мы их сразу:

```
[1, 2, 3].forEach(function () {});
```

А вызов колбэка `function () {}` произойдёт где-то во внутренностях метода `.forEach()`, описанного в движке JavaScript.

~1 минута

## Глава 3.1.1. Склейка элементов массива

Метод `.join()` приводит все элементы массива к строке и конкатенирует их в одну итоговую строку, разделяя переданным символом — разделителем.

```
const titles = ['Die hard', 'Terminator'];

const message = titles.join('. ');

console.log(message); // 'Die hard. Terminator'
```

Разделитель можно не передавать, по умолчанию это запятая:

```
const titles = ['Die hard', 'Terminator'];

const message = titles.join();

console.log(message); // 'Die hard,Terminator'
```

Если в массиве только один элемент, то он будет приведён к строке и возвращён без разделителя:

```
const titles = ['Die hard'];

const message = titles.join('.');

console.log(message); // 'Die hard', а не 'Die hard.'
```

**Обратите внимание,** что приведение элементов массива к строке производится по правилам приведения типов. И элементы вроде `undefined` или `null` будут приведены к пустой строке.

~1 минута

## Глава 3.1.2. Объединение массивов в один новый

Метод `.concat()` используется для склеивания двух и более массивов в один.

```
первый_массив.concat(второй_массив[ , третий_массив])
```

Например:

```
const ivanFavoriteFilms = ['Die hard', 'Terminator'];
const mariaFavoriteFilms = ['Kindergarten Cop'];

const favoriteFilms = ivanFavoriteFilms.concat(mariaFavoriteFilms);

console.log(favoriteFilms); // ['Die hard', 'Terminator', 'Kindergarten Cop']
```

Метод не изменяет исходный массив, а возвращает новый.

---

Если вы обнаружили ошибку или неработающую ссылку, выделите ее и нажмите Ctrl + Enter

~2 минуты

## Глава 3.1.3. Копирование массива или его части

Метод `slice` возвращает копию всех или части элементов исходного массива. Метод не изменяет исходный массив, а возвращает новый. Чтобы получить часть элементов в виде нового массива, нужно передать аргументами диапазон индексов:

```
массив.slice(минимальный_индекс, максимальный_индекс)
```

Например:

```
const films = ['Die hard', 'Terminator', 'Kindergarten Cop'];

console.log(films.slice(0, 2)); // ['Die hard', 'Terminator']
```

**Обратите внимание**, что элемент с максимальным индексом переданного диапазона в копию не попадает.

Например, элемент с индексом `2` в массиве `films` — это `'Kindergarten Cop'`, но он в копии не попал.

Кстати, если нужна копия элементов с какого-то индекса и до конца, максимальный индекс диапазона можно не указывать:

```
const films = ['Die hard', 'Terminator', 'Kindergarten Cop'];

console.log(films.slice(1)); // ['Terminator', 'Kindergarten Cop']
```

Кроме использования по прямому назначению — получение части элементов в виде нового массива — `slice` часто используют для создания копии целого массива. Для этого просто не указывают диапазон:

```
const films = ['Die hard', 'Terminator', 'Kindergarten Cop'];

const copyOfFilms = films.slice();

console.log(copyOfFilms); // ['Die hard', 'Terminator', 'Kindergarten Cop']

console.log(films === copyOfFilms); // false
```

Почему в конце при сравнении `films` и `copyOfFilms` получается `false`, рассказываем в главе:

- [Сравнение сложных типов данных](#)

~1 минута

## Глава 3.1.4. Расположение элементов массива в обратном порядке

Метод массива `.reverse()` «переворачивает» массив с ног на голову, то есть располагает элементы в том же массиве в обратном порядке. Рассмотрим на примере массива с числами:

```
const numbers = [0, 1, 2, 3, 4];
const reversedNumbers = numbers.reverse();

console.log(reversedNumbers); // [4, 3, 2, 1, 0]
console.log(numbers); // [4, 3, 2, 1, 0]
// ...
```

Так как элементы переставляются в исходном массиве, результат работы метода — ссылка на этот исходный массив:

```
// ...
console.log(numbers === reversedNumbers); // true
```

Такие методы ещё называют мутирующими, и чтобы избежать неожиданных мутаций, используют [метод `.slice\(\)`](#):

```
const numbers = [0, 1, 2, 3, 4];
const reversedNumbers = numbers.slice().reverse();

console.log(reversedNumbers); // [4, 3, 2, 1, 0]
console.log(numbers); // [0, 1, 2, 3, 4]
console.log(numbers === reversedNumbers); // false
```

~ 3 минуты

## Глава 3.1.5. Поиск элементов в массиве

Метод `.find()` позволяет решить одну из самых часто возникающих задач при работе с массивами — осуществить поиск элемента. Раз массивы позволяют хранить наборы значений, то рано или поздно может потребоваться найти значение, которое соответствует определённому условию.

Метод `.find()` как и другие методы аргументом принимает функцию, которая будет вызвана для каждого элемента массива, пока не найдётся элемент, который удовлетворяет условию. Как только такой элемент будет найден, метод `.find()` прекратит работу и вернёт найденный элемент.

```
массив.find((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Рассмотрим метод `.find()` на примере поиска подстроки в массиве:

```
const titles = ['Die hard', 'Terminator'];

const favoriteFilmTitle = titles.find((title) => title.includes('hard'));

console.log(favoriteFilmTitle); // 'Die hard'
```

### ▶ Что за метод `.includes()`

Но что вернёт функция, если в массиве есть несколько элементов, удовлетворяющих условию? Ответ прост: первый элемент, который соответствует условию. После этого работа метода будет прервана. А если метод `.find()` не найдёт ни одного элемента, удовлетворяющего условию, функция вернёт `undefined`.

## findIndex

Метод работает один в один как `.find()`, только результатом вернёт не найденный элемент, а его индекс в массиве.

```
массив.findIndex((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Например:

```
const titles = ['Die hard', 'Terminator'];

const favoriteFilmTitleIndex = titles.findIndex((title) => title.includes('hard'));

console.log(favoriteFilmTitleIndex); // 0
```

~ 2 минуты

## Глава 3.1.6. Перебор массива

Перебор значений, наверное, самая частая задача при работе с массивами. Если мы храним набор значений, то рано или поздно возникнет необходимость проделать какую-то операцию с элементами этого набора. Для решения этой задачи можно воспользоваться операторами циклов `for` или `while`, или сделать то же самое с помощью метода `.forEach()`.

Метод `.forEach()` позволяет выполнить произвольную функцию однократно для каждого элемента. Попросту говоря: он запускает перебор значений массива и для каждого значения выполняет функцию.

```
массив.forEach((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Рассмотрим пример:

```
const fruits = ['banana', 'apple', 'lemon', 'orange'];

fruits.forEach((value, index, array) => {
  console.log(value);
});

// Выведет в консоль:
//   'banana'
//   'apple'
//   'lemon'
//   'orange'
```

В примере выше мы определяем такую функцию с тремя параметрами:

- `value` — текущий элемент массива;
- `index` — порядковый номер (индекс) текущего элемента массива;
- `array` — ссылка на сам массив.

Эти параметры будут доступны при каждом вызове функции.

Если вам не требуется порядковый номер элемента в массиве или как-то взаимодействовать с массивом, то соответствующие параметры в функции можно не определять и воспользоваться более сокращённой записью:

```
массив.forEach((текущий_элемент_массива) => {})
```

Например:

```
fruits.forEach((value) => {
  console.log(value);
});
```

При использовании метода `.forEach()` стоит помнить одну важную деталь: работу метода нельзя остановить. Оператор `break` не поможет. Поэтому если вам требуется перебрать только часть массива, то `.forEach()` следует отодвинуть в сторонку и воспользоваться циклом `for`. Помните об этом.

## Глава 3.1.7. Преобразование массива

Преобразование — одна из частых задач при работе с массивами. С помощью метода `.map()` мы можем итерироваться по массиву, изменять его элементы и в результате получить новый массив.

```
массив.map((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Рассмотрим пример. Предположим, у нас есть массив `films` с фильмами. Каждый фильм описан в виде объекта с двумя ключами:

- `id` — идентификатор фильма;
- `title` — название фильма.

Примерно вот так:

```
const films = [
  {
    id: 0,
    title: 'Die hard',
  },
  {
    id: 1,
    title: 'Terminator',
  },
];
```

Наша задача заключается в получении массива, который будет содержать только названия фильмов. То есть на выходе мы должны получить массив вида: `['Die hard', 'Terminator']`.

У любой задачи всегда есть минимум два решения. Можно решить её «в лоб» и воспользоваться знаниями о методе `.forEach()`. Алгоритм будет таким: заводим новый массив и начинаем перебирать `films` методом `.forEach()`. В функции, которая будет вызываться для каждого элемента, напишем код для добавления названия фильма в новый массив. На этом задача, можно сказать, решена. Пример такого решения:

```
const films = [
  {
    id: 0,
    title: 'Die hard',
  },
  {
    id: 1,
    title: 'Terminator',
  },
];

const titles = [];

films.forEach((film, index) => {
  titles[index] = film.title;
});

console.log(titles); // ["Die hard", "Terminator"]
```

Код работает, однако нам нужно руками заводить пустой массив `titles`, руками в него добавлять элементы. Зачем, если существует метод `.map()`, который может взять эти обязанности на себя:

```
const films = [
  {
    id: 0,
    title: 'Die hard',
  },
  {
    id: 1,
    title: 'Terminator',
  },
];

const titles = films.map((film) => {
  return film.title;
});

console.log(titles); // ["Die hard", "Terminator"]
```

Результатом выполнения метода `.map()` будет новый массив, собранный из значений, которые вернёт функция, переданная в качестве параметра методу `.map()`.

Кстати, параметры колбэка у метода `.map()` такие же, как у колбэка `.forEach()` — `текущий_элемент_массива`, `индекс_текущего_элемента`, `ссылка_на_весь_массив`. Выходит, что метод `.map()` похож на `.forEach()`. Только он позволяет не просто перебирать все значения массива, а получить новый массив значений.

Метод `.map()` удобно использовать, когда требуется трансформировать массив, то есть создать новый массив на основе существующего.

## Глава 3.1.8. Свёртка массива

Может показаться, что `.reduce()` — ещё один метод, позволяющий перебрать содержимое массива, но его основная задача — свернуть массив, то есть из набора значений получить одно. Это значение может быть произвольного типа. За счёт этой возможности метод `.reduce()` становится мощным инструментом, позволяющим решить множество разных задач.

Перед тем как познакомиться с примером, давайте рассмотрим аргументы самого метода:

```
массив.reduce(колбэк_функция[, начальное_значение_результата]);
```

В отличие от других методов массива `.reduce()` может принимать второй аргумент — начальное значение результата, а точнее результирующего значения, того, что мы получим по итогу работы метода. Этот аргумент опциональный (в описании такие оборачиваются `[, ]`), его можно передать:

```
[1, 2, 3].reduce(() => {}, 0);
```

А можно не передавать:

```
[1, 2, 3].reduce(() => {});
```

Что будет, если его не передать, расскажем в конце главы.

Теперь разберём параметры колбэк-функции:

```
(результатившее_значение, текущий_элемент_массива, индекс_текущего_элемента,  
ссылка_на_сам_массив) => {};
```

Их аж четыре:

- `результатившее_значение` — параметр, через который передаётся предыдущий результат выполнения функции, таким образом этот параметр «кочует» от перебора одного элемента к перебору другого. Та отличительная особенность `.reduce()` от других методов;
- `текущий_элемент_массива` — элемент массива, для которого вызывается колбэк-функция;
- `индекс_текущего_элемента` и `ссылка_на_сам_массив` — тут должно быть понятно без лишних слов.

Рассмотрим применение метода `.reduce()` на практическом примере — подсчёт суммы. Для наглядности опишем задачу так: есть корзина с товарами (массив), где каждый товар представлен объектом с несколькими ключами: `title` (название товара), `quantity` (количество) и `price` (цена). Наша задача заключается в подсчёте общей суммы стоимости всех товаров, и в этом нам поможет метод `.reduce()`:

```
const goods = [  
  {  
    title: 'Кукуруза',  
    quantity: 3,  
    price: 99,  
  },  
  {  
    title: 'Корм для кота',  
    quantity: 2,  
    price: 113,  
  },  
];  
  
const sum = goods.reduce((total, product) => total + (product.quantity * product.price), 0);  
  
console.log(sum); // 523
```

В этом примере мы посчитали общую сумму стоимости всех товаров и для этого нам потребовалась одна строчка кода, никаких дополнительных переменных как в случае с `.forEach()` или `for`-циклом.

Как это работает? Метод `.reduce()` вызывает переданную функцию для каждого элемента массива. Результат выполнения этой функции доступен на следующей итерации через параметр `total`. Для первой итерации значение `total` будет `0`, потому что мы передали его вторым аргументом в сам `.reduce()`. После завершения всех итераций значение `total` станет результатом выполнения `.reduce()`.

Что происходит внутри колбэк-функции? Она задаёт новое результативное значение. Не изменяет, а именно задаёт новое! Да, на основе предыдущего, потому что мы прибавляем к результату прошлой итерации произведение количества товара и его стоимости.

Метод `.reduce()` штука сложная, поэтому давайте ещё раз разберём, как будет выполняться этот код, но уже по шагам.

На первой итерации результирующим значением будет `0` — мы определили его самостоятельно, передав вторым аргументом в метод `.reduce()`. Получается, что на первой итерации выражение будет таким:

```
0 + (3 * 99) = 297;
```

Это мы посчитали стоимость первого товара («Кукуруза»).

На второй итерации мы переходим к следующему товару — «Корм для кота». На этот раз значением `total` будет `297` — результат выполнения функции на прошлой итерации — и к этому значению мы прибавляем стоимость второго товара:

```
297 + (2 * 113) = 523;
```

Поскольку больше элементов в массиве нет, метод `.reduce()` завершит работу и вернёт значение `total`, поэтому результатом выполнения `.reduce()` будет `523`.

### Что будет, если не передать начальное значение результата?

Если забыть или нарочно не передать второй аргумент в `.reduce()`, тогда он начнёт обход массива со второго элемента, а начальным значением возьмёт первый элемент:

```
['x', 'y'].reduce((total, current, index) => {  
  console.log(total); // 'x'  
  console.log(current); // 'y'  
  console.log(index); // 1  
});
```

Из примера видно, что `.reduce()` начал работу сразу с `'y'`, а результирующим значением для первой проходки взял `'x'`.

Это не ошибка, а заложенное поведение. Удобно, когда вам нужно просуммировать элементы массива — числа:

```
[5, 2, 3].reduce((total, current) => total + current); // 10
```

Результат будет аналогичный «полней» записи:

```
[5, 2, 3].reduce((total, current) => total + current, 0); // 10
```

~2 минуты

## Глава 3.1.9. Проверка каждого элемента массива на условие «удовлетворяют все»

При помощи метода `.every()` можно проверить, что условию соответствуют все элементы массива.

