

Руководство по JavaScript



Перевод опубликован в [блоге](#) компании RUVDS. [Оригинал](#) статей.

Читать онлайн (много полезных комментариев):

- [Часть 1: первая программа, особенности языка, стандарты](#)
- [Часть 2: стиль кода и структура программ](#)
- [Часть 3: переменные, типы данных, выражения, объекты](#)
- [Часть 4: функции](#)
- [Часть 5: массивы и циклы](#)
- [Часть 6: исключения, точка с запятой, шаблонные литералы](#)
- [Часть 7: строгий режим, ключевое слово this, события, модули, математические вычисления](#)
- [Часть 8: обзор возможностей стандарта ES6](#)
- [Часть 9: обзор возможностей стандартов ES7, ES8 и ES9](#)

Оглавление

Часть 1: первая программа, особенности языка, стандарты

[Hello, world!](#)

[Общие сведения о JavaScript](#)

[Основные характеристики JavaScript](#)

[JavaScript и стандарты](#)

Часть 2: стиль кода и структура программ

[Стиль программирования](#)

[Стиль, используемый в этом руководстве](#)

[Лексическая структура JavaScript-кода](#)

[Unicode](#)

[Точка с запятой](#)

[Пробелы](#)

[Чувствительность к регистру](#)

[Комментарии](#)

[Литералы и идентификаторы](#)

[Зарезервированные слова](#)

Часть 3: переменные, типы данных, выражения, объекты

[Переменные](#)

[Ключевое слово var](#)

[Ключевое слово let](#)

[Ключевое слово const](#)

[Типы данных](#)

[Примитивные типы данных](#)

[Тип number](#)

[Тип string](#)

[Шаблонные литералы](#)

[Тип boolean](#)

[Тип null](#)

[Тип undefined](#)

[Объекты](#)

[Выражения](#)

[Арифметические выражения](#)

[Строковые выражения](#)

[Первичные выражения](#)

[Выражения инициализации массивов и объектов](#)

[Логические выражения](#)

[Выражения доступа к свойствам](#)

[Выражения создания объектов](#)

[Выражения объявления функций](#)

[Выражения вызова](#)

[Работа с объектами](#)

[Прототипное наследование](#)

[Функции-конструкторы](#)

[Классы](#)

[Объявление класса](#)

[Наследование, основанное на классах](#)

[Статические методы](#)

[Приватные методы](#)

[Геттеры и сеттеры](#)

[Часть 4: функции](#)

[Функции в JavaScript](#)

[Параметры функций](#)

[Значения, возвращаемые из функций](#)

[Вложенные функции](#)

[Методы объектов](#)

[Ключевое слово this](#)

[Немедленно вызываемые функциональные выражения](#)

[Поднятие функций](#)

[Стрелочные функции](#)

[Неявный возврат результатов работы функции](#)

[Ключевое слово this и стрелочные функции](#)

[Замыкания](#)

[Часть 5: массивы и циклы](#)

[Массивы](#)

[Инициализация массивов](#)

[Получение длины массива](#)

[Проверка массива с использованием метода every\(\)](#)

[Проверка массива с использованием метода some\(\)](#)

[Создание массива на основе существующего массива с использованием метода map\(\)](#)

[Фильтрация массива с помощью метода filter\(\)](#)

[Метод reduce\(\)](#)

[Перебор массива с помощью метода forEach\(\)](#)

[Перебор массива с использованием оператора for...of](#)

[Перебор массива с использованием оператора for](#)

[Метод @@iterator](#)

[Добавление элементов в конец массива](#)

[Добавление элементов в начало массива](#)

[Удаление элементов массива](#)

[Удаление элементов массива и вставка вместо них других элементов](#)

[Объединение нескольких массивов](#)

[Поиск элементов в массиве](#)

[Получение фрагмента массива](#)

[Сортировка массива](#)

[Получение строкового представления массива](#)

[Создание копий массивов](#)

[Циклы](#)

[Цикл for](#)

[Цикл forEach](#)

[Цикл do...while](#)

[Цикл while](#)

[Цикл for...in](#)

[Цикл for...of](#)

[Циклы и области видимости](#)

[Часть 6: исключения, точка с запятой, шаблонные литералы](#)

[Обработка исключений](#)

[Конструкция try...catch](#)

[Блок finally](#)

[Вложенные блоки try](#)

[Самостоятельное генерирование исключений](#)

[О точках с запятой](#)

[Правила автоподстановки точек с запятой](#)

[Примеры кода, который работает не так, как ожидается](#)

[Кавычки и шаблонные литералы](#)

[Многострочный текст](#)

[Интерполяция](#)

[Тегированные шаблоны](#)

[Часть 7: строгий режим, ключевое слово this, события, модули, математические вычисления](#)

[Строгий режим](#)

[Включение строгого режима](#)

[Борьба со случайной инициализацией глобальных переменных](#)

[Ошибки, возникающие при выполнении операций присваивания значений](#)

[Ошибки, связанные с удалением сущностей](#)

[Аргументы функций с одинаковыми именами](#)

[Восьмеричные значения](#)

[Оператор with](#)

[Особенности ключевого слова this](#)

[Ключевое слово this в строгом режиме](#)

[Ключевое слово this в методах объектов](#)

[Ключевое слово this и стрелочные функции](#)

[Привязка this](#)

[О привязке this в обработчиках событий браузера](#)

[События](#)

[Обработчики событий](#)

[Встроенные обработчики событий](#)

[Назначение обработчика свойству HTML-элемента](#)

[Использование метода addEventListener\(\)](#)

[О назначении обработчиков событий различным элементам](#)

[Объект Event](#)

[Всплытие событий](#)

[Часто используемые события](#)

[Событие load](#)

[События мыши](#)

[События клавиатуры](#)

[Событие scroll](#)

[Ограничение частоты выполнения вычислений в обработчиках событий](#)

[ES-модули](#)

[Синтаксис ES-модулей](#)

[Другие возможности импорта и экспорта](#)

[CORS](#)

[Атрибут nomodule](#)

[О модулях ES6 и WebPack](#)

[Модули CommonJS](#)

[Математические вычисления](#)

[Арифметические операторы](#)

[Сложение \(+\)](#)

[Вычитание \(-\)](#)

[Деление \(/\)](#)

[Остаток от деления \(%\)](#)

[Умножение \(*\)](#)

[Возведение в степень \(**\)](#)

[Унарные операторы](#)

[Инкремент \(++\)](#)

[Декремент \(--\)](#)

[Унарный оператор \(-\)](#)

[Унарный оператор \(+\)](#)

[Оператор присваивания и его разновидности](#)

[Приоритет операторов](#)

[Объект Math](#)

[Сравнение значений](#)

[Таймеры и асинхронное программирование](#)

[Часть 8: обзор возможностей стандарта ES6](#)

[О стандарте ES6](#)

[Стрелочные функции](#)

[Особенности ключевого слова this в стрелочных функциях](#)

[Промисы](#)

[Генераторы](#)

[Ключевые слова let и const](#)

[Классы](#)

[Конструктор класса](#)

[Ключевое слово super](#)

[Геттеры и сеттеры](#)

[Модули](#)

[Импорт модулей](#)

[Экспорт модулей](#)

[Шаблонные литералы](#)

[Параметры функций, задаваемые по умолчанию](#)

[Оператор spread](#)

[Деструктурирующее присваивание](#)

[Расширение возможностей объектных литералов](#)

[Упрощение включения в объекты переменных](#)

[Прототипы](#)

[Ключевое слово super](#)

[Вычисляемые имена свойств](#)

[Цикл for...of](#)

[Структуры данных Map и Set](#)

[Часть 9: обзор возможностей стандартов ES7, ES8 и ES9](#)

[Стандарт ES7](#)

[Метод Array.prototype.includes\(\)](#)

[Оператор возведения в степень](#)

[Стандарт ES8](#)

[Дополнение строк до заданной длины](#)

[Метод Object.values\(\)](#)

[Метод Object.entries\(\)](#)

[Метод getOwnPropertyDescriptors\(\)](#)

[Завершающие запятые в параметрах функций](#)

[Асинхронные функции](#)

[Последовательный вызов асинхронных функций](#)

[Разделяемая память и атомарные операции](#)

[Стандарт ES9](#)

[Применение операторов spread и rest к объектам](#)

[Асинхронные итераторы](#)

[Метод Promise.prototype.finally\(\)](#)

[Улучшения регулярных выражений](#)

[Управляющие последовательности Unicode в регулярных выражениях](#)

[Именованные группы](#)

[Флаг регулярных выражений s](#)

Часть 1: первая программа, особенности языка, стандарты

Hello, world!

Программа, которую по традиции называют «[Hello, world!](#)», очень проста. Она выводит куда-либо фразу «Hello, world!», или другую подобную, средствами некоего языка.

JavaScript — это язык, программы на котором можно выполнять в разных средах. В нашем случае речь идёт о браузерах и о серверной платформе Node.js. Если до сих пор вы не написали ни строчки кода на JS и читаете этот текст в браузере, на настольном компьютере, это значит, что вы буквально в считанных секундах от своей первой JavaScript-программы.

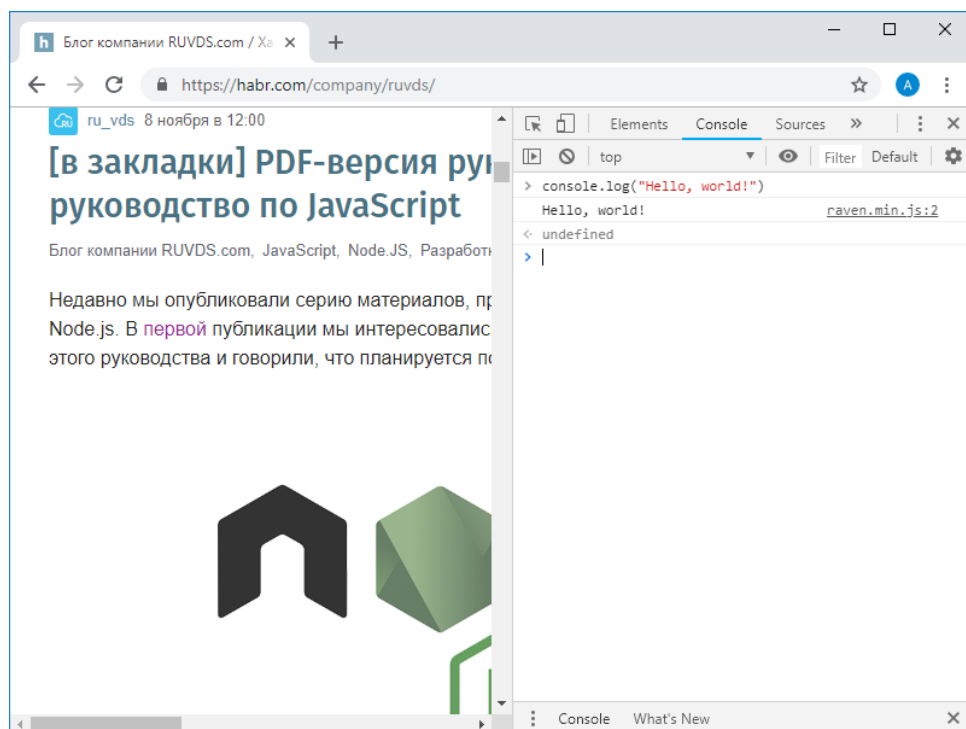
Для того чтобы её написать, если вы пользуетесь Google Chrome, откройте меню браузера и выберите в нём команду **Дополнительные инструменты > Инструменты разработчика**. Окно браузера окажется разделённым на две части. В одной из них будет видна страница, в другой откроется панель с инструментами разработчика, содержащая несколько закладок. Нас интересует закладка **Console** (Консоль). Щёлкните по ней. Не обращайте внимания на то, что уже может в консоли присутствовать (для её очистки можете воспользоваться комбинацией клавиш **Ctrl + L**). Нас сейчас интересует приглашение консоли. Именно сюда можно вводить JavaScript-код, который выполняется по нажатию клавиши **Enter**. Введём в консоль следующее:

```
console.log("Hello, world!")
```

Этот текст можно ввести с клавиатуры, можно скопировать и вставить его в консоль. Результат будет одним и тем же, но, если вы учитесь программировать, рекомендуется вводить тексты программ самостоятельно, а не копировать их.

После того, как текст программы оказался в консоли, нажмём клавишу **Enter**.

Если всё сделано правильно — под этой строчкой появится текст `Hello, world!`. На всё остальное пока не обращайте внимания.

