Теория

~ 3 минуты

Глава 1.1. Что такое JavaScript

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

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

~ 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

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 () {
```

```
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.

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.

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.

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.

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.random();

// Хорошо
const EARTH_RADIUS = 6371;
```

PascalCase

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

```
// Неправильно
const RAINBOW = {
  red: '#ff0000',
  orange: '#ffa500',
  yellow: '#ffff00',
  green: '#008000',
  lightBlue: '#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 `smmrz` 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.

Введение

~ 2 минуты

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

~ 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.

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.

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 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 = `Длинныййй`;
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
~ 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.

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

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.

Теория

~ 4 минуты

Глава 2.8. Стрелочные функции

Глава 2.6. 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. arrowfunction)*.

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.

Теория
~ 6 минут

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 поидет дальше, и функция вернет неожиданный результат. В примере мы «проваливаемся» сквозь все кейсы, начиная с `case 1` и до `case 4`, и получаем результат `'Чётное'`, потому что `break` только в `case 4`.

Это не ошибка

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

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

isEven(1); // 'Нечётное'
```

Запомните

— Если `case` не объединяются, то должны содержать `break` или `return`.

— Выражение, передаваемое в `switch`, представляется в любом виде (примитив, переменная, вычисляемое значение или вызов функции).

— Порядок случаев не важен.

— Блок `default` не является обязательным, но его удобно использовать на случай ошибок или значений по умолчанию.

— При наличии одинаковых случаев будет выбран первый попавшийся:

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

Резюме

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

~ 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.

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 намерено для сохранения совместимости с устаревшим кодом. Как включить строгий режим узнаем ниже.

Строгий режим

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

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;
somVar = 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 (`somVar`). В этой строке мы хотели переопределить переменную `someVar`, но допустили ошибку — пропустили букву в названии переменной. В нестрогом режиме интерпретатор это пропустит и вместо ошибки (переменная не существует) создаст ещё одну глобальную переменную.

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

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

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

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

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

Некорректные операции

Строгий режим предотвращает некоторые потенциально некорректные операции. Например, в нестрогом режиме ничего не мешает 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

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.

Теория

~ 9 минут

Глава 2.12. Контекст функций и проблема потери окружения

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

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

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

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

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

```
const cat = {
  nickname: 'Кекс',
```

```
  greeting: 'Мяу',
};

const dog = {
  nickname: 'Полиграф Шариков',
  greeting: '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` приклеится к функции `say`, и тогда дать голос другим персонажам, собаке или лисе, не получится.

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

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

const dog = {
  nickname: 'Полиграф Шариков',
  greeting: '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 намерено, 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 закрепляется в момент *объявления*, а не вызова:

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

const cat = {
  nickname: 'Кекс',
  greeting: 'Мяу',
  goodies: [
    'Свежую рыбку',
    '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.

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:

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».