Результатом вызова метода `.every()` будет булево значение: `true` или `false`.

```
массив.every((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Метод `.every()` также аргументом принимает функцию, которая будет вызываться для каждого элемента до тех пор, пока при проверке условия не вернётся `false`. В таком случае метод `.every()` прекратит выполнение и вернёт `false`. Если для каждого элемента массива будет возвращено значение `true`, то результатом метода `.every()` также станет `true`:

```
const numbers = [11, 12, 13, 15, 100];

const isEveryNumberOverTen = numbers.every((value) => {
  return value > 10;
}); // every вернёт true, потому что все элементы массива больше 10
```

Добавим в массив `numbers` элемент меньше `10`:

```
const numbers = [11, 12, 13, 15, 100, 9];

const isEveryNumberOverTen = numbers.every((value) => {
  return value > 10;
}); // every вернёт false, потому что один элемент массива меньше 10
```

У метода `.every()` есть одна особенность. Если вызвать его на пустом массиве, то результатом всегда будет `true` вне зависимости от условия:

```
const result = [].every(() => {
  return 1 === 1;
}); // every всё равно вернёт true, хотя массив пустой
```

~ 2 минуты

## Глава 3.1.10. Проверка каждого элемента массива на условие «удовлетворяет хоть один»

Метод `.some()` тоже относится к перебирающим методам массива. То есть это очередной метод, позволяющий перебрать элементы массива, но с одной особенностью. С его помощью можно проверить, присутствует ли в массиве элемент, который удовлетворяет определённому условию. Результатом выполнения метода `.some()` будет булево значение: `true` или `false`.

```
массив.some((текущий_элемент_массива, индекс_текущего_элемента, ссылка_на_весь_массив) => {})
```

Метод `some` перебирает элементы массива и для каждого элемента вызывает переданную функцию. Метод `.some()` будет вызывать функцию для каждого элемента, пока она не вернёт `true`. Как только это случится, метод `.some()` прервёт работу и вернёт в качестве результата значение `true`.

Если для всех элементов массива переданная функция вернёт `false`, тогда результатом `.some()` станет `false`. Получается, метод `.some()` решает задачу проверки элементов массива на соответствие какому-то условию. Рассмотрим на практическом примере:

```
const numbers = [1, 4, 10, 5];

const isExistsOverFive = numbers.some((value) => {
    return value > 5; // Проверяем каждый элемент, больше ли он, чем 5
}); // Когда some дойдёт до 10, то прекратит работу и вернёт true

const isExistsOverTwenty = numbers.some((value) => {
    return value > 20; // Проверяем каждый элемент, больше ли он, чем 20
}); // some пройдёт все элементы, они все меньше 20, поэтому some вернёт false
```

У метода `.some()` есть одна особенность. Если вызвать его на пустом массиве, то результатом всегда будет `false` вне зависимости от условия:

```
const result = [].some(() => {
    return 1 === 1;
}); // так как массив пустой, some вернёт false
```

## Глава 3.2. spread-синтаксис

spread-синтаксис или синтаксис расширения, распространения — это способ расширить одну перечисляемую структуру элементами другой. Можно сказать, spread-синтаксис в какой-то степени обратный rest-параметрам. Даже знак используется один и тот же, многоточие `...`

Для примера возьмём массивы. Для объединения массивов существует метод `.concat()`:

```
[1, 2].concat([3, 4]); // [1, 2, 3, 4]
```

С помощью spread-синтаксиса можно обойтись без использования `.concat()`:

```
[1, 2, ...[3, 4]]; // [1, 2, 3, 4]
```

*Мы как бы говорим JavaScript: «Отбрось скобки».*

Может возникнуть справедливый вопрос, а зачем нам ещё один способ сделать `.concat()`? Дело в том, что spread-синтаксис работает гораздо шире. Метод `.concat()` возвращает массив, а spread откидывает скобки и возвращает содержимое массива. И если место, где это произошло, может принять содержимое — это важно — мы получим элементы массива через запятую.

А кто умеет принимать значения через запятую? Ну, кроме объявления массива... Функции! Например, существует метод `Math.max()`, который принимает через запятую значения и возвращает максимальное из них:

```
Math.max(1, 2, Infinity, 100, -3); // Infinity
```

Но что делать, если у нас значения хранятся в массиве:

```
const numbers = [1, 2, Infinity, 100, -3];
Math.max(numbers); // NaN
```

`Math.max()` не умеет работать с массивами, поэтому результат `NaN`.

На помощь приходит spread-синтаксис, который «отбрасывает скобки», а содержимое массива передаёт в `Math.max()`:

```
const numbers = [1, 2, Infinity, 100, -3];
Math.max(...numbers); // Infinity
```

Мысленно можно представить, что произошёл вызов `Math.max(1, 2, Infinity, 100, -3)`.

Также в начале главы не просто так говорится о «перечисляемых структурах», а не просто о массивах. spread-синтаксис можно использовать, например, с объектами вместо `Object.assign()`:

```
// Обе записи равнозначны
Object.assign({ a:1 }, { b:2 }); // { a:1, b:2 }
({ a:1, ...{ b:2 }}); // { a:1, b:2 }
```

Круглые скобки нужны, чтобы JavaScript отличил объявление объекта от блока кода `{}`.

А вот в функцию передать такой объект не получится:

```
console.log(...{ a:1 }); // TypeError: ({a:1}) is not iterable
```

Чтобы понять почему, достаточно мысленно отбросить скобки — `console.log( a:1 )` — согласитесь, странный код.

Также не получится расширить массив содержимым объекта:

```
[1, 2, 3, ...{ a:0 }]; // TypeError: ({a:0}) is not iterable
```

А вот массивом расширить объект получится:

```
({ a:1, ...['b', 'c']}); // { 0:'b', 1:'c', a:1 }
```

► Попробуйте ответить «Почему?» самостоятельно, а после посмотрите ответ

Поэтому всегда учитывайте содержимое и место его вставки, когда используете spread-синтаксис.

## Глава 3.3. Отличия spread-синтаксиса от rest-параметров

Главный вопрос о rest-параметрах и spread-синтаксисе — это как отличить одно от другого, если в обоих случаях используется символ многоточие `...`

В этом поможет контекст, а по-русски — смысл. rest означает «оставшийся». Например «оставшиеся параметры функции»:

```
function doAll (a, b, ...rest) {}

doAll(1, 2, 3, 4); // В rest будет [3, 4]
```

Или «оставшиеся ключи объекта» при деструктуризации:

```
const {a, b, ...rest} = { a:1, b:2, c:3, d:4 }; // В rest будет { c:3, d:4 }
```

A spread означает «расширить», «распустить». Например, мы хотим «распустить элементы массива», чтобы они стали аргументами функции:

```
function doAll (a, b, c) {}

doAll(...[1, 2, 3]);
```

Или когда мы хотим «расширить один массив элементами другого»:

```
[1, ...[2, 3]]; // [1, 2, 3]
```

### Нюансы использования

#### rest-параметры

Здесь ничего нового, rest-параметры всегда должны идти последними и могут быть только одни. Никаких других нюансов.

#### spread-синтаксис

А вот здесь нужно быть внимательнее. Первое, что отличает синтаксис расширения от rest-параметров, что он не обязательно должен идти последним:

```
[1, ...[2, 3], 4]; // [1, 2, 3, 4]
```

Также можно использовать spread столько раз, сколько захочется, и с вложенными:

```
[1, ...[2, 3], 4, ...[5, ...[6, 7]]]; // [1, 2, 3, 4, 5, 6, 7]
```

А можно составить массив только из spread-синтаксиса:

```
[...[1, 2, 3], ...[4, 5], ...[6, 7]]; // [1, 2, 3, 4, 5, 6, 7]
```

Только нужно помнить, что в случае с массивом, где будет spread, туда элементы и вставляются:

```
[...[2, 3], 1, 4]; // [2, 3, 1, 4]
```

Потому что в массиве элементы упорядочены по индексам. А в случае с объектом такой проблемы нет, потому что пары ключ-значение в объектах не упорядочены:

```
({ ...{ c:3, b:2 }, a:1}); // { a:1, b:2, c:3 }
```

Кстати, порядок по алфавиту тоже не гарантирован, каждый браузер может выстраивать пары в угодном одному ему порядке. Единственное, что гарантируется, что все указанные ключи будут присутствовать.

### Резюме

spread-синтаксис и rest-параметры помогают очень быстро вытаскивать отдельные значения из списков или, наоборот, быстро создавать новые списки на основе других списков. spread-синтаксис к тому же мощный инструмент, который применим шире, чем `.concat()` и `Object.assign()`, а ещё он лаконичнее, за что полюбился многим фронтенд-разработчикам и встречается повсеместно.

## Глава 4.1. Что такое DOM

**DOM** (Document Object Model) – это объектная модель документа. Разберём определение по частям:

- Объектная – потому что состоит из объектов;
- Модель – слово в прямом своём значении;
- Документ – имеется в виду веб-страница.

Для примера возьмём простую HTML-страницу:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>DOM</title>
  </head>
  <body>
    <h1>Document Object Model</h1>
    <p>DOM <em>(объектная модель документа)</em> – способ представления разметки страницы в виде связанных между собой объектов</p>
    <p>Каждому элементу на странице – тегу, текстовому блоку, комментарию – в JS ставится в соответствие объект</p>
    <p>Каждый из объектов знает про свой родительский объект, соседние объекты и объекты, расположенные внутри него</p>
    <p>Главный объект, из которого начинают «растить» все остальные элементы DOM-дерева – document.</p>
  </body>
</html>
```

Проблема в том, что браузер понимает HTML, а JavaScript – нет. И чтобы управлять разметкой из JavaScript, например для добавления интерактивности на страницу, нам нужен специальный инструмент. Этим инструментом является DOM.

Образно выражаясь, DOM следует воспринимать как некий словарь для JavaScript к HTML-разметке веб-страницы. DOM описывает HTML-структурку объектами JS (теми самыми, которые в фигурных скобках). То есть всю нашу страницу можно представить в виде объекта `document`. В `document` есть ключ `documentElement`, который соответствует корневому элементу документа – тегу `html`. В `documentElement` лежит `head`, `body` и так далее.

```
const document = {
  documentElement: {
    head: { /*...*/ },
    body: {
      h1: { /*...*/ },
      p: { /*...*/ },
      p: { /*...*/ },
      p: { /*...*/ }
    }
  }
}
```

И тогда доступ к элементу объекта возможен по ключу. Например, к заголовку `h1` по пути `document.documentElement.body.h1`.

Следует помнить, что DOM – это не просто точное отражение разметки, а сверхточное, содержащее массу полезных вещей, которые позволяют программисту удобно работать с разметкой.

К таким удобным инструментам в DOM относятся ссылки для быстрого доступа. Например, чтобы получить доступ к узлу `body`, не обязательно обращаться `document.documentElement.body`, а можно сразу – `document.body`. Так же можно поступить, например, с формами – в ключе `document.forms` будут собраны все формы на странице. И таких ссылок для быстрого доступа существует множество.

Однако в DOM присутствуют не только теги. Например, после тега `body` есть перенос строки и табуляция в 4 пробела. Вспомним, как мы пишем код:

- открываем тег `<body>`,
- далее нажимаем клавишу [ENTER] для переноса строки,
- далее в новой строке нажимаем клавишу [TAB] для отступа,
- и далее, например, открываем тег `<h1>`,
- затем, перенос строки, табуляция,
- и, к примеру, в `<h1>` кладём ссылку `<a>`...

```
<body>^
  ^<h1>^
    ^<a href="#" target="_blank">Ссылка</a>
```

С тегом `<a>` тоже всё непросто. У него заданы атрибуты `href` и `target`, а ещё есть содержимое – текст. Поэтому с точки зрения DOM ссылка – это отдельный объект. Кстати, так бывает не всегда. Любой тег может быть представлен как примитив, а может как объект. Потому что DOM экономный.

Теги образуют узлы-элементы, а текст внутри тега образует текстовый узел. Переносы строки, пробелы и табуляция – всё это полноправные текстовые узлы. Даже комментарий является элементом DOM. И вообще всё, что есть в HTML – есть в DOM. Таким образом, HTML-разметка в веб-странице является основой для формирования первоначального, исходного состояния DOM.

Вернёмся к примеру с ссылкой. У неё есть адрес, куда она ведёт, и текст. Это как минимум. Помимо этого там может быть набор атрибутов, например «открыться в новом окне» или «не следить за мной» и так далее. Всё это превратится в поля объекта:

```
{
  /*...*/
  a: {
    href: '#',
    target: '_blank',
    textContent: 'Ссылка'
  }
  /*...*/
}
```

### Резюме

DOM – это огромный объект, который имеет древовидную структуру. Часто можно встретить выражение – DOM-дерево.

## Глава 4.1. Что такое DOM

**DOM** (Document Object Model) – это объектная модель документа. Разберём определение по частям:

- Объектная – потому что состоит из объектов;
- Модель – слово в прямом своём значении;
- Документ – имеется в виду веб-страница.

Для примера возьмём простую HTML-страницу:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>DOM</title>
  </head>
  <body>
    <h1>Document Object Model</h1>
    <p>DOM <em>(объектная модель документа)</em> – способ представления разметки страницы в виде связанных между собой объектов</p>
    <p>Каждому элементу на странице – тегу, текстовому блоку, комментарию – в JS ставится в соответствие объект</p>
    <p>Каждый из объектов знает про свой родительский объект, соседние объекты и объекты, расположенные внутри него</p>
    <p>Главный объект, из которого начинают «растить» все остальные элементы DOM-дерева – document.</p>
  </body>
</html>
```

Проблема в том, что браузер понимает HTML, а JavaScript – нет. И чтобы управлять разметкой из JavaScript, например для добавления интерактивности на страницу, нам нужен специальный инструмент. Этим инструментом является DOM.

Образно выражаясь, DOM следует воспринимать как некий словарь для JavaScript к HTML-разметке веб-страницы. DOM описывает HTML-структурку объектами JS (теми самыми, которые в фигурных скобках). То есть всю нашу страницу можно представить в виде объекта `document`. В `document` есть ключ `documentElement`, который соответствует корневому элементу документа – тегу `html`. В `documentElement` лежит `head`, `body` и так далее.

```
const document = {
  documentElement: {
    head: { /*...*/ },
    body: {
      h1: { /*...*/ },
      p: { /*...*/ },
      p: { /*...*/ },
      p: { /*...*/ }
    }
  }
}
```

И тогда доступ к элементу объекта возможен по ключу. Например, к заголовку `h1` по пути `document.documentElement.body.h1`.

Следует помнить, что DOM – это не просто точное отражение разметки, а сверхточное, содержащее массу полезных вещей, которые позволяют программисту удобно работать с разметкой.

К таким удобным инструментам в DOM относятся ссылки для быстрого доступа. Например, чтобы получить доступ к узлу `body`, не обязательно обращаться `document.documentElement.body`, а можно сразу – `document.body`. Так же можно поступить, например, с формами – в ключе `document.forms` будут собраны все формы на странице. И таких ссылок для быстрого доступа существует множество.

Однако в DOM присутствуют не только теги. Например, после тега `body` есть перенос строки и табуляция в 4 пробела. Вспомним, как мы пишем код:

- открываем тег `<body>`,
- далее нажимаем клавишу [ENTER] для переноса строки,
- далее в новой строке нажимаем клавишу [TAB] для отступа,
- и далее, например, открываем тег `<h1>`,
- затем, перенос строки, табуляция,
- и, к примеру, в `<h1>` кладём ссылку `<a>`...