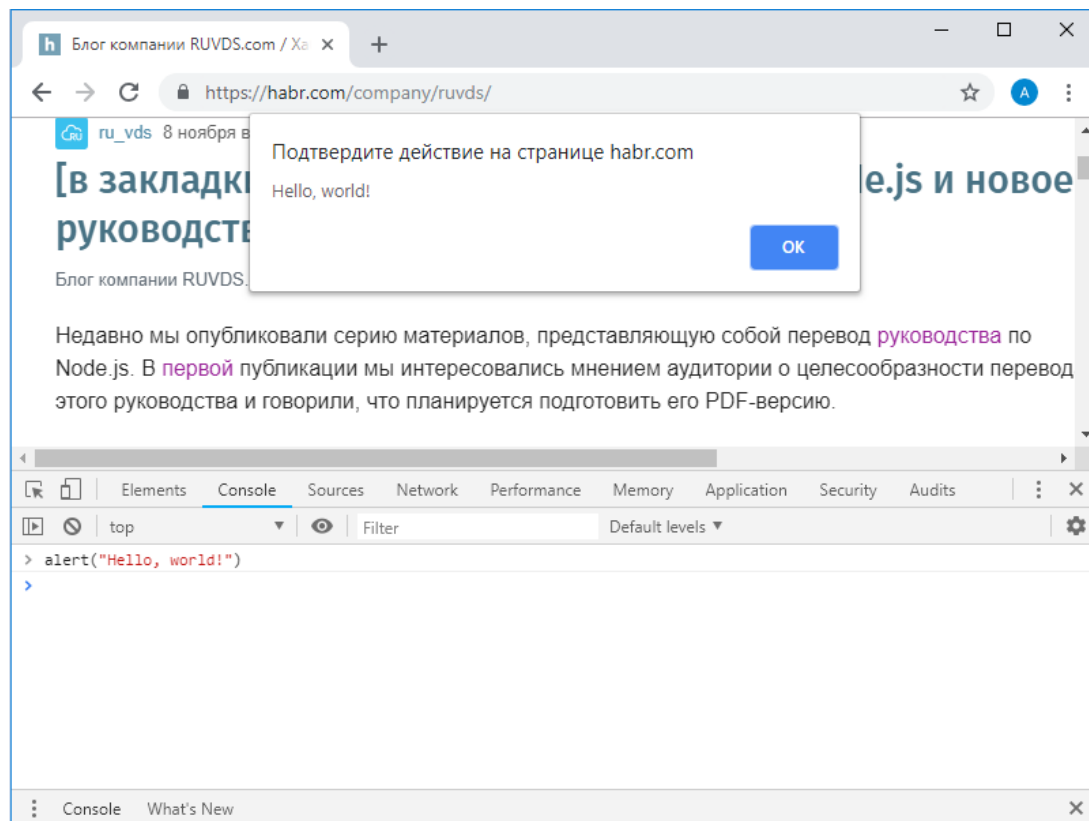


Первая программа в консоли браузера — вывод сообщения в консоль

Ещё один вариант браузерного «Hello, world!» заключается в выводе окна с сообщением. Делается это так:

```
alert("Hello, world!")
```

Вот результат выполнения этой программы.



Вывод сообщения в окне

Обратите внимание на то, что панель инструментов разработчика расположена теперь в нижней части экрана. Менять её расположение можно, воспользовавшись меню с тремя точками в её заголовке и выбирая соответствующую пиктограмму. Там же можно найти и кнопку для закрытия этой панели.

Инструменты разработчика, и, в том числе, консоль, имеются и в других браузерах. Консоль хороша тем, что она, когда вы пользуетесь браузером, всегда под рукой.

Существуют и другие способы запуска JS-кода в браузере. Так, если говорить об обычном использовании программ на JavaScript, они загружаются в браузер для обеспечения работы веб-страниц. Как правило, код оформляют в виде отдельных файлов с расширением `.js`, которые подключают к веб-страницам, но программный код можно включать и непосредственно в код страницы. Всё это делается с помощью тега `<script>`. Когда браузер обнаруживает такой код, он выполняет его. Подробности о теге [script](#) можно посмотреть на сайте [w3school.com](#). В частности, рассмотрим пример, демонстрирующий работу с веб-страницей средствами JavaScript, приведённый на этом ресурсе. Этот пример можно запустить и средствами данного ресурса (ищите кнопку `Try it Yourself`), но мы поступим немного иначе. А именно, создадим в каком-нибудь текстовом редакторе (например — в [VS Code](#) или в [Notepad++](#)) новый файл, который назовём `hello.html`, и добавим в него следующий код:

```
<!DOCTYPE html>

<html>

  <body>

    <p id="hello"></p>

    <script>
```



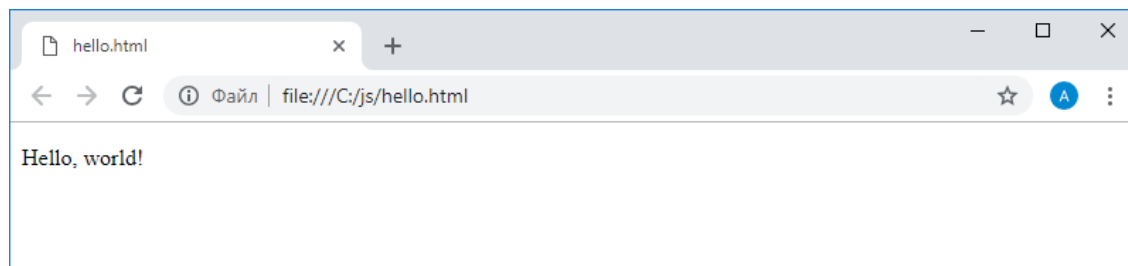
```
document.getElementById("hello").innerHTML = "Hello, world!";

</script>

</body>

</html>
```

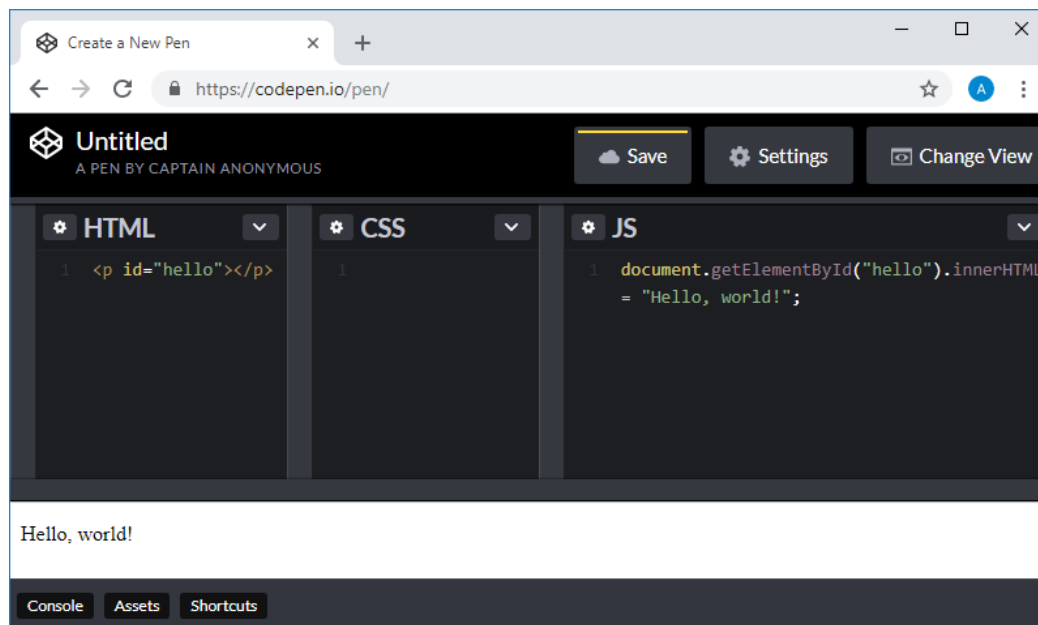
Здесь нас больше всего интересует строчка `document.getElementById("hello").innerHTML = "Hello, world!";`, представляющая собой JS-код. Этот код заключён в открывающий и закрывающий теги `<script>`. Он находит в документе HTML-элемент с идентификатором `hello` и меняет его свойство `innerHTML` (то есть — тот HTML код, который содержится внутри этого элемента) на `Hello, world!`. Если открыть файл `hello.html` в браузере, на ней будет выведен заданный текст.



Сообщение, выведенное средствами JavaScript в HTML-элемент

Как уже говорилось, примеры, приводимые на сайте [w3school.com](https://www.w3school.com), можно тут же и попробовать. Существуют и специализированные онлайн-среды для веб-разработки, и, в частности, для выполнения JS-кода. Среди них, например codepen.io, jsfiddle.net, jsbin.com.

Вот, например, как выглядит наш пример, воссозданный средствами CodePen.



В поле HTML попадает код, описывающий HTML-элемент, в поле JS — JavaScript-код, а в нижней части страницы выводится результат.

Выше мы говорили о том, что JavaScript-программы можно выполнять на различных платформах, одной из которых является серверная среда Node.js. Если вы собираетесь изучать JavaScript, ориентируясь именно на серверную разработку, вероятно, вам стоит запускать примеры именно средствами Node.js. Учтите, что, говоря упрощённо, и не учитывая особенности поддержки конкретных возможностей языка используемыми версиями Node.js и браузера, в Node.js и в браузере будет работать один и тот же код, в котором используются базовые механизмы языка. То есть, например, команда `console.log("Hello, world!")` будет работать и там и

там. Программы, использующие механизмы, специфичные для браузеров, в Node.js работать не будут (то же самое касается и попыток запуска программ, рассчитанных на Node.js, в браузере).

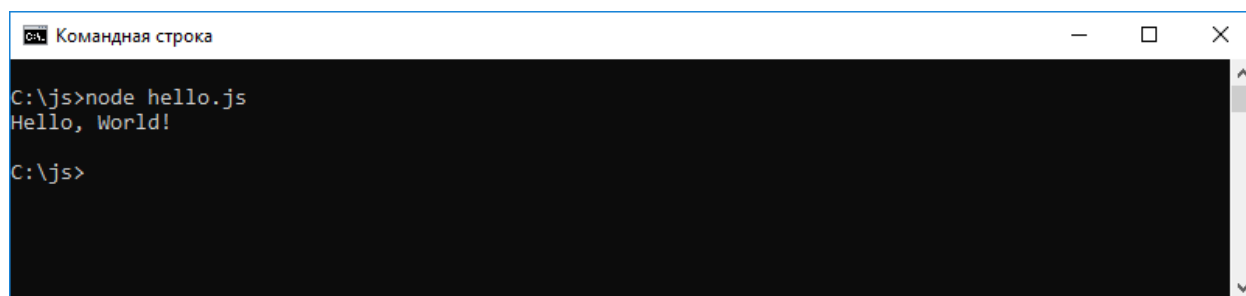
Для того чтобы запустить наш «Hello, world!» в среде Node.js, установим Node.js, скачав [отсюда](#) подходящий дистрибутив. Теперь создадим файл `hello.js` и поместим в него следующий код:

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

Средствами командной строки перейдём в папку, в которой хранится этот файл, и выполним такую команду:

```
node hello.js
```

Вот каким будет результат её выполнения:



Сообщение, выведенное средствами Node.js

Кстати, вы могли заметить (и, скорее всего, заметили, если набирали код самостоятельно), что в некоторых из вышеприведённых примеров, в конце строк, используется точка с запятой, а в некоторых — нет. В некоторых текст, который мы хотим вывести в консоль или в виде сообщения, обрамляется двойными кавычками, а в некоторых — одинарными. Возможно, сейчас вы задаётесь вопросом о том, почему это так, и о том, как, если разные варианты кода работают без ошибок, писать этот код правильно. Если ответить на этот вопрос, не вдаваясь в подробности и не учитывая некоторые мелкие детали, то можно сказать, что и тот и другой вариант использования кавычек и точки с запятой допустимы, и то, что работают они одинаково. Выбор конкретного варианта зависит от стиля написания кода, которого придерживается программист, и от потребностей некоего фрагмента программы. Кроме того, на этих простых примерах вы могли ощутить одну из характерных для JavaScript черт, которая заключается в том, что язык даёт программисту определённую свободу.

Теперь, после того, как состоялось ваше первое знакомство с JavaScript, предлагаем подробнее поговорить об этом языке.

Общие сведения о JavaScript

JavaScript — это один из самых популярных языков программирования в мире. Он, созданный более 20 лет назад, прошёл в своём развитии огромный путь. JavaScript задумывался как скриптовый язык для браузеров. В самом начале он обладал куда более скромными возможностями, чем сейчас. Его, в основном, использовали для создания несложных анимаций, вроде выпадающих меню, о нём знали как о части технологии [Dynamic HTML](#) (DHTML, динамический HTML).

Со временем потребности веб-среды росли, в частности, появлялись новые API, и JavaScript, для поддержки веб-разработки, нужно было не отставать от других технологий.

В наши дни JS используется не только в традиционных браузерах, но и за их пределами. В частности, речь идёт о серверной платформе Node.js, о возможностях по использованию JavaScript в разработке встраиваемых и мобильных приложений, о решении широкого спектра задач, для решения которых раньше JavaScript не использовался.

Основные характеристики JavaScript

JavaScript — это язык, который отличается следующими особенностями:

- **Высокоуровневый.** Он даёт программисту абстракции, которые позволяют не обращать внимание на особенности аппаратного обеспечения, на котором выполняются JavaScript-программы. Язык автоматически управляет памятью, используя сборщик мусора. Разработчик, в результате, может сосредоточиться на решении стоящих перед ним задач, не отвлекаясь на низкоуровневые механизмы (хотя, надо отметить, это не отменяет необходимости в рациональном использовании памяти). Язык предлагает мощные и удобные средства для работы с данными различных типов.
- **Динамический.** В отличие от статических языков программирования, динамические языки способны, во время выполнения программы, выполнять действия, которые статические языки выполняют во время компиляции программ. У такого подхода есть свои плюсы и минусы, но он даёт в распоряжение разработчика такие мощные возможности, как динамическая типизация, позднее связывание, рефлексия, функциональное программирование, изменение объектов во время выполнения программы, замыкания и многое другое.
- **Динамически типизированный.** Типы переменных при JS-разработке задавать необязательно. В одну и ту же переменную можно, например, сначала записать строку, а потом — целое число.
- **Слабо типизированный.** В отличие от языков с сильной типизацией, языки со слабой типизацией не принуждают программиста, например, использовать в неких ситуациях объекты определённых типов, выполняя, при необходимости, неявные преобразования типов. Это даёт больше гибкости, но JS-программы не являются типобезопасными, из-за этого усложняются задачи проверки типов (на решение этой проблемы направлены TypeScript и Flow).
- **Интерпретируемый.** Широко распространено мнение, в соответствии с которым JavaScript является интерпретируемым языком программирования, что означает, что программы, написанные на нём, не нуждаются в компиляции перед выполнением. JS в этом плане противопоставляют таким языкам, как C, Java, Go. На практике же браузеры, для повышения производительности программ, выполняют компиляцию JS-кода перед его выполнением. Этот шаг, однако, прозрачен для программиста, он не требует от него дополнительных усилий.
- **Мультипарадигменный.** JavaScript не навязывает разработчику использование какой-то конкретной парадигмы программирования, в отличие, например, от Java (объектно-ориентированное программирование) или C (императивное программирование). Писать JS-программы можно, используя объектно-ориентированную парадигму, в частности — применяя прототипы и появившиеся в стандарте ES6 классы. Программы на JS можно писать и в функциональном стиле, благодаря тому, что функции здесь являются объектами первого класса. JavaScript допускает и работу в императивном стиле, используемом в C.

Да, кстати, надо отметить, что у JavaScript и Java нет ничего общего. Это — совершенно разные языки.

JavaScript и стандарты

ECMAScript, или ES, это название стандарта, которым руководствуются разработчики JavaScript-движков, то есть — тех сред, где выполняются JS-программы. Различные стандарты вводят в язык новые возможности, говоря о которых нередко упоминают наименование стандартов в сокращённой форме, например — ES6. ES6 — это то же самое, что и ES2015, только в первом случае число означает номер версии стандарта (6), а во втором — год принятия стандарта (2015).

Сложилось так, что в мире веб-программирования очень долго был актуален стандарт ES3, принятый в 1999 году. Четвёртой версии стандарта не существует (в неё попытались добавить слишком много новых возможностей и так и не приняли). В 2009 году был принят стандарт ES5, который представлял собой прямо-таки огромное обновление языка, первое за 10 лет. После него, в 2011 году, был принят стандарт ES5.1, в нём тоже было немало нового. Весьма значительным, в плане новшеств, стал и стандарт ES6, принятый в 2015 году. Начиная с 2015 года, новые версии стандарта принимают каждый год.

Самой свежей версией стандарта на момент публикации этого материала является [ES9](#), принятая в июне 2018 года.

Часть 2: стиль кода и структура программ

Стиль программирования

«Стиль программирования», или «стандарт кодирования», или «стиль кода» — это набор соглашений, которые используются при написании программ. Они регламентируют особенности оформления кода и порядок использования конструкций, допускающих неоднозначности. В нашем случае речь идёт о программах, написанных на JavaScript. Если программист работает над неким проектом сам, то стиль кода, применяемый им, представляет его «договор» с самим собой. Если речь идёт о команде, то это — соглашения, которые используются всеми членами команды. Код, написанный с применением некоего свода правил, делает кодовую базу программного проекта единообразной, улучшает читабельность и понятность кода.

Существует немало руководств по стилю. Вот 2 из них, которыми в мире JavaScript пользуются чаще всего:

- [Руководство по стилю JavaScript от Google](#)
- [Руководство по стилю JavaScript от Airbnb](#)

Вы вполне можете выбрать любое из них или придумать собственные правила. Самое главное — последовательно использовать одни и те же правила при работе над неким проектом. При этом, если, например, вы придерживаетесь одного набора правил, а в существующем проекте, над которым вам нужно поработать, используются собственные правила, нужно придерживаться правил проекта.

Форматирование кода можно выполнять вручную, а можно воспользоваться средствами автоматизации этого процесса. На самом деле, форматирование JS-кода и его проверка до запуска — это отдельная большая тема. [Вот](#) одна из наших публикаций, посвящённая соответствующим инструментам и особенностям их использования.

Стиль, используемый в этом руководстве

Автор этого материала, в качестве примера собственного руководства по стилю, приводит свод правил, которым он старается следовать, оформляя код. Он говорит, что в коде примеров ориентируется на самую свежую версию стандарта, доступную в современных версиях браузеров. Это означает, что для выполнения такого кода в устаревших браузерах понадобится использовать транспилятор, такой как [Babel](#). JS-транспилаторы позволяют преобразовывать код, написанный с использованием новых возможностей языка, таким образом, чтобы его можно было бы выполнять в браузерах, не поддерживающих эти новые возможности. Транспилатор может обеспечивать поддержку возможностей языка, которые ещё не вошли в стандарт, то есть, не реализованы даже в самых современных браузерах.

Вот список правил, о которых идёт речь.

- Выравнивание. Для выравнивания блоков кода используются пробелы (2 пробела на 1 уровень выравнивания), знаки табуляции не используются.
- Точка с запятой. Точка с запятой не используется.
- Длина строки. 80 символов (если это возможно).
- Однострочные комментарии. Такие комментарии используются в коде.
- Многострочные комментарии. Эти комментарии используются для документирования кода.
- Неиспользуемый код. Неиспользуемый код не остаётся в программе в закомментированном виде на тот случай, если он понадобится позже. Такой код, если он всё же понадобится, можно будет найти в системе контроля версий, если она используется, или в чём-то вроде заметок программиста, предназначенных для хранения подобного кода.
- Правила комментирования. Не нужно комментировать очевидные вещи, добавлять в код комментарии, которые не помогают разобраться в его сути. Если код объясняет себя сам благодаря хорошо подобранным именам функций и переменных и JSДок-описаниям функций, дополнительные комментарии в этот код добавлять не стоит.
- Объявление переменных. Переменные всегда объявляются в явном виде для предотвращения загрязнения глобального объекта. Ключевое слово `var` не используется. Если значение переменной в

ходе выполнения программы менять не планируется, её объявляют в виде константы (подобные константы нередко тоже называют «переменными») с помощью ключевого слова `const`, используя его по умолчанию — кроме тех случаев, когда менять значение переменной планируется. В таких случаях используется ключевое слово `let`.

- Константы. Если некие значения в программе являются константами, их имена составляют из прописных букв. Например — `CAPS`. Для разделения частей имён, состоящих из нескольких слов, используется знак подчёркивания (`_`).
- Функции. Для объявления функций используется стрелочный синтаксис. Обычные объявления функций применяются только в особых случаях. В частности, в методах объектов или в конструкторах. Делается это из-за особенностей ключевого слова `this`. Функции нужно объявлять с использованием ключевого слова `const`, и, если это возможно, надо явно возвращать из них результаты их работы. Не возбраняется использование вложенных функций для того, чтобы скрыть от основного кода некие вспомогательные механизмы.

Вот пример пары простых стрелочных функций:

```
const test = (a, b) => a + b
```

```
const another = a => a + 2
```

- Именованное существование. Имена функций, переменных и методов объектов всегда начинаются со строчной буквы, имена, состоящие из нескольких слов, записываются с использованием верблюжьего стиля (выглядят такие имена как `camelCase`). С прописной буквы начинаются только имена функций-конструкторов и классов. Если вы используете некий фреймворк, предъявляющий особые требования к именованию существ — пользуйтесь предписываемыми им правилами. Имена файлов должны состоять из строчных букв, отдельные слова в именах разделяются тире (`-`).
- Правила построения и форматирования выражений.

if. Вот несколько способов записи условного оператора `if`:

```
if (condition) {  
    statements  
}
```

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

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

for. Для организации циклов используется либо стандартная конструкция `for`, пример которой приведён ниже, либо цикл `for of`. Циклов `for in` следует избегать — за исключением тех случаев, когда они используются вместе с конструкцией `.hasOwnProperty()`. Вот схема цикла `for`:

```
for (initialization; condition; update) {  
    statements  
}
```

while. Вот схематичный пример цикла `while`:

```
while (condition) {  
    statements  
}
```

do. Вот структура цикла `do`:

```
do {  
    statements  
} while (condition);
```

switch. Ниже показана схема условного оператора `switch`:

```
switch (expression) {  
    case expression:  
        statements  
    default:  
        statements  
}
```

try. Вот несколько вариантов оформления конструкции `try-catch`. Первый пример показывает эту конструкцию без блока `finally`, второй — с таким блоком.

```
try {  
    statements  
} catch (variable) {  
    statements  
}  
  
try {  
    statements  
} catch (variable) {  
    statements  
} finally {
```

statements

}

- Пробелы. Пробелы следует использовать разумно, то есть так, чтобы они способствовали улучшению читаемости кода. Так, пробелы ставят после ключевых слов, за которыми следует открывающая круглая скобка, ими обрамляют операторы, применяемые к двум операндам (+, -, /, *, && и другие). Пробелы используют внутри цикла `for`, после каждой точки с запятой, для отделения друг от друга частей заголовка цикла. Пробел ставится после запятой.
- Пустые строки. Пустыми строками выделяют блоки кода, содержащие логически связанные друг с другом операции.
- Кавычки. При работе со строками используются одинарные кавычки ('), а не двойные ("). Двойные кавычки обычно встречаются в HTML-атрибутах, поэтому использование одинарных кавычек помогает избежать проблем при работе с HTML-строками. Если со строками нужно выполнять некие операции, подразумевающие, например, их конкатенацию, следует пользоваться шаблонными литералами, которые оформляют с помощью обратных кавычек (`).

Лексическая структура JavaScript-кода

Поговорим о строительных блоках JavaScript-кода. В частности — об использовании кодировки Unicode, о точках с запятой, пробелах, о чувствительности языка к регистру символов, о комментариях, о литералах, об идентификаторах и о зарезервированных словах.

Unicode

JavaScript-код представляется с использованием кодировки Unicode. Это, в частности, означает, что в коде, в качестве имён переменных, можно использовать, скажем, символы смайликов. Делать так, конечно же, не рекомендуется. Важно здесь то, что имена идентификаторов, с учётом некоторых [правил](#), могут быть записаны на любом языке, например — на японском или на китайском.

Точка с запятой

Синтаксис JavaScript похож на синтаксис C. Вы можете встретить множество проектов, в которых, в конце каждой строки, находится точка с запятой. Однако точки с запятой в конце строк в JavaScript необязательны. В подавляющем большинстве случаев без точки с запятой можно обойтись. Разработчики, которые, до JS, пользовались языками, в которых точка с запятой не применяется, стремятся избегать их и в JavaScript.

Если вы, при написании кода, не используете странных конструкций, или не начинаете строку с круглой или квадратной скобки, то вы, в 99.9% случаев, не допустите ошибку (если что — вас о возможной ошибке может предупредить линтер). К «странным конструкциям», например, можно отнести такую:

```
return
```

```
variable
```

Использовать ли точку с запятой, или нет — это личное дело каждого программиста. Автор этого руководства, например, говорит, что решил не использовать точки с запятой там, где они не нужны, в результате в примерах, приведённых здесь, они встречаются крайне редко.

Пробелы

JavaScript не обращает внимания на пробелы. Конечно, в определённых ситуациях отсутствие пробела приведёт к ошибке (равно как и неуместный пробел там, где его быть не должно), но очень часто между отсутствием пробела в некоем месте программы и наличием одного или нескольких пробелов нет никакой разницы. Похожее утверждение справедливо не только для пробелов, но и для знаков перевода строки, и для знаков табуляции. Особенно хорошо это заметно, например, на минифицированном коде. Взгляните, например, во что превращается код, обработанный с помощью [Closure Compiler](#).