```
<body>^
  ^<h1>^
    ^<a href="#" target="_blank">Ссылка</a>
```

С тегом `<a>` тоже всё непросто. У него заданы атрибуты `href` и `target`, а ещё есть содержимое – текст. Поэтому с точки зрения DOM ссылка – это отдельный объект. Кстати, так бывает не всегда. Любой тег может быть представлен как примитив, а может как объект. Потому что DOM экономный.

Теги образуют узлы-элементы, а текст внутри тега образует текстовый узел. Переносы строки, пробелы и табуляция – всё это полноправные текстовые узлы. Даже комментарий является элементом DOM. И вообще всё, что есть в HTML – есть в DOM. Таким образом, HTML-разметка в веб-странице является основой для формирования первоначального, исходного состояния DOM.

Вернёмся к примеру с ссылкой. У неё есть адрес, куда она ведёт, и текст. Это как минимум. Помимо этого там может быть набор атрибутов, например «открыться в новом окне» или «не следить за мной» и так далее. Всё это превратится в поля объекта:

```
{
  /*...*/
  a: {
    href: '#',
    target: '_blank',
    textContent: 'Ссылка'
  }
  /*...*/
}
```

### Резюме

DOM – это огромный объект, который имеет древовидную структуру. Часто можно встретить выражение – DOM-дерево.

## Глава 4.2. DOM-дерево

**Дерево** — это структура. Такое название структуре дано не зря, она действительно напоминает дерево, только перевёрнутое: корень вверху, от корня вниз идут ветки и листья. Каждая часть дерева называется элементом, а в DOM-дереве элемент называется узлом. То есть **узел** — это абсолютно любой элемент дерева.

Узлы бывают родителями — когда у них есть дочерние узлы. И узлы бывают детьми — это узлы, у которых есть родитель.

Продолжая аналогию: у дерева и корень, и ветки, и листья — это узлы. Для тех веток, которые растут прямо из ствола, ствол — это родитель, с другой стороны для ствола ветка — это ребёнок. И такая иерархическая структура идёт от корня до листьев.

Почти все узлы могут быть одновременно и родителями и детьми. Почти, потому что есть такие узлы, как корень и листья. Корень — это тот элемент, с которого начинается дерево. То есть, выше корня ничего нет, и у корня нет родителя. Соответственно, каждый элемент, за исключением корня, имеет одного родителя. А лист — это узел, у которого нет детей. Соответственно, каждый узел, кроме листа, имеет любое количество детей.

👉 Корень — это узел, у которого есть только потомки, лист — это узел, у которого есть только родитель.

### Важные моменты

- Корень может быть только один.
- Родитель может быть только один.

У DOM-дерева корень — это `document`. Узел `html` не может являться корнем, потому что DOM-дерево содержит, кроме тегов, и отступы, и табуляции, и комментарии и так далее. И на одном уровне с `html` могут оказаться ещё узлы, а корень может быть только один, поэтому корень — это `document`. Для примера возьмём такую разметку:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
  </body>
</html>
```

У каждого узла, представленного объектом, есть свойство `parentElement`, которое содержит информацию о родителе элемента.

```
console.log(document.parentElement); // null
```

Потому что у корня не может быть родителей. В свойстве `children` записаны дочерние элементы узла:

```
console.log(document.children); // documentElement — в DOM дочерний элемент document называется не html, а именно documentElement
// Остальные DOM-узлы называются так же, как и теги в HTML
```

Свойство `children` можно воспринимать как массивоподобную коллекцию детей, а значит мы можем написать код, который будет перебирать всех детей и выводить в консоль структуру.

Чтобы узнать имя текущего элемента, нужно обратиться к свойству `tagName` или `nodeName`. По соглашению, для HTML-документов имя тега всегда возвращается в верхнем регистре, поэтому важно не забывать приводить к общему написанию с помощью метода строки `toLowerCase`.

Выведем структуру узла `html`, включая все дочерние элементы, на примере такой страницы:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Document Object Model</h1>
  </body>
</html>
```

Скрипт для вывода структуры узла `html` будет иметь следующий вид:

```
const html = document.documentElement;
for (let i = 0; i < html.children.length; i++) {
  const child = html.children[i];
  console.log(child.tagName.toLowerCase());
  for (let j = 0; j < child.children.length; j++) {
    const innerChild = child.children[j];
    console.log('|---' + innerChild.tagName.toLowerCase());
  }
}
```

Скрипт в цикле перебирает все дочерние элементы HTML-узла — `documentElement.children`. У каждого ребёнка в свою очередь перебирает его дочерние элементы и выводит имена тегов, приведённые к нижнему регистру.

Результат будет иметь следующий вид:

```
head
|---meta
body
|---h1
```

## Глава 4.3. Поиск в деревьях

В HTML все элементы, кроме `document`, являются вложенными: `head` и `body` вложены в `document`, элементы списка `li` могут быть вложены в элемент нумерованного списка `ol` или неупорядоченного списка `ul`. Структура, которую представляет HTML-документ, а соответственно и DOM, является деревом.

Когда возникает необходимость найти элемент в древовидной структуре, сделать это можно двумя способами:

1. Поиск в глубину (DFS – Depth-First Search)
2. Поиск в ширину (BFS – Breadth-First Search)

Хотя в JavaScript для поиска элемента используется первый способ — поиск в глубину, знать оба полезно.

Для разбора каждого из подходов возьмём следующую древовидную структуру, которая представляет HTML-документ. Соответственно, каждый узел — это HTML-элемент.

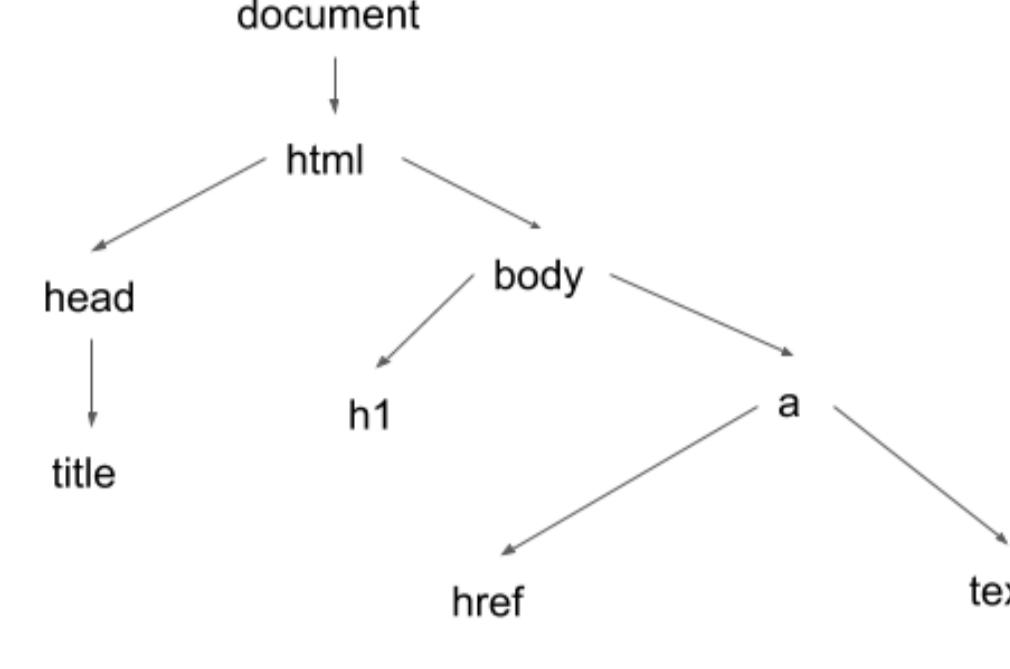


Рисунок 1. Схема DOM-дерева

Допустим, наша задача — найти ссылку, то есть HTML-элемент `a`. И поиск в глубину, и поиск в ширину начинаются с узла `document`. В `document` мы найдём единственного ребёнка — узел `html`. Так как ни `document`, ни `html` ссылками не являются, мы идём дальше и находим не один, а нескольких дочерних элементов. Разница между поиском в глубину и поиском в ширину заключается в том, каким образом мы продолжим поиск.

### Поиск в глубину

Мы находим одного из детей `html` — элемент `head`. Нам опять не повезло найти ссылку, поэтому пойдём вглубь, то есть в единственного ребёнка `head` — элемент `title`. `Title` также не соответствует требованиям поиска, и к тому же не имеет дочерних элементов.

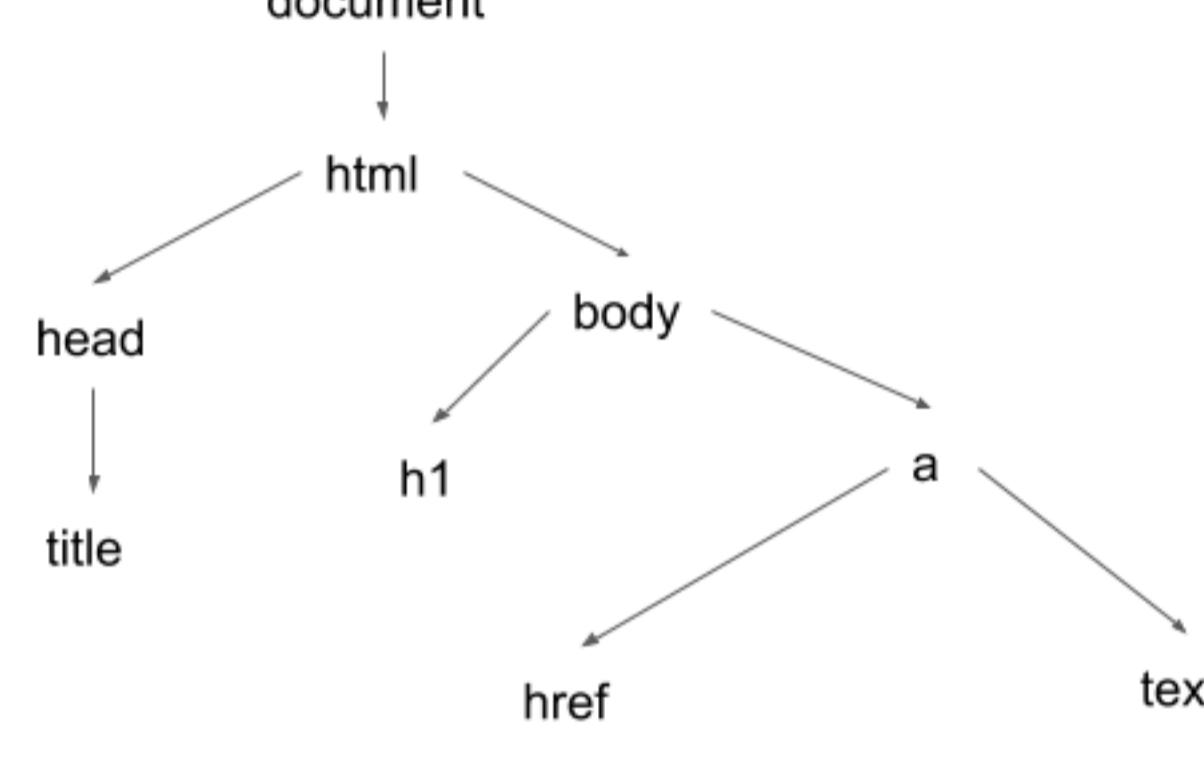


Рисунок 2. Поиск в глубину

Мы упёрлись в тупик. Самое время вернуться назад, в `html`, и пойти по второму доступному пути — в элемент `body`. Далее, по такой же цепочке, мы найдём элементы `h1` и, наконец, `a`.

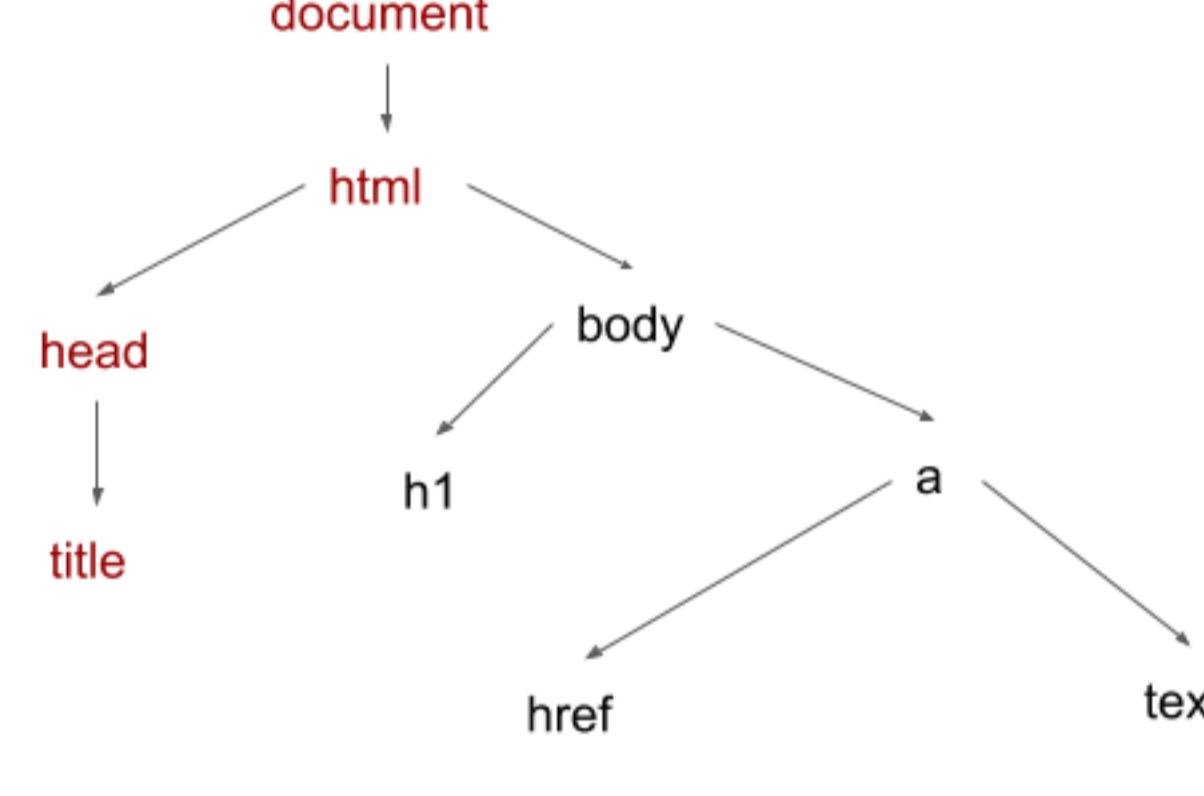


Рисунок 3. Поиск в глубину, продолжение

Когда имеется несколько доступных путей поиска, при поиске в глубину мы исследуем каждый из них до тех пор, пока не сталкиваемся с тупиком. Из тупика мы возвращаемся на последнюю развилку, исследуем её и так далее.

### Поиск в ширину

Теперь попробуем снова найти ссылку, но уже используя поиск в ширину. Вспомним, что начинаем мы из `document`, проваливаемся в `html` и сталкиваемся с распутьем.

В отличие от поиска в глубину, поиск в ширину сначала проверит всех детей элемента `html`, то есть `head` и `body`, на соответствие элементу-ссылке. Не найдя то, что ищет, он пойдёт в их непосредственно дочерние элементы и проверит их. Данный паттерн, проверка слой за слоем — это принцип работы поиска в ширину.

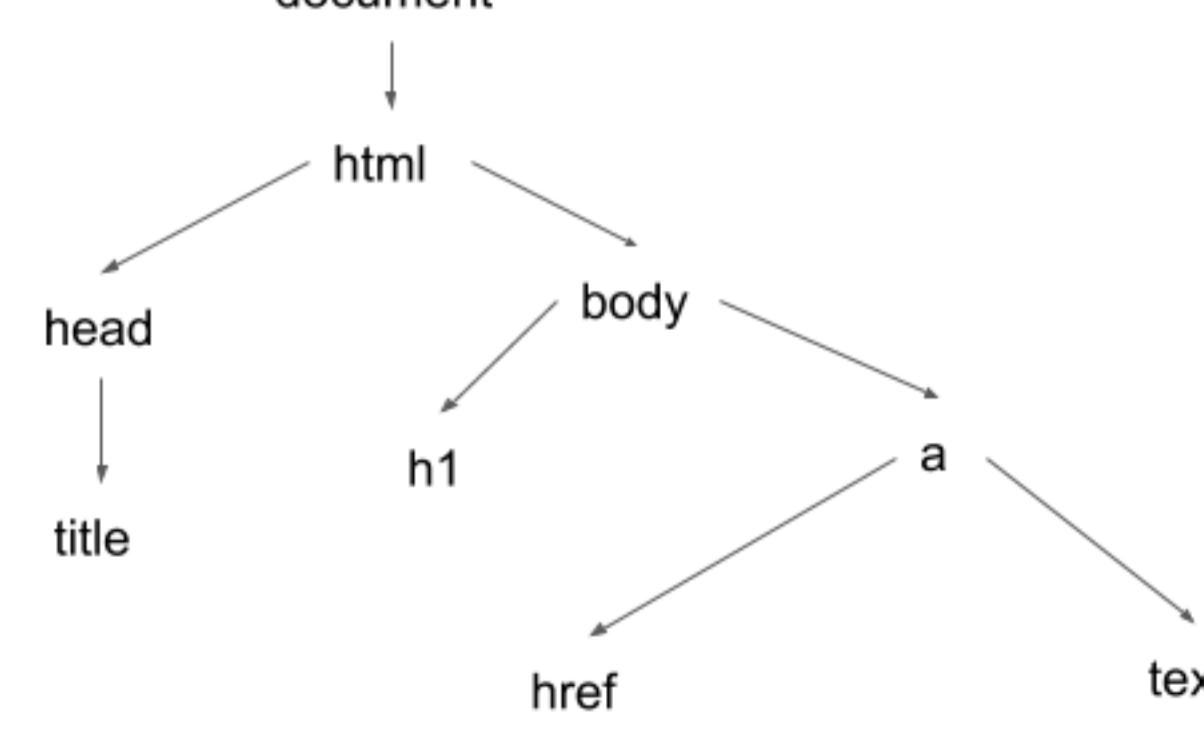


Рисунок 4. Поиск в ширину

### Резюме

Поиск в глубину и поиск в ширину — два подхода к обходу древовидных структур. Первый находит узел и проваливается в него, даже если у него есть соседи, которых можно проверить. Поиск в ширину, с другой стороны, сначала проверит все узлы одного уровня, и лишь затем будет спускаться в их непосредственных детях. JavaScript для обхода DOM использует поиск в глубину.

## Глава 4.4. Живые и неживые коллекции в JavaScript

Найти несколько DOM-элементов и получить к ним доступ из JavaScript можно разными способами: `querySelectorAll`, `getElementsByName`, `children` и так далее. В итоге в каждом случае будет возвращена коллекция — сущность, которая похожа на массив объектов, но при этом им не является, на самом деле это набор DOM-элементов. Стоит учесть, что фактически разные методы возвращают разные коллекции:

- `HTMLCollection` — коллекция непосредственно HTML-элементов.
- `NodeList` — коллекция узлов, более абстрактное понятие. Например, в DOM-дереве есть не только узлы-элементы, но также текстовые узлы, узлы-комментарии и другие, поэтому `NodeList` может содержать другие типы узлов.

При работе с DOM-элементами тип коллекции значительной роли не играет, поэтому для удобства будем рассматривать их как одну сущность — коллекцию.

Во время работы с коллекциями можно столкнуться с поведением, которое покажется странным, если не знать один нюанс — они бывают живыми (динамическими) и неживыми (статическими). То есть либо реагируют на любое изменение DOM, либо нет. Вид коллекции зависит от способа, с помощью которого она получена. Рассмотрим на примере.

### Разница между живыми и неживыми коллекциями

Допустим, в разметке есть список книг:

```
<ul class="books">
  <li class="book book--one"></li>
  <li class="book book--two"></li>
  <li class="book book--three"></li>
</ul>
```

Для взаимодействия с книгами получим с помощью JavaScript список всех нужных элементов. Чтобы в дальнейшем увидеть разницу между видами коллекций, используем разные способы поиска элементов — свойство `children` и метод `querySelectorAll`:

```
const booksList = document.querySelector('.books');
const liveBooks = booksList.children;

// Выведем все дочерние элементы списка .books
console.log(liveBooks);

const notLiveBooks = document.querySelectorAll('.book');

// Выведем коллекцию, содержащую все элементы с классом book
console.log(notLiveBooks);
```

Пока никакой разницы не видно. В обоих случаях `console.log` выведет одни и те же элементы. Но что, если попробовать удалить из DOM одну из книг?

```
const booksList = document.querySelector('.books');
const liveBooks = booksList.children;

// Удалим первую книгу
liveBooks[0].remove();
// Получим 2
console.log(liveBooks.length);
// Получим элемент book--two, который теперь стал первым в коллекции
console.log(liveBooks[0]);

const notLiveBooks = document.querySelectorAll('.book');

// Удалим первую книгу
notLiveBooks[0].remove();
// Получим 3
console.log(notLiveBooks.length);
// Получим ссылку на удалённый элемент book--one
console.log(notLiveBooks[0]);
```

В первом случае информация о количестве элементов внутри коллекции автоматически обновилась после удаления одного элемента из DOM — эта коллекция живая. Во втором случае в переменной `notLiveBooks` хранится первоначальное состояние коллекции, которое было актуально на момент вызова метода `querySelectorAll`. Эта коллекция неживая, она ничего не знает об изменении DOM. При этом доступна ссылка на удалённый элемент `book--one`, которого фактически больше нет в DOM.

### Другие способы получить коллекцию

Кроме `children` и `querySelectorAll` есть другие способы поиска DOM-элементов:

- `getElementsByTagName(tag)` — находит все элементы с заданным тегом,
- `getElementsByClassName(className)` — находит все элементы с заданным классом,
- `getElementsByName(name)` — находит все элементы с заданным атрибутом name.

Все эти методы могут встречаться в старом коде. Они возвращают живые коллекции и используются реже, потому что в большинстве случаев возможности живых коллекций не пригодятся. К тому же `querySelectorAll` в разы удобнее использовать из-за его универсальности.

### Как использовать

Для решения большинства задач можно ограничиться неживыми коллекциями. Но если нужно сохранить ссылку на реальное состояние DOM — понадобится живая коллекция. Это удобно в тех случаях, когда программе нужно постоянно манипулировать списком элементов, которые могут регулярно удаляться и добавляться. Хороший пример — задачи в системе учёта задач. С помощью живой коллекции можно хранить именно те задачи, которые фактически существуют в данный момент времени.

Структура и некоторые свойства коллекций имеют много общего с массивом. Например, у неё тоже есть свойство `length`, и элементы коллекции можно перебирать в цикле `for...of`, потому что это перечисляемая сущность. Но, как упоминалось ранее, коллекции не во всём похожи на обычные массивы. С коллекциями не работают такие методы массивов, как `push`, `splice` и другие. Для их использования нужно преобразовать коллекцию в массив — например, с помощью метода `Array.from`:

```
const booksList = document.querySelector('.books');
const books = booksList.children;

// Выведет обычный массив с элементами из коллекции books
console.log(Array.from(books));
```

При этом нужно помнить — массив статичен, поэтому при таком преобразовании теряются преимущества живых коллекций.

## Глава 4.5. DOM и разметка

Любое изменение, которое мы вносим в DOM, не является изменением разметки сайта. DOM может изменяться путём воздействия на него через JavaScript; либо же пользователем, путём взаимодействия с интерфейсом. Рассмотрим на примере: допустим, у нас на странице есть форма с двумя флажками, один из которых заранее отмечен:

```
<form>
  <label>
    <input type="checkbox" name="someCheckbox" value="1" checked>
    First checkbox
  </label>
  <label>
    <input type="checkbox" name="someCheckbox" value="2">
    Second checkbox
  </label>
</form>
```

First checkbox  Second checkbox

Рисунок 1. Форма с двумя флажками

На данной странице пользователь может каким-либо образом изменять значение данных флажков. Допустим, пользователь снял отметку с первого чекбокса и поставил во втором:

First checkbox  Second checkbox

Рисунок 2. Выбран другой флажок

При сохранении или отправке формы мы хотим узнать значение чекбокса, который отметил пользователь, для этого мы можем попробовать использовать следующий код, для того, чтобы найти чекбокс с нужным именем и атрибутом `checked`:

```
document.querySelector(`input[name="someCheckbox"] [checked]`).value;
```

Однако, это неверно – когда пользователь взаимодействует с разметкой, используя элементы управления, то он изменяет DOM. То же самое мы можем делать из JavaScript. А в селекторе, который используется в коде выше, идёт привязка к разметке, и в данной ситуации такой селектор вернёт первое поле ввода, так как только у первого поля ввода в разметке присутствует атрибут `checked`:

```
> document.querySelector('input[name="someCheckbox"] [checked]').value
< "1"
```

Рисунок 3. Поиск выбранного флажка

Чтобы такого не происходило, мы можем использовать псевдоклассы (как в CSS):

```
document.querySelector(`input[name="someCheckbox"] :checked`).value;
```

Тогда селектор найдёт поле ввода, которое выбрано на данный момент:

```
> document.querySelector('input[name="someCheckbox"] :checked').value
< "2"
```

Рисунок 4. Поиск элемента в DOM

Помните, что любая манипуляция с DOM не меняет разметку. В этом плане ввести в заблуждение могут инструменты разработчика. Если проинспектировать страницу, то во вкладке «Elements» можно увидеть, казалось бы, разметку:

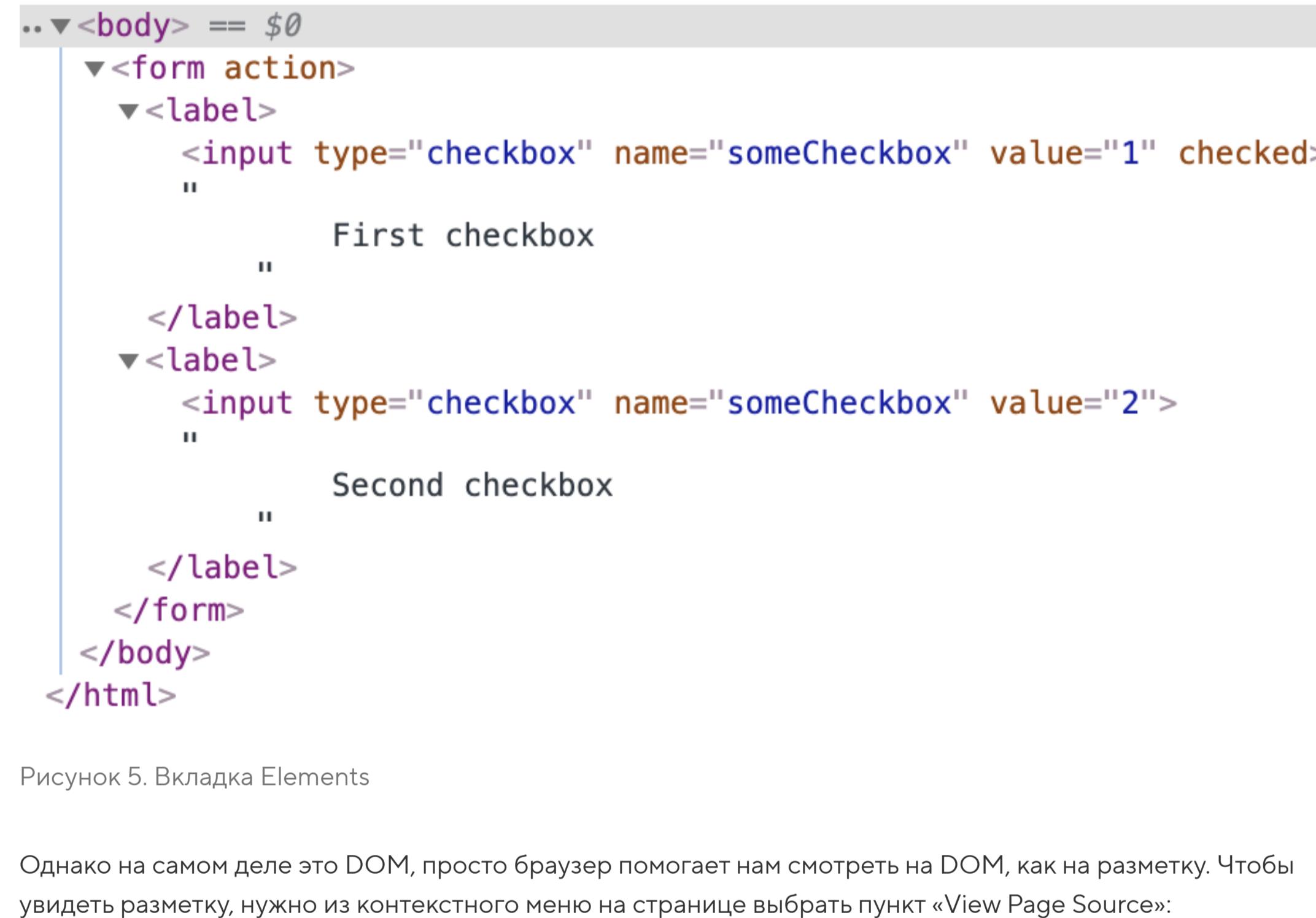


Рисунок 5. Вкладка Elements

Однако на самом деле это DOM, просто браузер помогает нам смотреть на DOM, как на разметку. Чтобы увидеть разметку, нужно из контекстного меню на странице выбрать пункт «View Page Source».

First checkbox  Second checkbox

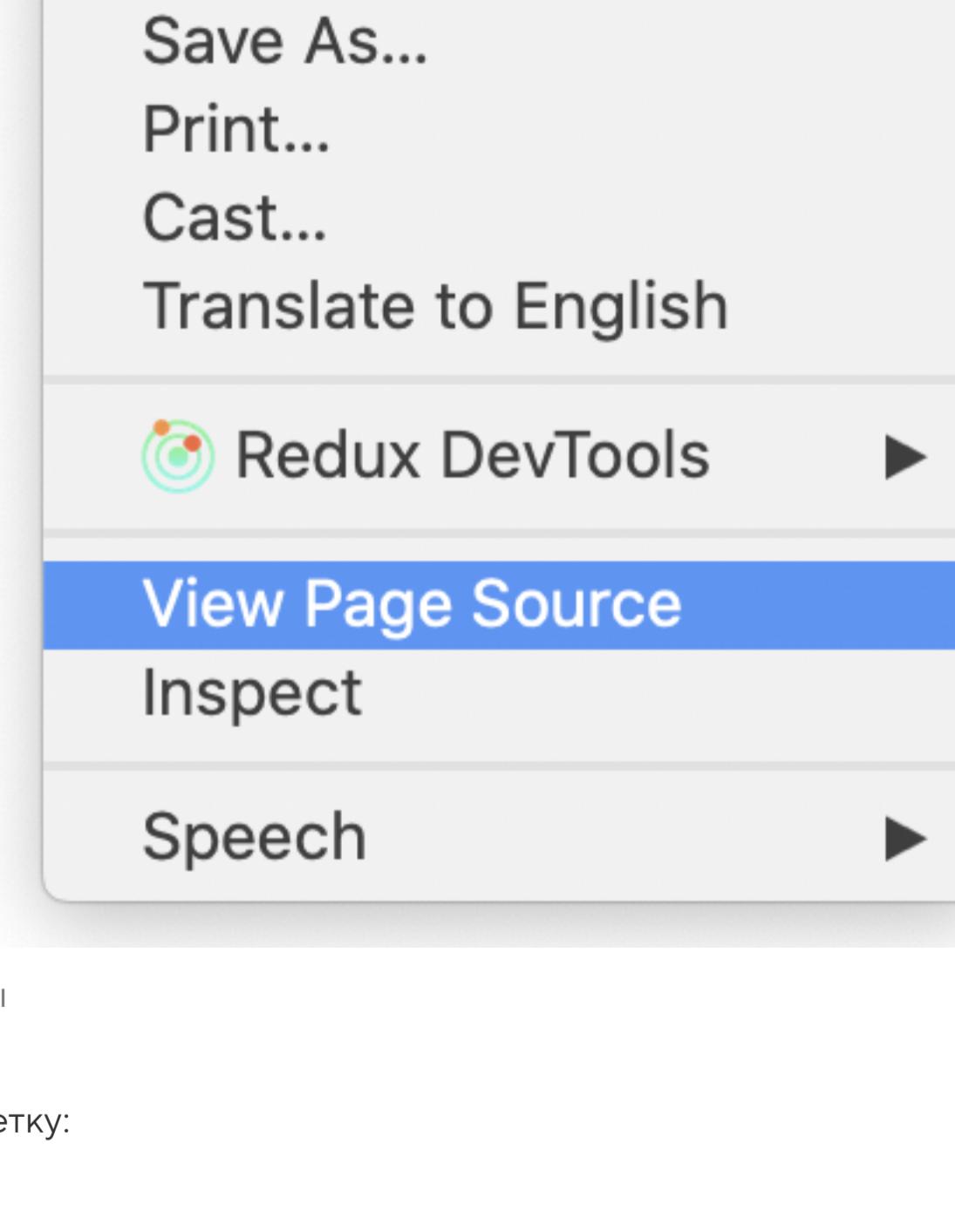


Рисунок 6. Открыть исходный код страницы

Тогда в открывшемся окне мы увидим разметку:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="UTF-8">
5     <title>Title</title>
6   </head>
7   <body>
8     <form action="">
9       <label>
10         <input type="checkbox" name="someCheckbox" value="1" checked>
11         First checkbox
12       </label>
13       <label>
14         <input type="checkbox" name="someCheckbox" value="2">
15         Second checkbox
16       </label>
17     </form>
18   </body>
19 </html>
```

Рисунок 7. Исходный код страницы

Разметка для DOM – это начальное состояние, то есть то состояние, которое появляется после загрузки страницы. После этого в ход вступают пользователь, JavaScript, какие-либо сторонние библиотеки, и DOM может измениться и уже не соответствовать разметке.

## Глава 4.6. Шаблоны и данные

Начнём с определений. **Шаблон** – некоторая оболочка для данных, разметка, любой способ отобразить информацию. Шаблон никогда не несёт содержательной информации.

**Данные** – информация, которую вводит пользователь, присыпает сервер или которая может быть сгенерирована компьютером.

Данные не должны повторять шаблон, они должны описывать параметры сущностей, которыми мы оперируем. Простой способ отделить шаблон от данных – попробовать изменить одно или другое.

Например, использовать иной способ отображения данных (отобразить товары в линейку вместо списка) или изменить отображаемую информацию (описать не утюг, а пылесос). Допустим, у нас есть структура, описывающая логотип:

```
const header = {
  logo: {
    src: 'logo.png',
    width: 100,
    height: 30
  }
};
```

С первого взгляда можно подумать, что это данные, однако, такую информацию неправильно хранить как данные. Эта информация описывает логотип, расположенный в шапке, она не приходит с сервера, не вводится пользователем, возможно, эта информация никогда не поменяется. Поэтому данная информация не является данными – это шаблон, который описывает, как некоторая сущность должна выглядеть. Данными могут быть: название компании, адрес, телефон и тому подобное.

### Зачем отделять данные и шаблоны

Допустим, что у нас есть список некоторых продуктов интернет-магазина:

```
const products = [
  {name: 'Утюг'},
  {name: 'Чайник'},
  {name: 'Пылесос'},
  {name: 'Стиральная машина'},
  {name: 'Кухонный комбайн'},
  {name: 'Автомобиль'}
];
```

Дизайнер говорит нам, что в интерфейсе мы должны для некоторых пользователей показывать товары списком, а для других (кто хочет) сеткой. Неужели нам придётся заводить два одинаковых списка продуктов? Нет! Как раз список продуктов, приведённый выше, является данными, а способы показа товаров являются шаблоном.

Данные не должны меняться в зависимости от того, каким образом они должны отображаться. Шаблон отвечает за то, куда вставить данные – в элемент списка или элемент сетки.

### Создание DOM элементов

DOM-элементы можно создать несколькиими способами:

- **На основе разметки** – в специальные места разметки в тексте разметки подставляются данные;
- **На основе строк**;
- **Компилируемые шаблоны** – использование сторонних библиотек, способных переводить некоторый язык в разметку;
- **На основе DOM-API**:
  - На основе шаблонного элемента (`template` из WebComponents);
  - С помощью обёрток над шаблонами (Incremental DOM).

Рассмотрим разделение на шаблон и данные на примере. Допустим, у нас на странице есть следующий блок, показывающий нескольких волшебников:



Рисунок 1. Несколько волшебников

Как можно заметить, волшебники выглядят однотипно, то есть способ их отображения един и не меняется. Однако, некоторые параметры у них отличаются: имя и цвет мантии.

В данном примере разметка для каждого из волшебников является шаблоном и не несёт в себе информации, а лишь служит обёрткой для отображения данных. Разметка для каждого из волшебников (шаблон) имеет следующий вид:

```
<template id="similar-wizard-template">
  <div class="setup-similar-item">
    <div class="setup-similar-content">
      <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 62 86" class="setup-similar-wizard">
        <g class="wizard">
          <use xlink:href="#wizard-coat" class="wizard-coat"></use>
          <use xlink:href="#wizard-head" class="wizard-head"></use>
          <use xlink:href="#wizard-eyes" class="wizard-eyes"></use>
          <use xlink:href="#wizard-hands" class="wizard-hands"></use>
        </g>
      </svg>
    </div>
    <p class="setup-similar-label"></p>
  </div>
</template>
```

В данном шаблоне нет конкретных данных, лишь способ отображения. В данный шаблон в нужные места можно подставить данные, нужные нам. Данные для нашего примера будут иметь следующий вид:

```
const wizards = [
  {
    name: 'Дамблдор',
    coatColor: 'rgb(241, 43, 107)'
  },
  {
    name: 'Волдеморт',
    coatColor: 'rgb(215, 210, 55)'
  },
  {
    name: 'Доктор',
    coatColor: 'rgb(101, 137, 164)'
  },
  {
    name: 'Гарри',
    coatColor: 'rgb(127, 127, 127)'
  }
];
```

Как видно, данные описывают параметры, изменяющиеся у различных волшебников.

### Резюме

Важно уметь выделять данные и шаблоны. К данным можно отнести ту информацию, которая не влияет на способ отображения, соответственно, к шаблонам нужно отнести то, что описывает, как некоторые однотипные элементы могут быть отображены.

Простой способ отдалить шаблон от данных – попробовать заменить их: изменить способ отображения данных или отображаемую информацию.

## Глава 5.1. Синхронные и асинхронные операции

Когда рассказывают про программы и алгоритмы, их часто сравнивают с инструкциями, написанными для человека. Например, вот так выглядит инструкция по приготовлению каши:

1. Насыпал кашу.
2. Включил плиту.
3. Помешиваю, помешиваю, помешиваю кашу.
4. Каша готова.

Или схематично:

### Синхронное приготовление завтрака



Рисунок 1. Синхронное приготовление завтрака

Человек, следующий этой инструкции, будет выполнять её пункты по очереди. Начнёт с первого пункта, завершив его, перейдёт ко второму, и так далее. Однако, есть кашу всухомятку довольно грустно, поэтому добавим к ней чай. В синхронной схеме чай мы приготовим после того, как каша будет готова:

3. ...
4. Каша готова.
5. Налил воды в чайник.
6. Поставил чайник на плиту.
7. Дождался, пока вода закипит.
8. Залил чай кипятком.
9. Подождал, пока чай заварится.
10. Чай готов.

Что можно сказать про эту инструкцию? Ну, во-первых, она действительно позволяет приготовить завтрак. Во-вторых, человеку, который решит ей воспользоваться, придётся встать пораньше, если он рассчитывает успеть на работу. Инструкция довольно долгая. Сначала нужно дождаться, пока приготовится каша, потом — пока вскипит чайник. Да и к моменту, когда чай будет готов, каша наверняка уже остынет. Разумно ли это? Конечно же, нет! Заменим кастрюлю на мультиварку, и чайник на электрический, тогда ждать понадобится намного меньше, потому что готовить можно одновременно.

Перепишем инструкцию с учётом новой техники:

1. Насыпал кашу.
2. Включил мультиварку.
3. Налил воды в чайник.
4. Включил чайник.
5. Приготовил тосты.
6. Накрыл на стол.
7. ...

### Асинхронное приготовление завтрака



Рисунок 2. Асинхронное приготовление завтрака

В этом случае пока каша готовится, мы успеем накрыть на стол. Пока мы накрываем на стол, вероятно, закипит чайник — мы сможем заварить чай, а тут и каша готова. Приятного аппетита.

### Асинхронное приготовление завтрака (завершение)

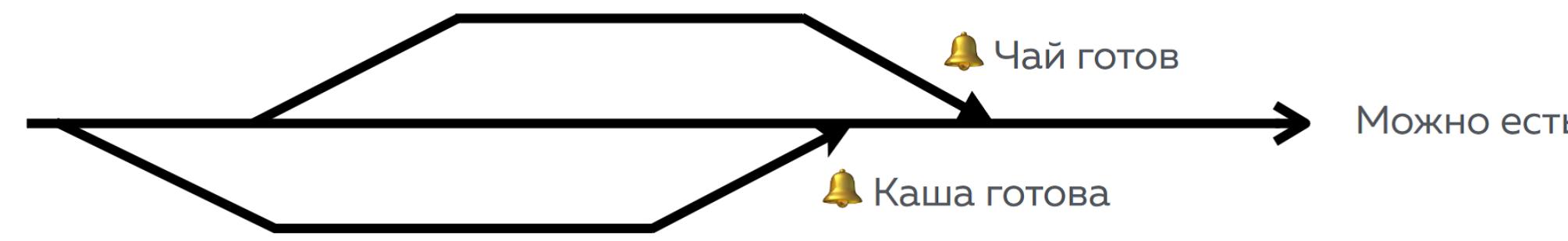


Рисунок 3. Асинхронное приготовление завтрака (завершение)

Согласитесь, оптимальная схема, и поспать с утра можно подольше. Однако мы незаметно отказались от одного свойства, которым обладала предыдущая инструкция — от последовательности.

Мы не можем знать наверняка, что мультиварка подаст сигнал раньше, чем чайник вскипит. Обратное тоже с уверенностью утверждать нельзя. Скорее всего, каша будет готовиться дольше, но может случиться по-всякому. Вдруг у исполнителя очень мощная мультиварка и очень старый чайник. А значит — мы не знаем, в каком порядке и когда завершится то или иное действие.

Второе важное различие: за приготовлением каши в кастрюле целиком и полностью следим мы, и только мы решаем, когда каша будет готова. В случае с мультиваркой готовность каши зависит от некоего внешнего фактора (мультиварки).

Если говорить на программистском языке, приготовление каши в кастрюле — череда синхронных операций. Синхронные операции производятся одна за другой в предсказуемом порядке. А приготовление каши в мультиварке включает в себя асинхронную операцию. Особенность асинхронных операций, что мы не можем предугадать, в каком порядке они завершатся и сколько времени займёт каждая.

«Но какое это имеет отношение к веб-программированию?» — спросит нетерпеливый читатель. Самое прямое: браузер — это тоже мультиварка. Разумеется, в переносном смысле. Браузер не умеет варить овсянку (по крайней мере, текущие веб-стандарты этого не предусматривают), зато он умеет выполнять асинхронные операции.

Мы даём ему команду загрузить файл, но не знаем, в какой момент произойдёт загрузка. Мы создаём на странице кнопку, но неизвестно, когда пользователь на неё нажмёт. Точно так же, как и при варке каши, мы можем отдать команду и заниматься своими делами. И когда всё будет готово, браузер подаст сигнал — опять же, совсем как мультиварка.