В целом же надо отметить, что, форматируя код программы, лучше не впадать в крайности, придерживаясь некоего свода правил.

Чувствительность к регистру

JavaScript — регистрозависимый язык. Это означает, что он различает, например, имена переменных `something` и `Something`. То же самое касается любых идентификаторов.

Комментарии

В JavaScript можно использовать два типа комментариев. Первый тип — однострочные комментарии:

```
// Это комментарий
```

Они, как следует из названия, располагаются в одной строке. Комментарием считается всё, что идёт за символами `//`.

Второй тип — многострочные комментарии:

```
/*
```

Многострочный

комментарий

```
*/
```

Тут комментарием считается всё, что находится между комбинацией символов `/*` и `*/`.

Литералы и идентификаторы

Литералом называется некое значение, записанное в исходном коде программы. Например — это может быть строка, число, логическое значение, или более сложная структура — объектный литерал (позволяет создавать объекты, оформляется фигурными скобками) или литерал массива (позволяет создавать массивы, оформляется с помощью квадратных скобок). Вот несколько примеров:

```
5
```

```
'Test'
```

```
true
```

```
['a', 'b']
```

```
{color: 'red', shape: 'Rectangle'}
```

Особой пользы от запуска программы, в которой встречаются подобные конструкции, не будет. Для того чтобы работать с литералами в программах, их сначала присваивают переменным или передают функциям.

Идентификатором называется последовательность символов, которая может быть использована для идентификации переменной, функции, объекта. Она может начинаться с буквы, знака доллара (`$`) или со знака подчёркивания (`_`), может содержать цифры, и, если нужно, символы Unicode вроде смайликов (хотя, как уже было сказано, лучше так не делать). Вот несколько примеров идентификаторов:

```
Test
```

```
test
```

```
TEST
```

```
_test
```

```
Test1
```


\$test

Знак доллара обычно используется при создании идентификаторов, хранящих ссылки на элементы DOM.

Зарезервированные слова

Ниже приведён список слов, которые зарезервированы языком. Использовать их в качестве идентификаторов нельзя.

- break
- do
- instanceof
- typeof
- case
- else
- new
- var
- catch
- finally
- return
- void
- continue
- for
- switch
- while
- debugger
- function
- this
- with
- default
- if
- throw
- delete
- in
- try
- class
- enum
- extends
- super
- const
- export
- import
- implements
- let
- private
- public
- interface
- package
- protected
- static
- yield

Часть 3: переменные, типы данных, выражения, объекты

Переменные

Переменная представляет собой идентификатор, которому присвоено некое значение. К переменной можно обращаться в программе, работая таким образом с присвоенным ей значением.

Сама по себе переменная в JavaScript не содержит информацию о типе значений, которые будут в ней храниться. Это означает, что записав в переменную, например, строку, позже в неё можно записать число. Такая операция ошибки в программе не вызовет. Именно поэтому JavaScript иногда называют «нетипизированным» языком.

Прежде чем использовать переменную, её нужно объявить с использованием ключевого слова `var` или `let`. Если речь идёт о константе, применяется ключевое слово `const`. Объявить переменную и присвоить ей некое значение можно и не используя эти ключевые слова, но делать так не рекомендуется.

Ключевое слово `var`

До появления стандарта ES2015 использование ключевого слова `var` было единственным способом объявления переменных.

```
var a = 0
```

Если в этой конструкции опустить `var`, то значение будет назначено необъявленной переменной. Результат этой операции зависит от того, в каком режиме выполняется программа.

Так, если включён так называемый строгий режим (strict mode), подобное вызовет ошибку. Если строгий режим не включён, произойдёт неявное объявление переменной и она будет назначена глобальному объекту. В частности, это означает, что переменная, неявно объявленная таким образом в некоей функции, окажется доступной и после того, как функция завершит работу. Обычно же ожидается, что переменные, объявляемые в функциях, не «выходят» за их пределы. Выглядит это так:

```
function notVar() {  
    bNotVar = 1 //лучше так не делать  
}  
  
notVar()  
  
console.log(bNotVar)
```

В консоль попадёт 1, такого поведения от программы обычно никто не ждёт, выражение `bNotVar = 1` выглядит не как попытка объявления и инициализации переменной, а как попытка обратиться к переменной, находящейся во внешней по отношению к функции области видимости (это — вполне нормально). Как результат, неявное объявление переменных сбивает с толку того, кто читает код и может приводить к неожиданному поведению программ. Позже мы поговорим и о функциях, и об областях видимости, пока же постарайтесь всегда, когда смысл некоего выражения заключается в объявлении переменной, пользоваться специализированными ключевыми словами. Если в этом примере тело функции переписать в виде `var bNotVar = 1`, то попытка запустить вышеприведённый фрагмент кода приведёт к появлению сообщения об ошибке (его можно увидеть в консоли браузера). Выглядеть оно, например, может так: `Uncaught ReferenceError: bNotVar is not defined`. Смысл его сводится к тому, что программа не может работать с несуществующей переменной. Гораздо лучше, при первом запуске программы, увидеть такое сообщение об ошибке, чем писать непонятный код, который способен неожиданно себя вести.

Если, при объявлении переменной, её не инициализируют, не присваивают ей какого-либо значения, ей автоматически будет присвоено значение `undefined`.

```
var a //typeof a === 'undefined'
```

Переменные, объявленные с помощью ключевого слова `var`, можно многократно объявлять снова, назначая им новые значения (но это может запутать того, кто читает код).

```
var a = 1
```

```
var a = 2
```

В одном выражении можно объявить несколько переменных:

```
var a = 1, b = 2
```

Областью видимости переменной (scope) называют участок программы, в котором доступна (видима) эта переменная.

Переменная, инициализированная с помощью ключевого слова `var` за пределами какой-либо функции, назначается глобальному объекту. Она имеет глобальную область видимости и доступна из любого места программы. Если переменная объявлена с использованием ключевого слова `var` внутри функции, то она видна только внутри этой функции, являясь для неё локальной переменной.

Если в функции, с использованием `var`, объявлена переменная, имя которой совпадает с именем некоей переменной из глобальной области видимости, она «перекроет» глобальную переменную. То есть, при обращении к такой переменной внутри функции будет использоваться именно её локальный вариант.

Важно понимать, что блоки (области кода, заключённые в фигурные скобки) не создают новых областей видимости. Новая область видимости создаётся при вызове функции. Ключевое слово `var` имеет так называемую функциональную область видимости, а не блочную.

Если в коде функции объявлена некая переменная, она видна всему коду функции. Даже если переменная объявлена с помощью `var` в конце кода функции, обратиться к ней можно и в начале кода, так как в JavaScript работает механизм поднятия переменных (hoisting). Этот механизм «поднимает» объявления переменных, но не операции их инициализации. Это может стать источником путаницы, поэтому возьмите себе за правило объявлять переменные в начале функции.

Ключевое слово `let`

Ключевое слово `let` появилось в ES2015, его, упрощённо, можно назвать «блочной» версией `var`. Область видимости переменных, объявленных с помощью ключевого слова `let`, ограничивается блоком, оператором или выражением, в котором оно объявлено, а также вложенными блоками.

Если само слово «let» кажется не очень понятным, можно представить, что вместо него используется слово «пусть». Тогда выражение `let color = 'red'` можно перевести на английский так: «let the color be red», а на русский — так: «пусть цвет будет красным».

При использовании ключевого слова `let` можно избавиться от неоднозначностей, сопутствующих ключевому слову `var` (например, не удастся два раза, используя `let`, объявить одну и ту же переменную). Использование `let` за пределами функции, скажем, при инициализации циклов, не приводит к созданию глобальных переменных.

Например, такой код вызовет ошибку:

```
for (let i = 0; i < 5; i++) {  
    console.log(i)  
}  
  
console.log(i)
```

Если же, при инициализации цикла, счётчик `i` будет объявлен с использованием ключевого слова `var`, то `i` будет доступно и за пределами цикла, после того, как он завершит работу.

В наши дни, при разработке JS-программ на основе современных стандартов, вполне можно полностью отказаться от `var` и использовать только ключевые слова `let` и `const`.

Ключевое слово `const`

Значения переменных, объявленных с использованием ключевых слов `var` или `let`, могут быть перезаписаны. Если же вместо этих ключевых слов используется `const`, то объявленной и инициализированной с его помощью константе новое значение присвоить нельзя.

```
const a = 'test'
```

В данном примере константе `a` нельзя присвоить новое значение. Но надо отметить, что если `a` — это не примитивное значение, наподобие числа, а объект, использование ключевого слова `const` не защищает этот объект от изменений.

Когда говорят, что в переменную записан объект, на самом деле имеют в виду то, что в переменной хранится ссылка на объект. Эту вот ссылку изменить не удастся, а сам объект, к которому ведёт ссылка, можно будет изменить.

Ключевое слово `const` не делает объекты иммутабельными. Оно просто защищает от изменений ссылки на них, записанные в соответствующие константы. Вот как это выглядит:

```
const obj = {}  
  
console.log(obj.a)  
  
obj.a = 1 //работает  
  
console.log(obj.a)  
  
//obj = 5 //вызывает ошибку
```

В константу `obj`, при инициализации, записывается новый пустой объект. Попытка обращения к его свойству `a`, несуществующему, ошибки не вызывает. В консоль попадает `undefined`. После этого мы добавляем в объект новое свойство и снова пытаемся обратиться к нему. В этот раз в консоль попадает значение этого свойства — `1`. Если раскомментировать последнюю строку примера, то попытка выполнения этого кода приведёт к ошибке

Ключевое слово `const` очень похоже на `let`, в частности, оно обладает блочной областью видимости.

В современных условиях вполне допустимо использовать для объявления всех сущностей, значения которых менять не планируется, ключевое слово `const`, прибегая к `let` только в особых случаях. Почему? Всё дело в том, что лучше всего стремиться к использованию как можно более простых из доступных конструкций для того, чтобы не усложнять программы и избегать ошибок.

Типы данных

JavaScript иногда называют «нетипизированным» языком, но это не соответствует реальному положению дел. В переменные, и правда, можно записывать значения разных типов, но типы данных в JavaScript, всё-таки, есть. В частности, речь идёт о примитивных и об объектных типах данных.

Для того чтобы определить тип данных некоего значения, можно воспользоваться оператором `typeof`. Он возвращает строку, указывающую тип операнда.

Примитивные типы данных

Вот список примитивных типов данных JavaScript:

- `number` (число)
- `string` (строка)
- `boolean` (логическое значение)
- `null` (специальное значение `null`)
- `undefined` (специальное значение `undefined`)
- `symbol` (символ, используется в особых случаях, появился в ES6)

Здесь названия типов данных приведены в том виде, в котором их возвращает оператор `typeof`.

Поговорим о наиболее часто используемых типах данных из этого списка.

Тun number

Значения типа `number` в JavaScript представлены в виде 64-битных чисел двойной точности с плавающей запятой.

В коде числовые литералы представлены в виде целых и дробных чисел в десятичной системе счисления. Для записи чисел можно использовать и другие способы. Например, если в начале числового литерала имеется префикс `0x` — он воспринимается как число, записанное в шестнадцатеричной системе счисления. Числа можно записывать и в экспоненциальном представлении (в таких числах можно найти букву `e`).

Вот примеры записи целых чисел:

`10`

`5354576767321`

`0xCC` // шестнадцатеричное число

Вот дробные числа.

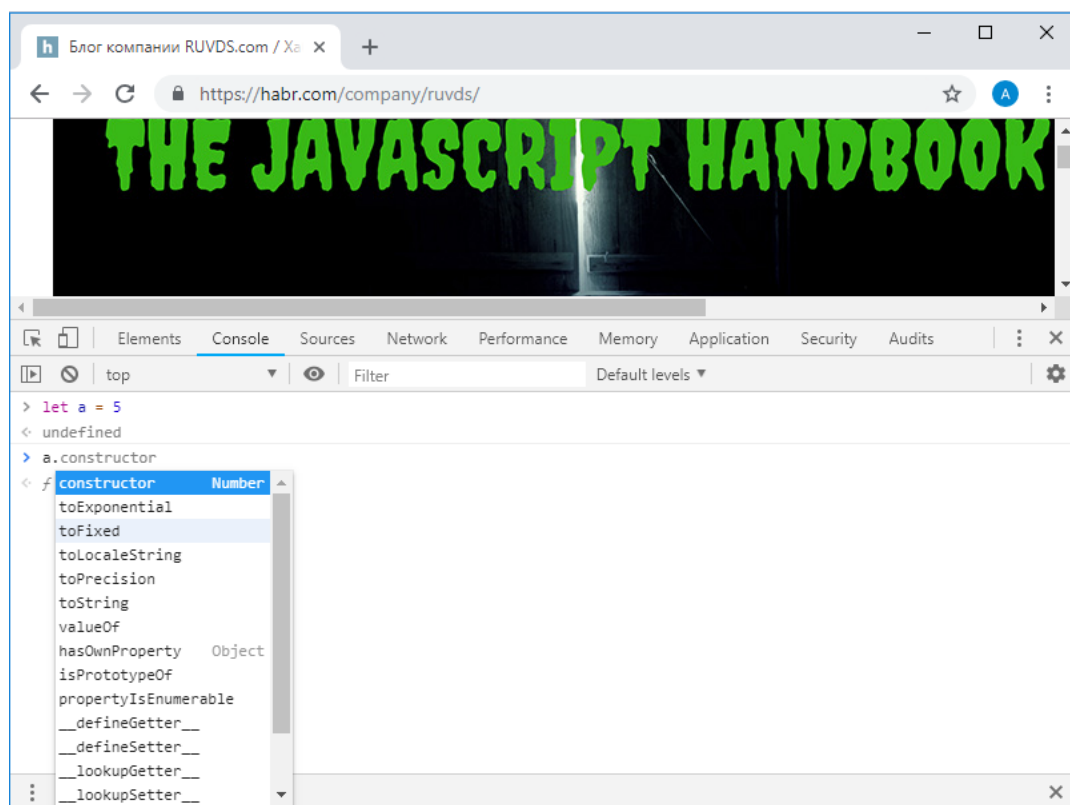
`3.14`

`.1234`

`5.2e4` // `5.2 * 10^4`

Числовые литералы (такое поведение характерно и для некоторых других примитивных типов), при попытке обращения к ним как к объектам, автоматически, на время выполнения операции, преобразуются в соответствующие объекты, которые называют «объектными обёртками». В данном случае речь идёт об объектной обёртке `Number`.

Вот, например, как выглядит попытка обратиться к переменной `a`, в которую записан числовой литерал, как к объекту, в консоли Google Chrome.



Подсказка по объектной обёртке Number

Если, например, воспользоваться методом `toString()` объекта типа `Number`, он возвратит строковое представление числа. Выглядит соответствующая команда, которую можно выполнить в консоли браузера (да и в обычном коде) так:

```
a.toString()
```

Обратите внимание на двойные скобки после имени метода. Если их не поставить, система не выдаст ошибку, но, вместо ожидаемого вывода, в консоли окажется нечто, совсем не похожее на строковое представление числа 5.

Глобальный объект `Number` можно использовать в виде конструктора, создавая с его помощью новые числа (правда, в таком виде его практически никогда не используют), им можно пользоваться и как самостоятельной сущностью, не создавая его экземпляры (то есть — некие числа, представляемые с его помощью). Например, его свойство `Number.MAX_VALUE` содержит максимальное числовое значение, представимое в JavaScript.

Tun string

Значения типа `string` представляют собой последовательности символов. Такие значения задают в виде строковых литералов, заключённых в одинарные или двойные кавычки.

```
'A string'
```

```
"Another string"
```

Строковые значения можно разбивать на несколько частей, используя символ обратной косой черты (backslash).

```
"A \n\nstring"
```

Строка может содержать так называемые escape-последовательности, интерпретируемые при выводе строки в консоль. Например, последовательность `\n` означает символ перевода строки. Символ обратной косой черты можно использовать и для того, чтобы добавлять кавычки в строки, заключённые в такие же кавычки.

Экранирование символа кавычки с помощью `\` приводит к тому, что система не воспринимает его как специальный символ.

```
'I\'m a developer'
```

Строки можно конкатенировать с использованием оператора `+`.

```
"A " + "string"
```

Шаблонные литералы

В ES2015 появились так называемые шаблонные литералы, или шаблонные строки. Они представляют собой строки, заключённые в обратные кавычки (```) и обладают некоторыми интересными свойствами.

```
`a string`
```

Например, в шаблонные литералы можно подставлять некие значения, являющиеся результатом вычисления JavaScript-выражений.

```
`a string with ${something}`
```

```
`a string with ${something+somethingElse}`
```

```
`a string with ${obj.something()}`
```

Использование обратных кавычек упрощает многострочную запись строковых литералов:

```
`a string
```

```
with
```

```
${something}`
```

Туп boolean

В JavaScript есть пара зарезервированных слов, использующихся при работе с логическими значениями — это `true` (истина), и `false` (ложь). Операции сравнения, например, такие, как `==`, `===`, `<`, `>`, возвращают `true` или `false`.

Логические выражения используются в конструкциях наподобие `if` и `while`, помогая управлять ходом выполнения программы.

При этом надо отметить, что там, где ожидается значение `true` или `false`, можно использовать и другие значения, которые автоматически расцениваются языком как истинные (`truthy`) или ложные (`falsy`).

В частности, ложными значениями являются следующие:

```
0
```

```
-0
```

```
NaN
```

```
undefined
```

```
null
```

```
' ' //пустая строка
```

Остальные значения являются истинными.

Tun null

В JavaScript имеется специальное значение `null`, которое указывает на отсутствие значения. Подобные значения используются и в других языках.

Tun undefined

Значение `undefined`, записанное в некую переменную, указывает на то, что эта переменная не инициализирована и значение для неё отсутствует.

Это значение автоматически возвращается из функций, результат работы которых не возвращается явно, с использованием ключевого слова `return`. Если функция принимает некий параметр, который, при её вызове, не указан, он также устанавливается в `undefined`.

Для того чтобы проверить значение на `undefined`, можно воспользоваться следующей конструкцией.

```
typeof variable === 'undefined'
```

Объекты

Все значения, не являющиеся примитивными, имеют объектный тип. Речь идёт о функциях, массивах, о том, что мы называем «объектами», и о многих других сущностях. В основе всех этих типов данных лежит тип `object`, и они, хотя и во многом друг от друга отличаются, имеют и много общего.

Выражения

Выражения — это фрагменты кода, которые можно обработать и получить на основе проведённых вычислений некое значение. В JavaScript существует несколько категорий выражений.

Арифметические выражения

В эту категорию попадают выражения, результатом вычисления которых являются числа.

```
1 / 2
```

```
i++
```

```
i -= 2
```

```
i * 2
```

Строковые выражения

Результатом вычисления таких выражений являются строки.

```
'A ' + 'string'
```

```
'A ' += 'string'
```

Первичные выражения

В эту категорию попадают литералы, константы, ссылки на идентификаторы.

```
2
```

```
0.02
```

```
'something'
```

```
true
```

```
false
```

```
this //контекст выполнения, ссылка на текущий объект
```

```
undefined
```


i //где i является переменной или константой

Сюда же можно отнести и некоторые ключевые слова и конструкции JavaScript.

function

class

function* //генератор

yield //команда приостановки/возобновления работы генератора

yield* //делегирование другому итератору или генератору

async function* //асинхронное функциональное выражение

await //организация ожидания выполнения асинхронной функции

/pattern/i //регулярное выражение

() //группировка

Выражения инициализации массивов и объектов

[] //литерал массива

{ } //объектный литерал

[1,2,3]

{a: 1, b: 2}

{a: {b: 1}}

Логические выражения

В логических выражениях используются логические операторы, результатом их вычисления оказываются логические значения.

a && b

a || b

!a

Выражения доступа к свойствам

Эти выражения позволяют обращаться к свойствам и методам объектов.

object.property //обращение к свойству (или методу) объекта

object[property]

object['property']

Выражения создания объектов

new object()

new a(1)

new MyRectangle('name', 2, {a: 4})

Выражения объявления функций

function() {}

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

(a, b) => a * b

a => a * 2

() => { return 2 }
```

Выражения вызова

Такие выражения используются для вызова функций или методов объектов.

```
a.x(2)

window.resize()
```

Работа с объектами

Выше мы уже сталкивались с объектами, говоря об объектных литералах, о вызове их методов, о доступе к их свойствам. Здесь мы поговорим об объектах подробнее, в частности, рассмотрим механизм прототипного наследования и использование ключевого слова `class`.

Прототипное наследование

JavaScript выделяется среди современных языков программирования тем, что поддерживает прототипное наследование. Большинство же объектно-ориентированных языков используют модель наследования, основанную на классах.

У каждого JavaScript-объекта есть особое свойство (`__proto__`), которое указывает на другой объект, являющийся его прототипом. Объект наследует свойства и методы прототипа.

Предположим, у нас имеется объект, созданный с помощью объектного литерала.

```
const car = {}
```

Или мы создали объект, воспользовавшись конструктором `Object`.

```
const car = new Object()
```

В любом из этих случаев прототипом объекта `car` будет `Object.prototype`.

Если создать массив, который тоже является объектом, его прототипом будет объект `Array.prototype`.

```
const list = []

//или так

const list = new Array()
```

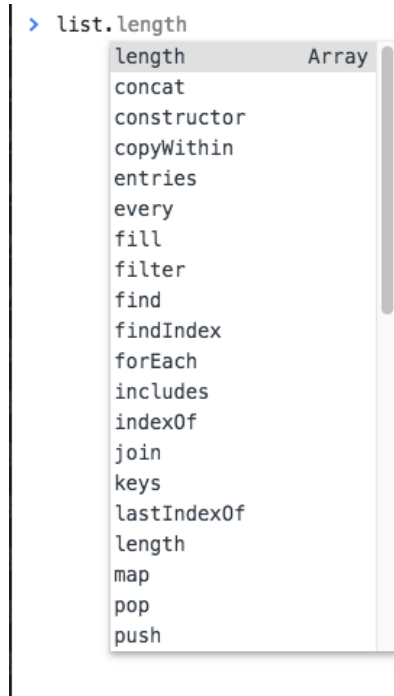
Проверить это можно следующим образом.

```
car.__proto__ == Object.prototype //true
car.__proto__ == new Object().__proto__ //true
list.__proto__ == Object.prototype //false
list.__proto__ == Array.prototype //true
list.__proto__ == new Array().__proto__ //true
```

Здесь мы пользовались свойством `__proto__`, оно не обязательно должно быть доступно разработчику, но обычно обращаться к нему можно. Надо отметить, что более надёжным способом получить прототип объекта является использование метода `getPrototypeOf()` глобального объекта `Object`.

```
Object.getPrototypeOf(new Object())
```

Все свойства и методы прототипа доступны объекту, имеющему этот прототип. Вот, например, как выглядит их список для массива.



Подсказка по массиву

Базовым прототипом для всех объектов является `Object.prototype`.

```
Array.prototype.__proto__ == Object.prototype
```

У `Object.prototype` прототипа нет.

То, что мы видели выше, является примером цепочки прототипов.

При попытке обращения к свойству или методу объекта, если такого свойства или метода у самого объекта нет, их поиск выполняется в его прототипе, потом — в прототипе прототипа, и так — до тех пор, пока искомое будет найдено, или до тех пор, пока цепочка прототипов не кончится.

Помимо создания объектов с использованием оператора `new` и применения объектных литералов или литералов массивов, создать экземпляр объекта можно с помощью метода `Object.create()`. Первый аргумент, передаваемый этому методу, представляет собой объект, который станет прототипом создаваемого с его помощью объекта.

```
const car = Object.create(Object.prototype)
```

Проверить, входит ли некий объект в цепочку прототипов другого объекта, можно с использованием метода `isPrototypeOf()`.

```
const list = []
```

```
Array.prototype.isPrototypeOf(list)
```

Функции-конструкторы

Выше мы создавали новые объекты, пользуясь уже имеющимися в языке функциями-конструкторами (при их вызове используется ключевое слово `new`). Такие функции можно создавать и самостоятельно. Рассмотрим пример.

```
function Person(name) {  
    this.name = name  
}  
  
Person.prototype.hello = function() {  
    console.log(this.name)  
}  
  
let person = new Person('Flavio')  
  
person.hello()  
  
console.log(Person.prototype.isPrototypeOf(person))
```

Здесь мы создаём функцию-конструктор. При её вызове создаётся новый объект, на который указывает ключевое слово `this` в теле конструктора. Мы добавляем в этот объект свойство `name` и записываем в него то, что передано конструктору. Этот объект возвращается из конструктора автоматически. С помощью функции-конструктора можно создать множество объектов, свойства `name` которых будут содержать то, что передано при их создании конструктору.

После создания конструктора мы добавляем в его прототип функцию, которая будет выводить в консоль значение свойства `name` объекта, созданного с помощью этой функции. Все объекты, созданные с помощью этого конструктора, будут иметь один и тот же прототип, а значит и пользоваться одной и той же функцией `hello()`. Это несложно проверить, создав ещё один объект типа `Person` и сравнив его функцию `hello()` с функцией уже имеющегося в примере объекта (имя функции в таком случае записывают без скобок).

Классы

В стандарте ES6 в JavaScript пришло такое понятие как «класс».

До этого в JavaScript можно было пользоваться лишь вышеописанным механизмом прототипного наследования. Этот механизм непривычно выглядел для программистов, пришедших в JS из других языков. Поэтому в языке и появились классы, которые, по сути, являются «синтаксическим сахаром» для прототипного механизма наследования. То есть, и объекты, созданные традиционным способом, и объекты, созданные с использованием классов, имеют прототипы.

Объявление класса

Вот как выглядит объявление класса.