## Глава 5.2. Функции обратного вызова (колбэки)

При изучении программирования мы привыкаем мыслить последовательно: строки кода выполняются по порядку. Для многих языков это утверждение верно на 100%, но всё начинает меняться, когда речь заходит про асинхронное программирование. Самым простым и часто используемым способом достижения асинхронности в коде являются колбэки. Это одна из важнейших тем программирования на JavaScript, ни одна более-менее серьёзная программа не обойдётся без применения колбэк-функций.

► Ещё раз, что такая колбэк-функция, если забыли

### Как писать код для колбэков

Посмотрим на колбэки с практической стороны. Выше мы сказали, что колбэки неразрывно связаны с асинхронностью и позволяют «запланировать» действие, которое будет совершено после выполнения какого-то другого, возможно длительного действия. Пример с заказом пиццы это прекрасно иллюстрирует. Давайте посмотрим, как это может выглядеть в коде, но для начала взглянем на синхронный код:

```
// Приготовление пиццы. Длительный процесс
const newPizza = makePizza('pepperoni');
// Читаем книгу пока готовят пиццу
readBook();

// Съедаем пиццу
eatPizza(newPizza);
```

Что в этом коде больше всего бросается в глаза? Правильно — последовательность. Здесь представлен синхронный код, который будет выполняться последовательно:

1. Мы ждём, пока для нас приготовят пиццу «Пепперони».
2. Затем мы читаем книгу.
3. Наконец-таки откладываем книгу в сторону и ужинаем пиццей.

Проблема видна невооружённым глазом — пока готовится пицца, мы вынуждены ждать и ничего не делать. Стока `readBook()` будет выполнена только **после** приготовления пиццы. Фактически мы начнём читать книгу после приготовления пиццы, а не **во время** готовки.

Само собой, в реальном мире вместо выпекания пиццы может быть любой долгий процесс, например, запрос на получение данных с сервера.

Такой запрос не выполняется мгновенно: браузеру понадобится время, чтобы найти IP-адрес сервера, установить соединение, передать запрос, дождаться ответа и т. д. Все эти действия занимают разное количество времени. Временные задержки будут постоянно отличаться и зависеть от скорости соединения с сетью, времени выполнения запроса на сервере и некоторых других факторов.

Синхронные запросы к серверу будут блокировать дальнейшее выполнение веб-приложения, и это уже очень плохо. Представьте, что каждый раз при отправке запроса к серверу интерфейс вашего приложения становится полностью недоступным.

Эту проблему решает асинхронность, и длительные операции лучше выполнять именно асинхронно. В этом варианте мы как бы откладываем длительную операцию «на потом» и вместо ожидания завершения выполняем другой код. В этой схеме прозрачно всё, кроме вопроса: «Как выполнить код после завершения асинхронной операции?». Ответ прост — функции обратного вызова.

В JavaScript функции являются объектами высшего порядка. Это означает, что функции можно передавать в другие функции в виде параметров или возвращать в виде результата выполнения.

Рассмотрим пример:

```
const foo = function () {
  return 'Hello, world!';
}

// Вызываем функцию и выводим результат в консоль
console.log(foo()); // Hello, world

// Выводим функцию в консоль без вызова
console.log(foo); // f () { return 'Hello, world!'; }
```

В первом случае мы вызываем функцию `foo` при помощи круглых скобок и выводим результат выполнения в консоль. Во втором примере мы не делаем вызов функции (обратите внимание на отсутствие круглых скобок), и в консоль выводится содержимое функции. Выходит, нам ничего не мешает передать функцию в виде параметра для других функций:

```
const runIt = function (fn) {
  return fn(); // Вызываем функцию, переданную в качестве параметра
}

console.log(runIt(foo)); // Hello, world
```

Мы передали функцию `foo` в виде параметра и вызывали её внутри функции `runIt`. Вызов функции мы сделали стандартным образом — применяя круглые скобки.

Что в итоге? Мы передали ссылку на функцию в виде параметра и вызвали её внутри другой функции. В этом и заключается идея колбэков: мы передаём в виде параметров функции, которые будут вызваны «когда-нибудь потом».

### И снова пицца

Вернёмся к примеру с приготовлением пиццы. Попробуем поэкспериментировать с кодом и перевести его на асинхронные рельсы. Напомню, наша задача — попросить приготовить пиццу, и читать книгу, пока пицца не будет готова.

```
const makePizza = function (title, cb) {
  console.log(`Заказ на приготовление пиццы "${title}" получен. Начинаем готовить...`);

  // setTimeout —строенная функция для отложенного вызова колбэка.
  // Интерфейс у неё прост: первым аргументом нужно передать колбэк, а вторым — задержку
  // (таймут) в миллисекундах.
  // Более с setTimeout вы познакомитесь позже, в отдельном материале раздела.
  setTimeout(cb, 3000);
}

const readBook = function () {
  console.log('Читаю книгу "Колдун и кристалл..."');
}

const eatPizza = function () {
  console.log('Ура! Пицца готова, пора подкрепиться.');
}

makePizza('Пепперони', eatPizza);
readBook();
```

Это рабочий код, попробуйте выполнить его в консоли и посмотреть на результат вывода. Он будет таким:

```
Заказ на приготовление пиццы «Пепперони» получен. Начинаем готовить...
Читаю книгу «Колдун и кристалл...»

// Здесь будет пауза

Ура! Пицца готова, пора подкрепиться.
```

Функция `makePizza` выполняется мгновенно, и сразу за ней последовал вызов `readBook`. Пока мы читали книгу, приготовилась пицца, и произошёл вызов функции `eatPizza` из функции `makePizza`.

### Резюме

Как видите, ничего сверхъестественного в колбэках нет. Это обычная функция, которая будет выполнена не сейчас, а когда-нибудь потом. «Когда-нибудь» — не преувеличение. Мы не можем сказать, в какой момент времени это случится, но можем сказать, после какой именно функции — после выполнения функции приготовления пиццы.

## Глава 5.3. Функции обратного вызова на практике

В прошлом разделе мы на примере заказа пиццы разобрались с тем, зачем нужны колбэки. Сейчас посмотрим на то, как работают запросы к серверу, как получать данные через AJAX.

Практически любое приложение на JavaScript взаимодействует с сервером. Нам постоянно приходится получать данные с помощью методологии AJAX. Попробуем запросить информацию о произвольном аккаунте с GitHub и вывести её в консоль. Такая программа может выглядеть так:

```
const loadData = function(url, cb) {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', url);
  xhr.responseType = 'json';
  // После получения данных выполним cb.
  xhr.addEventListener('load', cb);
  xhr.send();
}

loadData('https://api.github.com/users/AntonovIgor', function (evt) {
  const response = evt.currentTarget.response;
  console.log(response);
});
```

В этом коде мы делаем запрос к серверу с помощью объекта `XMLHttpRequest`. Для удобства мы создали функцию `loadData`, которая принимает два параметра: адрес сервера, который отдаст нам набор информации, и функцию обратного вызова. Вызов этой функции произойдёт сразу после получения данных от сервера. Если вас смущило наименование параметра `cb` — не переживайте. Это общепринятый вариант обозначения функций обратного вызова (сокращение от callback). При виде `cb` становится очевидно, что в этот параметр должна передаваться ссылка на функцию.

Мы не будем детально рассматривать содержимое функции `loadData`. Подробно про взаимодействие с сервером расскажем в одном из следующих разделов учебника. Если коротко, то эта функция делает запрос к серверу и вызывает нашу функцию после получения ответа.

Обратите внимание, как мы вызываем функцию `loadData`. В первый параметр мы передаём адрес сервера, а во второй — ссылку на функцию. В этот раз мы не описывали функцию отдельно, а сделали это прямо «в параметре» — создали анонимную функцию. Подобный подход часто применяется, когда функция нужна один раз. В таких ситуациях нет смысла писать отдельный код — достаточно создать анонимную функцию в нужном месте.

Наша функция сработает сразу после получения ответа от сервера (произойдёт событие `load`) и выведет ответ сервера в консоль.

### Колбэки повсюду

Даже если вам не требуется работать с сервером, довольно высоки шансы столкнуться с функциями обратного вызова.

Не верите? Хорошо, а как насчёт стандартных возможностей вроде перебирающих методов массивов: `.forEach()`, `.every()`, `.some()`, `.reduce()`, `.filter()`, или функции сортировки `.sort()`, или метода `.addEventListener()` (второй аргумент — функция, которая будет вызвана при наступлении события). Список можно продолжать до бесконечности.

Всякий раз, когда вы пишете код, похожий на этот, вы применяете функции обратного вызова:

```
[1, 2, 3, 4].forEach(function(it) {
  console.log(it)
});
```

Аналогично с установкой обработчиков событий. Каждый раз, когда вы подписываетесь на событие с помощью `.addEventListener`, через её второй параметр вы определяете функцию обратного вызова, которая сработает при наступлении события. В мире JavaScript колбэки повсюду.

### Как передать параметры в колбэк-функцию

Это очень важный вопрос, и в начале понимания сути колбэк-функций он может загнать в тупик.

Проблема в том, что мы привыкли передавать параметры явно: пишем имя функции, открываем круглые скобки и передаём значения. Этот подход применяется постоянно, но с колбэк-функциями дело обстоит иначе. Если мы укажем круглые скобки, то произойдёт вызов функции, а нам требуется передать ссылку на неё. Как поступить в этом случае?

Для этого нужно вспомнить про замыкания. Функции в JavaScript могут возвращать в качестве результата выполнения другие функции. А уже у них будет доступ к родительским областям видимости.

С таким подходом легко решить задачу передачи параметров в колбэк-функцию.

Код заказа пиццы, который мы написали в первой части, выглядел так:

```
const makePizza = function (title, cb) {
  console.log(`Заказ на приготовление пиццы «${title}» получен. Начинаем готовить...`);
  setTimeout(cb, 3000);
}
```

```
const readBook = function () {
  console.log(`Читаю книгу «Колдун и кристалл»...`);
```

```
const eatPizza = function () {
  console.log(`Ура! Пицца готова, пора подкрепиться.`);
```

```
makePizza('Пепперони', eatPizza);
readBook();
```

Давайте модифицируем его и добавим для функции `eatPizza` параметр `drink`, через который будем передавать напиток:

```
const makePizza = function (title, cb) {
  console.log(`Заказ на приготовление пиццы «${title}» получен. Начинаем готовить...`);
  setTimeout(cb, 3000);
}
```

```
const readBook = function () {
  console.log(`Читаю книгу «Колдун и кристалл»...`);
```

```
const eatPizza = function (drink) {
  return function() {
    console.log(`Ура! Пицца готова, пора подкрепиться и запить ${drink}.`);
```

```
}
```

```
makePizza('Пепперони', eatPizza('Coca-Cola'));
readBook();
```

Первое, что мы сделали — внесли изменения в функцию `eatPizza`. Теперь она принимает параметр `drink` и возвращает новую функцию. В теле новой функции происходит вывод информации в консоль. Помимо текста, который у нас был, мы добавили вывод информации о напитке (`drink`). Теперь самое интересное. Функция `eatPizza` перестаёт быть функцией обратного вызова, вместо неё эту роль будет выполнять функция, которую возвращает `eatPizza`.

Мы изменили тело функции, и теперь нам требуется обновить вызов `makePizza`. Вторым параметром мы указываем не ссылку на `eatPizza`, а вызов функции, передав информацию о напитке. Получается, на место второго параметра будет передана новая функция, полученная в результате выполнения `eatPizza`.

Если представленный пример вызывает затруднение — обязательно перечитайте теорию замыканий.

### Резюме

Колбэк-функции просты в применении и открывают много возможностей, но с большой силой приходит большая ответственность. Легко потеряться во вложенности колбэк-функций и познать все прелести [эда функций обратного вызова](#). Любым подходом нужно пользоваться с осторожностью. Как не попасть в ад обратных вызовов, мы поговорим в одной из ближайших статей.

## Глава 5.4. Действия браузера по умолчанию

Возможно, вам уже когда-то встречалась в коде такая строка — `evt.preventDefault()`. Например, в интерактивных курсах по JavaScript. Давайте подробно разберём, зачем она нужна.

При разработке таких типичных элементов интерфейса, как форма или попап, часто нужно изменить поведение браузера по умолчанию. Допустим, при клике по ссылке мы хотим, чтобы открывался попап, но вместо этого браузер будет автоматически переходить по адресу, указанному в атрибуте `href`. Или вот другая проблема — мы хотим перед отправкой формы проверять корректность введённых данных, но после нажатия на кнопку `submit` форма каждый раз будет отправляться на сервер, даже если там куча ошибок. Такое поведение браузера нам не подходит, поэтому мы научимся его переопределять.

### Объект события и метод `preventDefault`

Событие — это какое-то действие, произошедшее на странице. Например, клик, нажатие кнопки, движение мыши, отправка формы и так далее. Когда срабатывает событие, браузер создаёт объект события `Event`. Этот объект содержит всю информацию о событии. У него есть свои свойства и методы, с помощью которых можно эту информацию получить и использовать. Один из методов как раз позволяет отменить действие браузера по умолчанию — `preventDefault()`.

`Event` можно передать в функцию-обработчик события и в ней указать инструкции, которые должны быть выполнены, когда оно сработает. При передаче объекта события в обработчик обычно используется сокращённое написание — `evt`.

#### Пример: когда ссылка — не ссылка

Ранее мы уже говорили о попапе, который должен появляться при клике на ссылку — давайте разберём этот кейс на практике. Так будет выглядеть разметка в упрощённом виде:

```
<a class="click-button" href="pop-up.html">Клик</a>

<div class="content">
  <!-- Здесь содержимое попапа -->
</div>
```

Мы хотим при клике на ссылку `click-button` добавлять элементу с классом `content` класс `show`. Он сделает попап видимым, поменяв значение свойства `display` с `none` на `block`. Напишем логику добавления этого класса с помощью JavaScript:

```
// Находим на странице кнопку и попап
const button = document.querySelector('.click-button');
const popup = document.querySelector('.content');

// Навешиваем на кнопку обработчик клика
button.onclick = function (evt) {
  // Отменяем переход по ссылке
  evt.preventDefault();

  // Добавляем попапу класс show, делая его видимым
  popup.classList.add('show');
};
```

Если мы уберём строку `evt.preventDefault()`, вместо попапа откроется отдельная страница `pop-up.html`, адрес которой прописан в атрибуте `href` у ссылки. Такая страница нужна, потому что мы хотим, чтобы вся функциональность сайта была доступна, если скрипт по какой-то причине не будет загружен. Именно поэтому мы изначально реализовали кнопку с помощью тега `a`, а не `button`. Но у нас с JavaScript всё в порядке, поэтому вместо отдельной страницы мы открыли попап, отменив действие браузера по умолчанию.