```
class Person {  
    constructor(name) {  
        this.name = name
```

```
}

hello() {

    return 'Hello, I am ' + this.name + ' .'

}

}
```

У класса есть идентификатор, который можно использовать для создания новых объектов с применением конструкции `new ClassIdentifier()`.

При создании нового объекта вызывается метод `constructor`, ему передаются параметры.

В классе можно объявлять методы. В нашем случае `hello()` — это метод, который могут вызывать все объекты, созданные на основе класса. Вот как выглядит создание нового объекта с использованием класса `Person`.

```
const flavio = new Person('Flavio')

flavio.hello()
```

Наследование, основанное на классах

Классы могут расширять другие классы. Объекты, созданные на основе таких классов, будут наследовать и методы исходного класса, и методы, заданные в расширенном классе.

Если класс, расширяющий другой класс (наследник этого класса) имеет метод, имя которого совпадает с тем, который есть у класса-родителя, этот метод имеет преимущество перед исходным.

```
class Programmer extends Person {

    hello() {

        return super.hello() + ' I am a programmer.'

    }

}

const flavio = new Programmer('Flavio')

flavio.hello()
```

При вызове метода `hello()` в вышеприведённом примере будет возвращена строка `Hello, I am Flavio. I am a programmer.`

В классах не предусмотрено наличие переменных (свойств), свойства создаваемых с помощью классов объектов нужно настраивать в конструкторе.

Внутри класса можно обращаться к родительскому классу с использованием ключевого слова `super`.

Статические методы

Методы, описываемые в классе, можно вызывать, обращаясь к объектам, созданным на основе этого класса, но не к самому классу. Статические (`static`) методы можно вызывать, обращаясь непосредственно к классу.

Приватные методы

В JavaScript нет встроенного механизма, который позволяет объявлять приватные (частные, закрытые) методы. Это ограничение можно обойти, например, с использованием замыканий.

Геттеры и сеттеры

В классе можно описывать методы, предваряя их ключевыми словами `get` или `set`. Это позволяет создавать так называемые геттеры и сеттеры — функции, которые используются для управления доступом к свойствам объектов, созданных на основе класса. Геттер вызывается при попытке чтения значения псевдо-свойства, а сеттер — при попытке записи в него нового значения.

```
class Person {  
  
    constructor(name) {  
  
        this.userName = name  
  
    }  
  
    set name(value) {  
  
        this.userName = value  
  
    }  
  
    get name() {  
  
        return this.userName  
  
    }  
  
}
```

Часть 4: функции

Функции в JavaScript

Поговорим о функциях в JavaScript, сделаем их общий обзор и рассмотрим подробности о них, знание которых позволит вам эффективно ими пользоваться.

Функция — это самостоятельный блок кода, который можно, один раз объявив, вызывать столько раз, сколько нужно. Функция может, хотя это и необязательно, принимать параметры. Функции возвращают единственное значение.

Функции в JavaScript являются объектами, если точнее, то они являются объектами типа `Function`. Их ключевое отличие от обычных объектов, дающее им те исключительные возможности, которыми они обладают, заключается в том, что функции можно вызывать.

Кроме того, функции в JavaScript называют «функциями первого класса» так как их можно назначать переменным, их можно передавать другим функциям в качестве аргументов, их можно возвращать из других функций.

Сначала рассмотрим особенности работы с функциями и соответствующие синтаксические конструкции, которые существовали в языке до появления стандарта ES6 и актуальны до сих пор.

Вот как выглядит объявление функции (function declaration).

```
function doSomething(foo) {  
  
    //сделать что-нибудь  
  
}
```

В наши дни такие функции называют «обычными», отличая их от «стрелочных» функций, которые появились в ES6.

Функцию можно назначить переменной или константе. Такая конструкция называется функциональным выражением (function expression).

```
const doSomething = function(foo) {  
    //сделать что-нибудь  
}
```

Можно заметить, что в вышеприведённом примере функция назначена константе, но сама она имени не имеет. Такие функции называют анонимными. Подобным функциям можно назначать имена. В таком случае речь идёт об именованном функциональном выражении (named function expression).

```
const doSomething = function doSomFn(foo) {  
    //сделать что-нибудь  
}
```

Использование таких выражений повышает удобство отладки (в сообщениях об ошибках, где проводится трассировка стека, видно имя функции). Имя функции в функциональном выражении может понадобиться и для того, чтобы функция могла бы сама себя вызывать, без чего не обойтись при реализации рекурсивных алгоритмов.

В стандарте ES6 появились стрелочные функции (arrow function), которые особенно удобно использовать в виде так называемых «встроенных функций» (inline function) — в роли аргументов, передаваемых другим функциям (коллбэков).

```
const doSomething = foo => {  
    //сделать что-нибудь  
}
```

Стрелочные функции, помимо того, что структуры, используемые для их объявления, получаются более компактными, чем при использовании обычных функций, отличаются от них некоторыми важными особенностями, о которых мы поговорим ниже.

Параметры функций

Параметры представляют собой переменные, которые задаются на этапе объявления функции и будут содержать передаваемые ей значения (эти значения называют аргументами). Функции в JavaScript могут либо не иметь параметров, либо иметь один или несколько параметров.

```
const doSomething = () => {  
    //сделать что-нибудь  
}  
  
const doSomethingElse = foo => {  
    //сделать что-нибудь  
}  
  
const doSomethingElseAgain = (foo, bar) => {
```

```
//сделать что-нибудь  
}
```

Здесь показано несколько примеров стрелочных функций.

Начиная со стандарта ES6 у функций могут быть так называемые «параметры по умолчанию» (default parameters).

```
const doSomething = (foo = 1, bar = 'hey') => {  
  
  //сделать что-нибудь  
  
}
```

Они представляют собой стандартные значения, задаваемые параметрам функций в том случае, если при её вызове значения некоторых параметров не задаются. Например, функцию, показанную выше, можно вызвать как с передачей ей всех двух принимаемых ей параметров, так и другими способами.

```
doSomething(3)
```

```
doSomething()
```

В ES8 появилась возможность ставить запятую после последнего аргумента функции (это называется trailing comma). Эта возможность позволяет повысить удобство редактирования кода при использовании систем контроля версий в ходе разработки программ. Подробности об этом можно почитать [здесь](#) и [здесь](#).

Передаваемые функциям аргументы можно представлять в виде массивов. Для того чтобы разобрать эти аргументы можно воспользоваться оператором, который выглядит как три точки (это — так называемый «оператор расширения» или «оператор spread»). Вот как это выглядит.

```
const doSomething = (foo = 1, bar = 'hey') => {  
  
  //сделать что-нибудь  
  
}
```

```
const args = [2, 'ho!']
```

```
doSomething(...args)
```

Если функции нужно принимать много параметров, то запомнить порядок их следования может быть непросто. В таких случаях используются объекты с параметрами и возможности по деструктурированию объектов ES6.

```
const doSomething = ({ foo = 1, bar = 'hey' }) => {  
  
  //сделать что-нибудь  
  
  console.log(foo) // 2  
  
  console.log(bar) // 'ho!'  
  
}  
  
const args = { foo: 2, bar: 'ho!' }  
  
doSomething(args)
```

Этот приём позволяет, описывая параметры в виде свойств объекта и передавая функции объект, получить в функции доступ к параметрам по их именам без использования дополнительных конструкций. Подробнее об этом приёме можно почитать [здесь](#).

Значения, возвращаемые из функций

Все функции возвращают некое значение. Если команда возврата явно не задана — функция возвратит `undefined`.

```
const doSomething = (foo = 1, bar = 'hey') => {  
  //сделать что-нибудь  
}
```

```
console.log(doSomething())
```

Выполнение функции завершается либо после того, как оказывается выполненным весь код, который она содержит, либо после того, как в коде встречается ключевое слово `return`. Когда в функции встречается это ключевое слово, её работа завершается, а управление передаётся в то место, откуда была вызвана функция.

Если после ключевого слова `return` указать некое значение, то это значение возвращается в место вызова функции в качестве результата выполнения этой функции.

```
const doSomething = () => {  
  return 'test'  
}  
  
const result = doSomething() // result === 'test'
```

Из функции можно возвращать лишь одно значение. Для того чтобы получить возможность возврата нескольких значений, возвращать их можно либо в виде объекта, используя объектный литерал, либо в виде массива, а при вызове функции применять конструкцию деструктурирующего присваивания. Имена параметров при этом сохраняются. При этом, если нужно работать с объектом или массивом, возвращённым из функции, именно в виде объекта или массива, можно обойтись без деструктурирующего присваивания.

```
const doSomething = () => {  
  return ['Roger', 6]  
}
```

```
const [ name, age ] = doSomething()  
  
console.log(name, age) //Roger 6
```

Конструкцию `const [name, age] = doSomething()` можно прочитать следующим образом: «объявить константы `name` и `age` и присвоить им значения элементов массива, который возвратит функция».

Вот как то же самое выглядит с использованием объекта.

```
const doSomething = () => {  
  return {name: 'Roger', age: 6}  
}
```

```
const { name, age } = doSomething()

console.log(name, age) //Roger 6
```

Вложенные функции

Функции можно объявлять внутри других функций.

```
const doSomething = () => {

  const doSomethingElse = () => {}

  doSomethingElse()

  return 'test'

}
```

```
doSomething()
```

Область видимости вложенной функции ограничена внешней по отношению к ней функцией, её нельзя вызвать извне.

Методы объектов

Когда функции используются в качестве свойств объектов, такие функции называют методами объектов.

```
const car = {

  brand: 'Ford',

  model: 'Fiesta',

  start: function() {

    console.log(`Started`)

  }

}

car.start()
```

Ключевое слово this

Если сравнить стрелочные и обычные функции, используемые в качестве методов объектов, можно обнаружить их важное различие, заключающееся в смысле ключевого слова `this`. Рассмотрим пример.

```
const car = {

  brand: 'Ford',

  model: 'Fiesta',

  start: function() {

    console.log(`Started ${this.brand} ${this.model}`)

  },

  stop: () => {

    console.log(`Stopped ${this.brand} ${this.model}`)

  }

}
```

```

    }
}

car.start() //Started Ford Fiesta

car.stop() //Stopped undefined undefined

```

Как видно, вызов метода `start()` приводит ко вполне ожидаемому результату, а вот метод `stop()` явно работает неправильно.

Происходит это из-за того, что ключевое слово `this` по-разному ведёт себя при его использовании в стрелочных и обычных функциях. А именно, ключевое слово `this` в стрелочной функции содержит ссылку на контекст, включающий в себя функцию. В данном случае, если речь идёт о браузере, этим контекстом является объект `window`.

Вот как выглядит выполнение такого кода в консоли браузера.

```

const test = {

  fn: function() {

    console.log(this)

  },

  arrFn: () => {

    console.log(this)

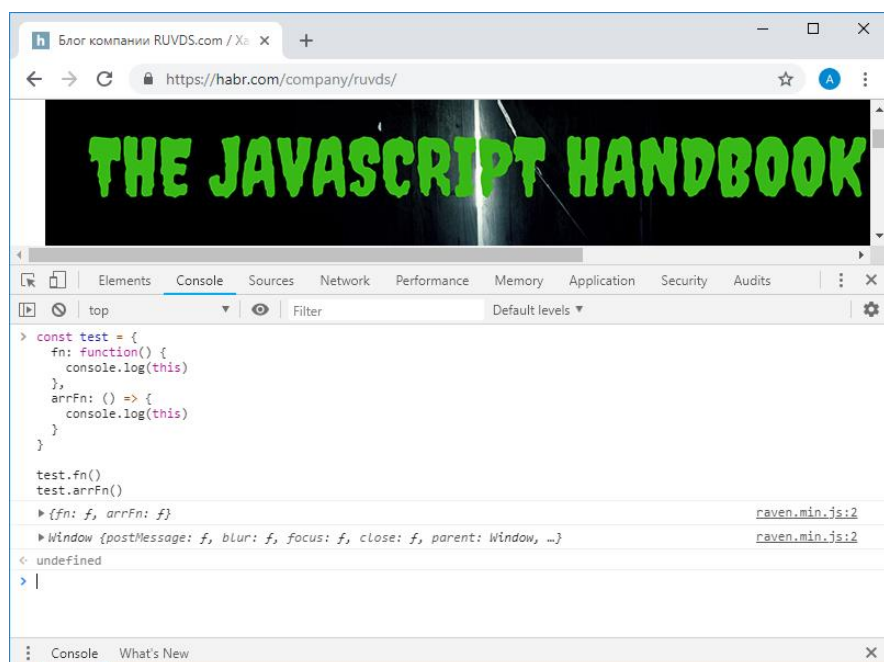
  }

}

```

```
test.fn()
```

```
test.arrFn()
```



Особенности ключевого слова `this` в обычных и стрелочных функциях

Как можно заметить, обращение к `this` в обычной функции означает обращение к объекту, а `this` в стрелочной функции указывает на `window`.

Всё это означает, что стрелочные функции не подходят на роль методов объектов и конструкторов (если попытаться использовать стрелочную функцию в роли конструктора — будет выдана ошибка `TypeError`).

Немедленно вызываемые функциональные выражения

Немедленно вызываемое функциональное выражение (Immediately Invoked Function Expression, IIFE) — это функция, которая автоматически вызывается сразу после её объявления.