#### Пример: проверка формы перед отправкой

Разберём следующий кейс — отправку формы при нажатии на кнопку `submit`. Допустим, мы хотим перед отправкой проверять введённые данные, потому что в поле ввода обязательно должно быть значение 'Кекс' и никакое другое. Разметка формы:

```
<form class="form" action="#" method="post">
  <input class="name" type="text" id="name" name="name">
  <label for="name">Введите имя</label>

  <button type="submit">Готово!</button>
</form>
```

При нажатии на кнопку «Готово» сработает событие отправки формы `submit`, и форма отправится вне зависимости от корректности введённого значения, поэтому мы должны перехватить отправку.

```
// Находим на странице форму и инпут
const form = document.querySelector('.form');
const name = document.querySelector('.name');

// Навешиваем на форму обработчик отправки
form.onsubmit = function(evt) {
  // Проверяем введённое значение на соответствие
  if (name.value !== 'Кекс') {
    // Если значение не подходит, отменяем автоматическую отправку формы
    evt.preventDefault();
    // И выводим предупреждение в консоль
    console.log('Вы не Кекс!');
  }
};
```

Здесь мы не дали отправить форму при неверно введённом значении. Но если всё в порядке, условие не выполнится, и форма будет отправлена как обычно.

### Неотменяемые события

Не для всех событий можно отменить действие по умолчанию. Например, событие прокручивания страницы `scroll` проигнорирует попытки отменить его. Чтобы узнать, можно отменить действие по умолчанию или нет, нужно обратиться к свойству `cancelable` объекта `Event`. Оно будет равно `true`, если событие можно отменить, и `false` — в обратном случае.

```
document.onscroll = function(evt) {
  // В консоль выведется false
  console.log(evt.cancelable);
  // Отмена не сработает
  evt.preventDefault();
};
```

В статье мы разобрали базовые примеры, когда может понадобиться отмена действия браузера по умолчанию. В реальной разработке вы будете сталкиваться с такой необходимостью довольно часто — при сложной валидации форм, предотвращении ввода пользователем неверных символов, создании самописного меню вместо стандартного (при клике правой кнопкой мыши) и так далее.

## Глава 5.5. Фазы событий

Работать с событиями несложно: достаточно подписать на нужное событие с помощью `addEventListener` и подготовить функцию обратного вызова с кодом. При наступлении события эта функция выполнится. Рассмотрим небольшой пример:

```
<body>
  <form>
    <button type="submit" />
  </form>
</body>
```

При нажатии на кнопку событие `click` произойдёт почти на всех элементах. Порядок того, на каком элементе оно произойдёт первым, зависит от фазы события. В настоящее время в стандарте закреплено три фазы: захват, целевое событие и всплытие. Фаза целевого события, как правило, не используется.

### Фаза захвата

Другие названия: погружение, перехват, capturing. На этой фазе, когда пользователь нажмёт на кнопку, событие произойдёт сначала на `body`, затем на `form`, и только потом оно опустится до `button`.

Отследить это можно, навесив обработчики одного и того же события (например, `click`) на каждый из тегов и добавив к ним вывод `console.log()` или `alert` — код, позволяющий идентифицировать элемент. В этом случае на стадии захвата сначала выведется последовательно результат обработчика на `body`, следом `form`, и в конце `button`.

```
<body>
  <form>
    <button type="submit" />
  </form>
</body>
```

Рисунок 1. Фаза захвата

### Фаза всплытия

Вторая фаза называется всплытие (bubbling). Это та самая фаза, которая используется по умолчанию в `addEventListener` и в устаревшем способе добавления обработчика через `onclick`.

Здесь наступление событий работает с точностью наоборот. Когда пользователь кликает по кнопке, сначала отработает обработчик на самой кнопке, затем на её первом родителе `form` и только потом дойдёт очередь до `body`. После этого (вне примера), событие поднимется до `html`, `document`, а в некоторых случаях и `window`.

```
<body>
  <form>
    <button type="submit" />
  </form>
</body>
```

Рисунок 2. Фаза всплытия

### Порядок фаз

- Захват
- Событие на элементе (испускание, отправка, dispatch)
- Всплытие (если возможно)

### Резюме

Важно запомнить: захват произойдёт в любом случае, а всплытие только если возможно. Всплытие возможно для почти всех событий, кроме `focus/onfocus`, `blur`, `mouseleave`, `mouseenter`.

## Глава 5.6. Делегирование событий

Делегирование событий — это подход, который сокращает количество однотипных обработчиков в коде.

Разберёмся на примере. Представим, что наша задача звучит так: обрабатывать клик на каждом элементе списка одинаковым обработчиком. Можно пройтись по каждому элементу и добавить обработчики:

```
<ul>
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
  <li>...</li> <!-- onListItemClick() -->
</ul>
```

```
// script.js
const onListItemClick = function (evt) {
  // Действия
}
```

### Минусы решения

- Для каждого обработчика выделяется место в оперативной памяти. Если элементов станет много, то количество однотипных обработчиков может повлиять на быстродействие сайта
- Если в список добавятся новые элементы, то придётся за ними следить и вешать на них обработчики вручную, что увеличивает количество поддерживающего кода
- Хороший программист следует принципу **DRY** — Don't repeat yourself, избегает лишних повторений и не плодит одинаковые сущности без надобности

### Делегирование

Вместо добавления одинаковых обработчиков на каждый элемент списка, добавим один обработчик на родительском элементе, то есть родителю делегируется (поручается) обработка событий его дочерних элементов:

```
<ul> <!-- onListClick() -->
  <li>...</li>
  <li>...</li>
  <li>...</li>
  <li>...</li>
  <li>...</li>
</ul>
```

```
// script.js
const onListClick = function (evt) {
  if (evt.target.nodeName === 'LI') {
    // Действия
  }
}
```

### Как это работает

1. В момент клика на элементе `li`, ссылка на него записывается в свойство `target` интерфейса Event
2. Начинается стадия всплытия
3. Всплытие доходит и срабатывает на `ul`. Обработчик проверяет, является ли `evt.target` элементом списка и выполняет код.

### Резюме

Используя делегирование событий, мы избавились от однотипных обработчиков и сделали нашу вёрстку легко масштабируемой: можем добавить сколько угодно дополнительных элементов списка и ничего не сломается.

## Глава 6.1. Области видимости переменных и функций

Механизм областей видимости в JavaScript ограничивает видимость переменных и функций в разных частях программы, иначе бы разработчикам пришлось придумывать уникальные имена для всех переменных и функций в программе.

Области видимости могут вкладываться друг в друга и создавать иерархию. Переменные и функции, что объявлены в области видимости выше по иерархии, доступны в большем количестве мест программы, но только «вниз»:

```
const name = 'Кекс'; // Переменная объявлена вверху иерархии - просто в файле

function sayMyName () {
    const otherName = 'Борис'; // Переменная объявлена ниже по иерархии - внутри функции
}

sayMyName();
```

«Вниз» значит, что переменные из родительских областей видимости доступны в дочерних областях, а наоборот — нет:

```
const name = 'Кекс';

console.log(otherName); // Получим ReferenceError. Мы не можем обратиться к переменной,
                       // объявленной ниже по иерархии

function sayMyName () {
    const otherName = 'Борис';

    console.log(name); // Если бы не ошибка выше, вывелось бы "Кекс". Внутри функции можно
                      // обратиться к переменной, объявленной выше по иерархии
}

sayMyName();
```

Запомните как правило: переменные доступны только вниз

И дело не в том, в каком порядке выводятся переменные. Можем поменять их местами, результат будет прежний:

```
const name = 'Кекс';

function sayMyName () {
    console.log(name); // Выведет "Кекс".

    const otherName = 'Борис';
}

sayMyName();

console.log(otherName); // Получим ReferenceError.
```

Потому что дело именно во вложенности: переменная `name` объявлена просто в файле, а переменная `otherName` — внутри функции в этом же файле.

Дальше мы разберём, как синтаксически области видимости отделяются друг от друга, как их называют и о каких нюансах нужно помнить.

## Глава 6.1.1. Глобальная и блочная области видимости

Теперь, когда с механизмом областей видимости разобрались, давайте рассмотрим, какие области видимости бывают и какие у них особенности. Начнём с двух самых важных областей, с которыми чаще всего придётся иметь дело в работе фронтенда.

### Глобальная область видимости

Самая верхняя в иерархии областей видимости. Родительская для всех. Переменные, объявленные в глобальной области видимости, доступны везде на странице. Буквально!

Переменную `mascot` мы объявим в теге `<script>` HTML-файла страницы:

```
<!DOCTYPE html>
<html lang="ru">
<body>
  <script type="text/javascript">
    let mascot;
  </script>
  <script type="text/javascript" src="main.js"></script>
  <script type="text/javascript" src="log.js"></script>
</body>
</html>
```

Значение присвоим в подключённом к странице файле `main.js`:

```
// Файл main.js
mascot = 'Кекс';
```

А используем (выведем в консоль) — в подключённом файле `log.js`:

```
// Файл log.js
console.log(mascot); // "Кекс"
```

Чтобы переменная или функция попали в глобальную область видимости, нужно объявить их «прямо в JS-файле» (или в теге `<script>`) вне других конструкций.

#### ▶ Что такое `window`

Благодаря глобальной области видимости можно объявить переменную в одном файле, задать ей значение в другом файле, а воспользоваться — вообще в третьем месте. До появления модулей в JavaScript (о них поговорим отдельно) у глобальной области видимости было одно преимущество: с её помощью можно обмениваться данными между подключёнными к странице JS-файлами.

Сейчас, когда в JavaScript есть модули, остались одни недостатки. И основной недостаток — это возможность случайно изменить значение глобальной переменной в одном из файлов, так сказать затереть исходное значение. И при использовании этой переменной в другом файле мы получим не то значение, которое ожидали. На первый взгляд может показаться, что такая ситуация — затереть исходную переменную — невозможна в нашем коде. Увы, на самом деле это весьма распространённая ошибка. Поэтому мы рекомендуем изучить модули и никогда не использовать глобальную область видимости.

### Блочная область видимости

Самая универсальная из областей, прямо как тег `<div>` вёрстке. А «блочная», потому что ограничителем выступает блок кода `[]`. Переменные, объявленные в блочной области видимости, доступны в ней самой и во вложенных в неё других блочных областях:

```
{
  const mascot = 'Кекс';
  if (2 > 1) {
    console.log(mascot); // "Кекс"
  }
}

{
  const mascot = 'Борис';
  console.log(mascot); // "Борис"
}

console.log(mascot); // ReferenceError: mascot is not defined
```

В двух блоках кода есть свои переменные `mascot`, в первом блоке переменная выводится в консоль во вложенной области видимости `if`, а вне блоков кода переменная `mascot` отсутствует, поэтому при попытке обратиться к ней мы получим ошибку `ReferenceError`. Правило «Переменные доступны только вниз» сохраняется.

#### ▶ Что же тогда «Локальная область видимости»

### Вместо резюме

Давайте закрепим на условном файле `scope.js` знания о глобальной и блочном областях видимости:

```
scope.js - Visual Studio Code
C: > Users > work > JS scopejs > ...
1  function createRandomMascot () { /*...*/ }
2
3  √ function getMascot (isHtmlAcademy) {
4    const otherMascot = createRandomMascot();
5
6  √   if (isHtmlAcademy) {
7    const htmlAcademyMascot = 'Кекс';
8
9      return htmlAcademyMascot;
10
11
12   return otherMascot;
13 }
```

Функции `createRandomMascot()` и `getMascot()` объявлены просто в JS-файле, а значит попадают в глобальную область видимости, но сами создают блочные области видимости. Внутри блочной области видимости функции `getMascot()` у нас есть дочерняя блочная область видимости, созданная условием `if`.

Теперь к переменным:

```
scope.js - Visual Studio Code
C: > Users > work > JS scopejs > ...
1  function createRandomMascot () { /*...*/ }
2
3  √ function getMascot (isHtmlAcademy) {
4    const otherMascot = createRandomMascot();
5
6  √   if (isHtmlAcademy) {
7    const htmlAcademyMascot = 'Кекс';
8
9      return htmlAcademyMascot;
10
11
12   return otherMascot;
13 }
```

Правило «Переменные доступны только вниз» сохраняется, поэтому внутри `getMascot()` доступна функция `createRandomMascot()` из глобальной области видимости, а вот переменные `otherMascot` и `htmlAcademyMascot`, объявленные внутри функции `getMascot()`, в глобальную область видимости не попадают. Переменная `htmlAcademyMascot` вообще доступна только в условии `if` и не попадает даже в область видимости функции `getMascot()`.

## Глава 6.1.2. Модульная область видимости, лексическое окружение и прочие нюансы

Теперь, когда с [двумя основными областями видимости разобрались](#), давайте рассмотрим нюансы.

### Модульная область видимости

Это область видимости, создаваемая модулем. Модуль – это JS-файл (или тег `<script>`), написанный и подключаемый по общим правилам. О модулях мы ещё поговорим отдельно. Пока достаточно знать, что обычный JS-код становится модульным, если при подключении указать атрибут `type="module"`.

«**Ещё раз**, если скрипт подключен как модуль, то все переменные и функции, объявленные в этом модуле, в глобальную область видимости не попадают. При этом переменные из глобальной области видимости модулю доступны.

В область видимости модуля попадают переменные и функции, объявленные «прямо в JS-модуле» (или в теге `<script>`-модуле). Возьмём наш пример из [прошлой главы](#) и немного его изменим:

```
<!DOCTYPE html>
<html lang="ru">
  <body>
    <script type="text/javascript">
+   <script type="module">
      let mascot;
      </script>
      <script type="text/javascript" src="main.js"></script>
    </body>
  </html>
```

```
// Файл main.js
mascot = 'Кекс'; // ReferenceError: assignment to undeclared variable mascot
```

Как только мы превратили наш скрипт в модуль, указав `type="module"`, всё сломалось. Мы больше не можем в файле `main.js` обратиться к переменной `mascot`, потому что она существует только в модульной области видимости.

Но во всех вложенных областях переменная доступна, например она доступна во вложенной блочной области видимости функции `log()`:

```
<!DOCTYPE html>
<html lang="ru">
  <body>
    <script type="module">
      let mascot;

      function log () {
        mascot = 'Кекс';
        console.log(mascot);
      }

      log(); // "Кекс"
    </script>
    <script type="text/javascript" src="main.js"></script>
  </body>
</html>
```

Внутри модуля тоже работает правило «Переменные доступны только вниз».

### Затенение переменных

Когда браузер читает JavaScript-код и встречает переменную, он ищет её объявление снизу вверх. То есть сперва в той области, где встретил переменную, а потом вверх по иерархии. Если вплоть до глобальной области видимости переменная будет не найдена, то выполнение прервётся ошибкой `ReferenceError`:

```
function log () {
  if (2 > 1) {
    console.log(mascot);
  }
}

log(); // ReferenceError: mascot is not defined
```

Чтобы вывести в консоль переменную `mascot`, JavaScript сперва поискает её в блоке кода `if`, затем в блоке функции `log()`, после в модульной области видимости (если файл – модуль) и в конце концов в глобальной области видимости, где тоже переменную не найдёт. Тут и случится ошибка.