```
;(function () {  
    console.log('executed')  
})()
```

Точка с запятой перед IIFE необязательна, но её использование позволяет застраховаться от ошибок, связанных с автоматической расстановкой точек с запятой.

В вышеприведённом примере в консоль попадёт слово `executed`, после чего IIFE завершит работу. IIFE, точно так же как и другие функции, могут возвращать результаты своей работы.

```
const something = (function () {  
    return 'IIFE'  
})()
```

```
console.log(something)
```

После выполнения этого простого примера в консоль попадёт строка `IIFE`, которая оказалась в константе `something` после выполнения немедленно вызываемого функционального выражения. Может показаться, что особой пользы от такой конструкции нет. Однако если в IIFE выполняются некие сложные вычисления, которые нужно выполнить лишь однажды, после чего соответствующие механизмы оказываются ненужными — полезность IIFE оказывается очевидной. А именно, при таком подходе после выполнения IIFE в программе будет доступен лишь возвращённый функцией результат. Кроме того, можно вспомнить, что функции способны возвращать другие функции и объекты. Речь идёт о замыканиях, о них мы поговорим ниже.

Поднятие функций

Перед выполнением JavaScript-кода производится его реорганизация. Мы уже говорили о механизме поднятия (hoisting) переменных, объявленных с использованием ключевого слова `var`. Похожий механизм действует и при работе с функциями. А именно, речь идёт о том, что объявления функций в ходе обработки кода перед его выполнением перемещаются в верхнюю часть их области видимости. В результате, например, оказывается, что вызвать функцию можно до её объявления.

```
doSomething() //did something  
  
function doSomething() {  
    console.log('did something')  
}
```

Если переместить вызов функции так, чтобы он шёл после её объявления, ничего не изменится.

Если же в похожей ситуации воспользоваться функциональным выражением, то похожий код выдаст ошибку.

```
doSomething() //TypeError

var doSomething = function () {

  console.log('did something')

}
```

В данном случае оказывается, что хотя объявление переменной `doSomething` и поднимается в верхнюю часть области видимости, это не относится к операции присваивания.

Если вместо `var` в похожей ситуации использовать ключевые слова `let` или `const`, такой код тоже работать не будет, правда, система выдаст другое сообщение об ошибке (`ReferenceError` а не `TypeError`), так как при использовании `let` и `const` объявления переменных и констант не поднимаются.

Стрелочные функции

Сейчас мы подробнее поговорим о стрелочных функциях, с которыми мы уже встречались. Их можно считать одним из наиболее значительных новшеств стандарта ES6, они отличаются от обычных функций не только внешним видом, но и особенностями поведения. В наши дни они используются чрезвычайно широко. Пожалуй, нет ни одного современного проекта, где они не использовались бы в подавляющем большинстве случаев. Можно сказать, что их появление навсегда изменило и внешний вид JS-кода и особенности его работы.

С чисто внешней точки зрения синтаксис объявления стрелочных функций оказывается компактнее синтаксиса обычных функций. Вот объявление обычной функции.

```
const myFunction = function () {

  //...

}
```

Вот объявление стрелочной функции, которое, в целом, если не учитывать особенности стрелочных функций, аналогично предыдущему.

```
const myFunction = () => {

  //...

}
```

Если тело стрелочной функции содержит лишь одну команду, результат которой возвращает эта функция, его можно записать без фигурных скобок и без ключевого слова `return`. Например, такая функция возвращает сумму переданных ей аргументов.

```
const myFunction = (a,b) => a + b

console.log(myFunction(1,2)) //3
```

Как видите, параметры стрелочных функций, как и в случае с обычными функциями, описывают в скобках. При этом, если такая функция принимает всего один параметр, его можно указать без скобок. Например, вот функция, которая возвращает результат деления переданного ей числа на 2.

```
const myFunction = a => a / 2

console.log(myFunction(8)) //4
```

В результате оказывается, что стрелочные функции очень удобно использовать в ситуациях, в которых нужны маленькие функции.

Неявный возврат результатов работы функции

Мы уже касались этой особенности стрелочных функций, но она настолько важна, что её следует обсудить подробнее. Речь идёт о том, что однострочные стрелочные функции поддерживают неявный возврат результатов своей работы. Пример возврата примитивного значения из однострочной стрелочной функции мы уже видели. Как быть, если такая функция должна вернуть объект? В таком случае фигурные скобки объектного литерала могут запутать систему, поэтому в теле функции используются круглые скобки.

```
const myFunction = () => ({value: 'test'})
```

```
const obj = myFunction()
```

```
console.log(obj.value) //test
```

Ключевое слово `this` и стрелочные функции

Выше, когда мы рассматривали особенности ключевого слова `this`, мы сравнивали обычные и стрелочные функции. Этот раздел призван обратить ваше внимание на важность их различий. Ключевое слово `this`, само по себе, может вызывать определённые сложности, так как оно зависит и от контекста выполнения кода, и от того, включен или нет строгий режим (strict mode).

Как мы уже видели, при использовании ключевого слова `this` в методе объекта, представленного обычной функцией, `this` указывает на объект, которому принадлежит метод. В таком случае говорят о привязке ключевого слова `this` к значению, представляющему собой контекст выполнения функции. В частности, если функция вызвана в виде метода объекта, то ключевое слово `this` привязано к этому объекту.

В случае же со стрелочными функциями оказывается так, что в них привязка `this` не выполняется, они пользуются ключевым словом `this` из содержащих их областей видимости. В результате их не рекомендуется использовать в качестве методов объектов.

Та же самая проблема возникает и при использовании функций в качестве обработчиков событий элементов DOM. Например, HTML-элемент `button` используют для описания кнопок. Событие `click` вызывается при щелчке мышью по кнопке. Для того чтобы отреагировать на это событие в коде, нужно сначала получить ссылку на соответствующий элемент, а потом назначить ему обработчик события `click` в виде функции. В качестве такого обработчика можно использовать и обычную функцию, и стрелочную. Но, если в обработчике событий нужно обращаться к тому элементу, для которого оно вызвано (то есть — к `this`), стрелочная функция тут не подойдёт, так как доступное в ней значение `this` указывает на объект `window`. Для того чтобы проверить это на практике, создайте HTML-страницу, код которой показан ниже, и понажимайте на кнопки.

```
<!DOCTYPE html>

<html>

  <body>

    <button id="fn">Function</button>

    <button id="arrowFn">Arrow function</button>

    <script>

      const f = document.getElementById("fn")

      f.addEventListener('click', function () {
```

```

        alert(this === f)

    })

    const af = document.getElementById("arrowFn")

    af.addEventListener('click', () => {

        alert(this === window)

    })

</script>

</body>

</html>

```

В данном случае при нажатии на эти кнопки будут появляться окна, содержащие `true`. Однако в обработчике события `click` кнопки с идентификатором `fn` проверяется равенство `this` самой кнопке, а в кнопке с идентификатором `arrowFn` проверяется равенство `this` и объекта `window`.

В результате, если в обработчике события HTML-элемента нужно обращаться к `this`, стрелочная функция для оформления такого обработчика не подойдёт.

Замыкания

Замыкания — это важная концепция в JavaScript. Фактически, если вы писали JS-функции, то вы пользовались и замыканиями. Замыкания применяются в некоторых паттернах проектирования — в том случае, если нужно организовать строгий контроль доступа к неким данным или функциям.

Когда функция вызывается, у неё есть доступ ко всему тому, что находится во внешней по отношению к ней области видимости. Но к тому, что объявлено внутри функции, извне доступа нет. То есть, если в функции была объявлена некая переменная (или другая функция), они недоступны внешнему коду ни во время выполнения функции, ни после завершения её работы. Однако если из функции вернуть другую функцию, то эта новая функция будет иметь доступ ко всему тому, что было объявлено в исходной функции. При этом всё это будет скрыто от внешнего кода в замыкании.

Рассмотрим пример. Вот функция, которая принимает имя собаки, после чего выводит его в консоль.

```

const bark = dog => {

    const say = `${dog} barked!`

    ;(() => console.log(say)) ()

}

bark(`Roger`) // Roger barked!

```

Значение, возвращаемое этой функцией нас пока не интересует, текст выводится в консоль с помощью IIFE, что в данном случае особой роли не играет, однако, это поможет нам увидеть связь между этой функцией и её вариантом, в котором, вместо вызова функции, которая выводит текст в консоль, мы эту функцию из переписанной функции `bark()` возвратим.

```

const prepareBark = dog => {

    const say = `${dog} barked!`

```

```
    return () => console.log(say)
  }

  const bark = prepareBark(`Roger`)

  bark() // Roger barked!
```

Результат работы код в двух случаях оказывается одинаковым. Но во втором случае то, что было передано исходной функции при её вызове (имя собаки, `Roger`), хранится в замыкании, после чего используется другой функцией, возвращённой из исходной.

Проведём ещё один эксперимент — создадим, пользуясь исходной функцией, две новых, для разных собак.

```
const prepareBark = dog => {
  const say = `${dog} barked!`

  return () => {
    console.log(say)
  }
}

const rogerBark = prepareBark(`Roger`)
const sydBark = prepareBark(`Syd`)

rogerBark()

sydBark()
```

Этот код выведет следующее.

```
Roger barked!
```

```
Syd barked!
```

Оказывается, что значение константы `say` привязано к функции, которая возвращена из функции `prepareBark()`.

Обратите внимание на то, что `say`, при повторном вызове `prepareBark()`, получает новое значение, при этом значение, записанное в `say` при первом вызове `prepareBark()`, не меняется. Речь идёт о том, что при каждом вызове этой функции создаётся новое замыкание.

Часть 5: массивы и циклы

Массивы

Массивы, объекты типа `Array`, развиваются вместе с остальными механизмами языка. Они представляют собой списки пронумерованных значений.

Первый элемент массива имеет индекс (ключ) 0, такой подход используется во многих языках программирования.

В этом разделе мы рассмотрим современные методы работы с массивами.

Инициализация массивов

Вот несколько способов инициализации массивов.

```
const a = []
```

```
const a = [1, 2, 3]
```

```
const a = Array.of(1, 2, 3)
```

```
const a = Array(6).fill(1) //инициализация каждого элемента массива, состоящего из 6 элементов, числом 1
```

Для того чтобы получить доступ к отдельному элементу массива, используют конструкцию, состоящую из квадратных скобок, в которых содержится индекс элемента массива. Элементы массивов можно как считывать, так и записывать.

```
const a = [1, 2, 3]
```

```
console.log(a) //[ 1, 2, 3 ]
```

```
const first = a[0]
```

```
console.log(first) //1
```

```
a[0] = 4
```

```
console.log(a) //[ 4, 2, 3 ]
```

Конструктор `Array` для объявления массивов использовать не рекомендуется.

```
const a = new Array() //не рекомендуется
```

```
const a = new Array(1, 2, 3) //не рекомендуется
```

Этот способ следует использовать лишь при объявлении [типизированных массивов](#).

Получение длины массива

Для того чтобы узнать длину массива, нужно обратиться к его свойству `length`.

```
const l = a.length
```

Проверка массива с использованием метода `every()`

Метод массивов `every()` можно использовать для организации проверки всех их элементов с использованием некоего условия. Если все элементы массива соответствуют условию, функция возвратит `true`, в противном случае она возвратит `false`.

Этому методу передаётся функция, принимающая аргументы `currentValue` (текущий элемент массива), `index` (индекс текущего элемента массива) и `array` (сам массив). Он может принимать и необязательное значение, используемое в качестве `this` при выполнении переданной ему функции.

Например, проверим, превышают ли значения всех элементов массива число 10.