Благодаря такому механизму поиска в областях разного уровня вложенности можно использовать переменные с одним и тем же именем:

```
function log () {
  const mascot = 'Кекс';
  console.log(mascot); // "Кекс"

  if (2 > 1) {
    const mascot = 'Борис';
    console.log(mascot); // "Борис"
  }
}

log();
```

Это называется затенение переменных родительских областей видимости, когда мы во вложенной области `if` объявляем одноимённую переменную с родительской областью функции `log()`.

Может показаться, что это удобно, что не придётся придумывать уникальные имена, однако именно затенение переменных часто приводит к долгим часам отладки кода, потому что стоит забыть затенить переменную, JavaScript ошибки не выдаст, он просто возьмёт значение переменной из родительской области видимости:

```
function log () {
  const mascot = 'Кекс';
  console.log(mascot); // "Кекс"

  if (2 > 1) {
    console.log(mascot); // "Кекс"
  }
}

log();
```

Мы получим оба раза `"Кекс"`, хотя ожидали одного Кекса и одного Бориса. Если вам кажется, что забыть объявить переменную трудно, то хочу напомнить, что параметры функции, те же переменные:

```
const members = [['Саша', 'Игорь'], ['Лидия', 'Сергей']];

members.forEach(() => {
  members.forEach((member) => {
    console.log(member);
  });
});
```

Данный код выведет в консоль:

```
// ["Саша", "Игорь"]
// ["Лидия", "Сергей"]
// ["Саша", "Игорь"]
// ["Лидия", "Сергей"]
```

Хотя мы ожидали:

```
'Саша'
'Игорь'
'Лидия'
'Сергей'
```

А дело в том, что мы забыли параметр:

```
const members = [['Саша', 'Игорь'], ['Лидия', 'Сергей']];

members.forEach(() => {
+  members.forEach((members) => {
    members.forEach((member) => {
      console.log(member);
    });
  });
});
```

Если писать код правильно и придумать уникальные имена параметрам и переменным, такой ошибки не возникнет:

```
const groups = [['Саша', 'Игорь'], ['Лидия', 'Сергей']];

groups.forEach((members) => {
  members.forEach((member) => {
    console.log(member);
  });
});

// Выведет в консоли:
// 'Саша'
// 'Игорь'
// 'Лидия'
// 'Сергей'
```

### Лексическая область видимости

Нет, это не ёщё один вид области видимости. Лексической областью видимости называют набор всех переменных и параметров, доступных функции при объявлении. Из всех областей видимости, например:

```
const MAGIC = 42;
```

```
function print (intro) {
  function getMascot () {
    const good = 'Кекс';
    const bad = 'Мистер Бигглсуорт';

    if (MAGIC === 42) {
      return good;
    }

    return bad;
  }

  console.log(
    intro + ': ' + getMascot()
  );
}
```

```
print('Наш талисман'); // "Наш талисман: Кекс"
```

Функции `print()` доступна переменная `MAGIC` из глобальной области видимости, а в самой функции объявлен параметр `intro` и другая функция `getMascot()`. Значит лексическую область видимости функции `print()` составят значения `MAGIC`, `intro` и `getMascot()`.

Для функции `getMascot()` лексическая область будет похожей, плюс параметры и переменные внутри самой функции, итого: `MAGIC`, `intro`, `good` и `bad`. Мы не обязаны использовать все доступные значения, например `intro`, но они будут доступны просто потому что.

Важно понимать, что составляет лексическую область видимости, потому что её значения как бы прикрепляются к функции в момент объявления и доступны всегда и везде, где бы мы функции ни использовали (вызывали). А на этом строится другой механизм – замыкания.

## Глава 6.2. Введение в ECMAScript-модули

**Модуль** – это функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом.

Отдельный файл – это отдельный файл. Нечего добавить. Поговорим подробнее про «функционально законченный фрагмент программы».

Законченный фрагмент кода – это фрагмент, который выполняет конкретно поставленную задачу и ничего лишнего. Мы будем писать модули в разных файлах, где 1 файл = 1 модуль, который выполняет одну конкретную задачу. Но сначала давайте на примере разберёмся, зачем вообще нужны модули.

Представим, что нам поставили задачу построить самолёт. Нельзя просто так взять и построить самолёт. Сперва одна команда инженеров должна разработать чертежи, после другая команда техников должна собрать по этим чертежам фюзеляж, параллельно третья команда должна разработать двигатели для самолёта, четвёртая – авионику, пятая – салон и так далее. Если в будущем потребуется модифицировать этот самолёт, то при таком подходе не нужно будет разрабатывать самолёт заново, достаточно будет изменить что-то одно. Например, салон. Возьмём для примера самолёт Boeing 737. Кто часто летает, обращат внимание: у национального перевозчика несколько рядов бизнес-класса и пару десятков эконом-класса. А вот у лоукостера 32 ряда и все эконом-класса. В обоих случаях самолёт один и тот же – Boeing 737. Так и с приложением.

### Преимущества модульного подхода

- Каждая команда фокусируется на своей задаче.
- Некоторые команды могут делать свою работу параллельно и независимо друг от друга.
- Команду или результат её работы можно легко заменить (в сравнении с немодульным подходом). Пример выше – в самолёте без проблем можно установить салон другой комплектации или от другого производителя, не изменяя конструкцию самолёта.

Из преимуществ вытекает другое правило – модули должны быть максимально универсальными.

### Задачи модуля

#### Пространство имён

Модуль изолирует пространство имён, чтобы переменные из одного модуля не попадали в другой модуль, как это бывает с глобальными переменными. Всё, что создаётся в модуле, остаётся в модуле.

```
// Файл mother.js
const name = 'Eve';
```

```
// Файл father.js
const name = 'Adam';
```

Таким образом мы создадим два модуля `mother.js` и `father.js`, где в каждом будет своя переменная `name`. Ни одна из них не попадёт в глобальную область видимости `window`, а значит, если мы подключим оба файла к странице, не будет никаких конфликтов.

#### Зависимости

Модуль должен описывать и давать понимание, какие у него есть зависимости. Например, модуль может зависеть от других модулей.

```
// Файл cain.js
import {name} from './mother.js';

'My mother is ${name}'; // My mother is Eve
```

Модуль `cain.js` зависит от модуля `mother.js`, а конкретно от переменной `name`, полученной из этого модуля.

По синтаксису `import` и `export` сейчас нужно знать лишь то, что они есть, что `import` – это импорт, а `export` – экспорт в прямых своих значениях. О всех остальных нюансах мы поговорим далее.

#### Интерфейс

Модуль должен описывать интерфейс – методы и свойства – которые он может предоставить другим модулям. Чтобы предыдущий пример работал, нам нужно показать, что у модуля `mother.js` в интерфейсе есть свойство `name`, которое могут использовать другие модули.

```
// Файл mother.js
const name = 'Eve';

export {name};
```

#### Модули до ECMAScript 2015

Раньше в JavaScript не было штатной возможности писать модульный код, поэтому разработчики выдумывали обходные пути. Например, использовали [IIFE \(Immediately Invoked Function Expression\)](#):

```
// Файл mother.js
'use strict';
(function() {
  window.mother = {
    name: 'Eve',
  };
})();
```

Суть этого подхода была в том, что задача пространства имён решалась с помощью области видимости IIFE, интерфейс реализовывался с помощью явной записи методов модуля в глобальную область видимости `window`, таким же образом использовались зависимости.

В качестве примера давайте рассмотрим организацию семьи в модульном подходе с помощью IIFE:

```
// Файл mother.js
'use strict';
(function() {
  window.mother = {
    name: 'Eve',
  };
})();
```

```
// Файл father.js
'use strict';
(function() {
  window.father = {
    name: 'Adam',
  };
})();
```

```
// Файл cain.js
'use strict';
(function() {
  'My mother is ${window.mother.name}'; // My mother is Eve
  'My father is ${window.father.name}'; // My father is Adam
})();
```

У такого подхода был ряд проблем: ручное подключение, необходимость помнить порядок использования зависимостей и так далее.

На замену этому подходу начали появляться другие: [AMD](#), [CommonJS](#), [UMD](#). Не будем останавливаться на них подробно, при желании описание и принципы их работы вы найдёте по ссылкам.

Всё это переизобретение велосипеда продолжалось достаточно долго времени. И вот в 2015 году случилось то, чего многие так ждали. В стандарте ECMAScript 2015 появилась возможность описывать модули штатными средствами языка:

```
// Файл mother.js
const motherName = 'Eve';

export {motherName};
```

```
// Файл father.js
const fatherName = 'Adam';

export {fatherName};
```

```
// Файл cain.js
import {motherName} from './mother.js';
import {fatherName} from './father.js';

'My mother is ${motherName}'; // My mother is Eve
'My father is ${fatherName}'; // My father is Adam
```

Чтобы браузер считал JS-файл модулем, его нужно подключить как модуль:

```
<script type="module" src="cain.js"></script>
```

В противном случае модульные конструкции вроде `import` и `export` ничего кроме ошибки не вызовут.

«**Кстати**, модули по умолчанию работают в строгом режиме, поэтому нет больше необходимости писать `'use strict'`.

Модули ECMAScript 2015 выполняют все задачи, которые мы ставили в начале статьи:

- они ограничивают область видимости переменных, и нам больше не нужно городить IIFE и подобное;
- у модулей есть синтаксис для описания зависимостей;
- у модулей есть синтаксис для описания интерфейса.

Кроме того модули, за счёт `import` и `export` непосредственно внутри JavaScript кода, частично решают проблему порядка подключения JS-файлов. С модулями об этом можно больше не думать.

## Глава 6.3. Экспорт и импорт

Ранее мы затронули понятия экспорта `export` и импорта `import`, как способ взаимодействия между модулями. Существует два вида такого взаимодействия и несколько их комбинаций. В этой статье мы рассмотрим виды, в следующих – их комбинации. Начнём с самого простого вида, и в то же время самого надёжного.

### Просто импорт

Когда нам нужно только выполнить код модуля, достаточно его просто импортировать:

```
// Файл alert.js
alert('Hello, world!');
```

```
// Файл index.js
import './alert.js';
```

### Именованные экспорт и импорт

**« Обратите внимание**, экспорт и импорт похожи на деструктуризацию, но только похожи.

Всё просто, в одном модуле мы явно говорим, что хотим экспортить:

```
// Файл mother.js
const name = 'Eve';
const age = 18;

export const sex = 'female'; // Экспорт сразу при объявлении

export {name, age}; // Экспорт уже объявленных переменных
```

А в другом явно это импортируем:

```
// Файл cain.js
import {name} from './mother.js';

'My mother is ${name}'; // My mother is Eve
```

Имена переменных должны совпадать полностью. Если запрашиваемой переменной нет среди экспортруемых, то будет ошибка, и модуль не загрузится. Импортировать всё, что экспортит модуль, необязательно, поэтому мы ограничились только переменной `name`, а могли бы импортировать все:

```
import {sex, name, age} from './mother.js';
```

Экспортить одну и ту же переменную дважды нельзя:

```
// Файл mother.js
const name = 'Eve';
const age = 18;

export const sex = 'female';

export {sex};
```

Но если очень надо, есть способ – переименование при экспорте и импорте. О нём поговорим чуть позже.

#### Импорт – не объявление

Важно помнить, что импорт переменной не то же самое, что её объявление. При импорте переменная не создаётся, а честно импортируется, самая настоящая переменная из другого модуля. Такое поведение справедливо для всех видов импорта, которые мы разберём дальше (экспортить как-то иначе тоже нельзя), поэтому будьте внимательны при работе с импортированными переменными. Ведь сложные типы данных, вроде объектов или массивов, при импорте передаются по ссылке, и их можно нечаянно испортить:

```
// Файл items.js
const items = ['one', 'two', 'three'];

export {items};
```

```
// Файл reverse.js
import {items} from './items.js';

items.reverse(); // Испортили массив в модуле items.js, хотя переворачиваем его в reverse.js
```

Поэтому не забывайте про `Object.assign()` и `Array.prototype.slice()` и им подобные методы, когда работаете с импортированной структурой:

```
// Файл items.js
const items = ['one', 'two', 'three'];

export {items};
```

```
// Файл reverse.js
import {items} from './items.js';

items.slice().reverse(); // Перевернули копию, исходный массив в items.js остался цел
```

С примитивами тоже не всё так просто. Даже если импортируете `let`-переменную, по спецификации движок JavaScript не позволит вам изменить её. Она считается `read-only` переменной (доступной только для чтения):

```
// Файл mother.js
let name = 'Eve';

export {name};
```

```
// Файл cain.js
import {name} from './mother.js';

name = 'Adam'; // Ничего не выйдет, кроме ошибки
```

Поэтому распространённой практикой является экспорт только `const`-значений и им подобных (классов, например). На курсе мы тоже будем экспортить только `const`-переменные или `class`.

### Экспорт по умолчанию

Синтаксический экспорт по умолчанию отличается вводом дополнительного ключевого слова `default`:

```
// Файл mother.js
const mother = {
  name: 'Eve',
  age: 18,
  sex: 'female',
};

export default mother;
```

Фигурные скобки не нужны, поскольку экспортовать по умолчанию можно лишь одну переменную.

Главное отличие такого экспорта от именованного – возможность экспортить значение напрямую, без переменной:

```
// Файл mother.js
export default {
  name: 'Eve',
  age: 18,
  sex: 'female',
};
```

Из-за этого отличается и импорт такого значения «по умолчанию». Раз у нас нет переменной, то под каким именем импортировать? Под любым!

```
import father from './mother.js';
```

В этом заключается и главная особенность, и главная проблема экспорта по умолчанию – мы можем задать любое имя при импорте. Такое поведение усложняет отладку и навигацию по коду, поэтому мы не рекомендуем использовать экспорт по умолчанию.

### Нюансы

#### Код модуля выполняется целиком

Независимо от того, импортирован ли просто модуль `import './alert.js'` или какие-то конкретные переменные из модуля `import {name} from './mother.js'`, браузер выполнит код всего модуля. Другими словами, если модуль импортирован, его код будет выполнен целиком независимо от способа импорта.

#### Только первый уровень

`import` и `export` не могут быть вложены в функции или другие блоки кода:

```
// Файл cain.js
if (true) {
  // Так нельзя
  import {motherName} from './mother.js';
}

const name = 'cain';

(() => {
  // Так тоже нельзя
  export {name};
})()
```

#### Никакого поднятия (hoisting)

Импортированные переменные не поднимаются, поэтому `import` всегда должен быть в начале файла:

```
const message = `My mother is ${motherName}`;

// Так нельзя, браузер выдаст ошибку
import {motherName} from './mother.js';
```

#### Импорт того, чего нет

Если импортировать переменную, которая в модуле не экспортится, получим ошибку.

#### Динамический импорт

Он существует. Разбирать мы его пока не будем, потому что у него нет полной поддержки браузерами.

### Когда что использовать

Мы рекомендуем всегда использовать только именованный экспорт. Исключение – экспорт `class`, но об этом потом.