```
const a = [11, 12, 13]
```

```
const b = [5, 6, 25]
```

```
const test = el => el > 10
```

```
console.log(a.every(test)) //true
```

```
console.log(b.every(test)) //false
```

Здесь нас, в функции `test()`, интересует лишь первый передаваемый ей аргумент, поэтому мы объявляем её, указывая лишь параметр `el`, в который и попадёт соответствующее значение.

Проверка массива с использованием метода `some()`

Этот метод очень похож на метод `every()`, но он возвращает `true`, если хотя бы один из элементов массива удовлетворяет условию, заданному переданной ему функцией.

Создание массива на основе существующего массива с использованием метода `map()`

Метод массивов `map()` позволяет перебирать массивы, применяя к каждому их элементу, переданную этому методу, функцию, преобразующую элемент, и создавать из полученных значений новые массивы. Вот, например, как получить новый массив, являющийся результатом умножения всех элементов исходного массива на 2.

```
const a = [1, 2, 3]

const double = el => el * 2

const doubleA = a.map(double)

console.log(a) //[ 1, 2, 3 ]

console.log(doubleA) //[ 2, 4, 6 ]
```

Фильтрация массива с помощью метода `filter()`

Метод `filter()` похож на метод `map()`, но он позволяет создавать новые массивы, содержащие лишь те элементы исходных массивов, которые удовлетворяют условию, задаваемому передаваемой методу `filter()` функцией.

Метод `reduce()`

Метод `reduce()` позволяет применить заданную функцию к аккумулятору и к каждому значению массива, сведя массив к единственному значению (это значение может иметь как примитивный, так и объектный тип). Этот метод принимает функцию, выполняющую преобразования, и необязательное начальное значение аккумулятора. Рассмотрим пример.

```
const a = [1, 2, 3, 4].reduce((accumulator, currentValue, currentIndex, array) => {

    return accumulator * currentValue

}, 1)

console.log(a) //24

//итерация 1: 1 * 1 = 1
//итерация 2: 1 * 2 = 2
//итерация 3: 2 * 3 = 6
//итерация 4: 6 * 4 = 24
```

Здесь мы ищем произведение всех элементов массива, описанного с помощью литерала, задавая в качестве начального значения аккумулятора 1.

Перебор массива с помощью метода `forEach()`

Метод массивов `forEach()` можно использовать для перебора значений массивов и для выполнения над ними неких действий, задаваемых передаваемой методу функцией. Например, выведем, по одному, элементы массива в консоль.

```
const a = [1, 2, 3]
```

```
a.forEach(el => console.log(el))

//1

//2

//3
```

Если при переборе массива нужно остановить или прервать цикл, то при использовании `forEach()` придётся выбрасывать исключение. Поэтому если в ходе решения некоей задачи может понадобиться прерывание цикла, лучше всего выбрать какой-нибудь другой способ перебора элементов массива.

Перебор массива с использованием оператора `for...of`

Оператор `for...of` появился в стандарте ES6. Он позволяет перебирать итерируемые объекты (в том числе — массивы). Вот как им пользоваться.

```
const a = [1, 2, 3]

for (let v of a) {

    console.log(v)

}

//1

//2

//3
```

На каждой итерации цикла в переменную `v` попадает очередной элемент массива `a`.

Перебор массива с использованием оператора `for`

Оператор `for` позволяет организовывать циклы, которые, в частности, можно использовать и для перебора (или инициализации) массивов, обращаясь к их элементам по индексам. Обычно индекс очередного элемента получают, пользуясь счётчиком цикла.

```
const a = [1, 2, 3]

for (let i = 0; i < a.length; i += 1) {

    console.log(a[i])

}

//1

//2

//3
```

Если, в ходе выполнения цикла, нужно пропустить его итерацию, можно воспользоваться командой `continue`. Для досрочного завершения цикла можно воспользоваться командой `break`. Если в цикле, например, расположенном в некоей функции, использовать команду `return`, выполнение цикла и функции завершится, а возвращённое с помощью `return` значение попадёт туда, откуда была вызвана функция.

Метод `@@iterator`

Этот метод появился в стандарте ES6. Он позволяет получать так называемый «итератор объекта» — объект, который в данном случае позволяет организовывать перебор элементов массива. Итератор массива можно получить, воспользовавшись символом (такие символы называют «известными символами»)

`Symbol.iterator`. После получения итератора можно обращаться к его методу `next()`, который, при каждом его вызове, возвращает структуру данных, содержащую очередной элемент массива.

```
const a = [1, 2, 3]

let it = a[Symbol.iterator]()

console.log(it.next().value) //1
console.log(it.next().value) //2
console.log(it.next().value) //3
```

Если вызвать метод `next()` после того, как будет достигнут последний элемент массива, он возвратит, в качестве значения элемента, `undefined`. Объект, возвращаемый методом `next()`, содержит свойства `value` и `done`. Свойство `done` принимает значение `true` до тех пор, пока не будет достигнут последний элемент массива. В нашем случае, если вызвать `it.next()` в четвёртый раз, он возвратит объект `{ value: undefined, done: true }`, в то время как при трёх предыдущих вызовах этот объект имел вид `{ value: значение, done: false }`.

Метод массивов `entries()` возвращает итератор, который позволяет перебирать пары ключ-значение массива.

```
const a = [1, 2, 3]

let it = a.entries()

console.log(it.next().value) //[0, 1]
console.log(it.next().value) //[1, 2]
console.log(it.next().value) //[2, 3]
```

Метод `keys()` позволяет перебирать ключи массива.

```
const a = [1, 2, 3]

let it = a.keys()

console.log(it.next().value) //0
console.log(it.next().value) //1
console.log(it.next().value) //2
```

Добавление элементов в конец массива

Для добавления элементов в конец массива используют метод `push()`.

```
a.push(4)
```

Добавление элементов в начало массива

Для добавления элементов в начало массива используют метод `unshift()`.

```
a.unshift(0)

a.unshift(-2, -1)
```

Удаление элементов массива

Удалить элемент из конца массива, одновременно возвратив этот элемент, можно с помощью метода `pop()`.

```
a.pop()
```

Аналогичным образом, с помощью метода `shift()`, можно удалить элемент из начала массива.

```
a.shift()
```

То же самое, но уже с указанием позиции удаления элементов и их количества, делается с помощью метода `splice()`.

```
a.splice(0, 2) // удаляет и возвращает 2 элемента из начала массива
```

```
a.splice(3, 2) // удаляет и возвращает 2 элемента, начиная с индекса 3
```

Удаление элементов массива и вставка вместо них других элементов

Для того чтобы, воспользовавшись одной операцией, удалить некие элементы массива и вставить вместо них другие элементы, используется уже знакомый вам метод `splice()`.

Например, здесь мы удаляем 3 элемента массива начиная с индекса 2, после чего в то же место добавляем два других элемента

```
const a = [1, 2, 3, 4, 5, 6]
```

```
a.splice(2, 3, 'a', 'b')
```

```
console.log(a) //[ 1, 2, 'a', 'b', 6 ]
```

Объединение нескольких массивов

Для объединения нескольких массивов можно воспользоваться методом `concat()`, возвращающим новый массив.

```
const a = [1, 2]
```

```
const b = [3, 4]
```

```
const c = a.concat(b)
```

```
console.log(c) //[ 1, 2, 3, 4 ]
```

Поиск элементов в массиве

В стандарте ES5 появился метод `indexOf()`, который возвращает индекс первого вхождения искомого элемента массива. Если элемент в массиве найти не удаётся — возвращается `-1`.

```
const a = [1, 2, 3, 4, 5, 6, 7, 5, 8]
```

```
console.log(a.indexOf(5)) //4
```

```
console.log(a.indexOf(23)) //-1
```

Метод `lastIndexOf()` возвращает индекс последнего вхождения элемента в массив, или, если элемент не найден, `-1`.

```
const a = [1, 2, 3, 4, 5, 6, 7, 5, 8]
```

```
console.log(a.lastIndexOf(5)) //7
```

```
console.log(a.lastIndexOf(23)) //-1
```

В ES6 появился метод массивов `find()`, который выполняет поиск по массиву с использованием передаваемой ему функции. Если функция возвращает `true`, метод возвращает значение первого найденного элемента. Если элемент найти не удаётся, функция возвратит `undefined`.

Выглядеть его использование может следующим образом.

```
a.find(x => x.id === my_id)
```

Здесь в массиве, содержащем объекты, осуществляется поиск элемента, свойство `id` которого равняется заданному.

Метод `findIndex()` похож на `find()`, но он возвращает индекс найденного элемента или `undefined`.

В ES7 появился метод `includes()`, который позволяет проверить наличие некоего элемента в массиве. Он возвращает `true` или `false`, найдя или не найдя интересующий программиста элемент.

```
a.includes(value)
```

С помощью этого метода можно проверять на наличие некоего элемента не весь массив, а лишь некоторую его часть, начинающуюся с заданного при вызове этого метода индекса. Индекс задаётся с помощью второго, необязательного, параметра этого метода.

```
a.includes(value, i)
```

Получение фрагмента массива

Для того чтобы получить копию некоего фрагмента массива в виде нового массива, можно воспользоваться методом `slice()`. Если этот метод вызывается без аргументов, то возвращённый массив окажется полной копией исходного. Он принимает два необязательных параметра. Первый задаёт начальный индекс фрагмента, второй — конечный. Если конечный индекс не задан, то массив копируется от заданного начального индекса до конца.

```
const a = [1, 2, 3, 4, 5, 6, 7, 8, 9]

console.log(a.slice(4)) // [ 5, 6, 7, 8, 9 ]

console.log(a.slice(3,7)) // [ 4, 5, 6, 7 ]
```

Сортировка массива

Для организации сортировки элементов массива в алфавитном порядке (0–9A–Za–z) используется метод `sort()` без передачи ему аргументов.

```
const a = [1, 2, 3, 10, 11]

a.sort()

console.log(a) // [ 1, 10, 11, 2, 3 ]

const b = [1, 'a', 'Z', 3, 2, 11]

b.sort()

console.log(b) // [ 1, 11, 2, 3, 'Z', 'a' ]
```

Этому методу можно передать функцию, задающую порядок сортировки. Функция принимает, для сравнения двух элементов, параметры `a` и `b`. Она возвращает отрицательное число в том случае, если `a` меньше `b` по какому-либо критерию, 0 — если они равны, и положительное число — если `a` больше `b`. При написании подобной функции для сортировки числовых массивов она может вернуть результат вычитания `a` и `b`. Так, возврат результата вычисления выражения `a - b` означает сортировку массива по возрастанию, возврат результата вычисления выражения `b - a` даст сортировку массива по убыванию.

```
const a = [1, 10, 3, 2, 11]

console.log(a.sort((a, b) => a - b)) // [ 1, 2, 3, 10, 11 ]

console.log(a.sort((a, b) => b - a)) // [ 11, 10, 3, 2, 1 ]
```

Для того чтобы обратить порядок следования элементов массива можно воспользоваться методом `reverse()`. Он, так же, как и `sort()`, модифицирует массив для которого вызывается.

Получение строкового представления массива

Для получения строкового представления массива можно воспользоваться его методом `toString()`.

```
a.toString()
```

Похожий результат даёт метод `join()`, вызванный без аргументов.

```
a.join()
```

Ему, в качестве аргумента, можно передать разделитель элементов.

```
const a = [1, 10, 3, 2, 11]

console.log(a.toString()) //1,10,3,2,11

console.log(a.join()) //1,10,3,2,11

console.log(a.join(' ')) //1, 10, 3, 2, 11
```

Создание копий массивов

Для создания копии массива путём копирования в новый массив значений исходного массива можно воспользоваться методом `Array.from()`. Он подходит и для создания массивов из массивоподобных объектов (из строк, например).

```
const a = 'a string'

const b = Array.from(a)

console.log(b) //[ 'a', ' ', 's', 't', 'r', 'i', 'n', 'g' ]
```

Метод `Array.of()` тоже можно использовать для копирования массивов, а также для «сборки» массивов из различных элементов. Например, для копирования элементов одного массива в другой можно воспользоваться следующей конструкцией.

```
const a = [1, 10, 3, 2, 11]

const b = Array.of(...a)

console.log(b) // [ 1, 10, 3, 2, 11 ]
```

Для копирования элементов массива в некое место самого этого массива используется метод `copyWithin()`. Его первый аргумент задаёт начальный индекс целевой позиции, второй — начальный индекс позиции источника элементов, а третий параметр, необязательный, указывает конечный индекс позиции источника элементов. Если его не указать, в указанное место массива будет скопировано всё, начиная от начального индекса позиции источника до конца массива.

```
const a = [1, 2, 3, 4, 5]

a.copyWithin(0, 2)

console.log(a) //[ 3, 4, 5, 4, 5 ]
```

Циклы

Выше, говоря о массивах, мы уже сталкивались с некоторыми способами организации циклов. Однако циклы в JavaScript используются не только для работы с массивами, да и рассмотрели мы далеко не все их виды. Поэтому

сейчас мы уделим некоторое время рассмотрению разных способов организации циклов в JavaScript и поговорим об их особенностях.

Цикл `for`

Рассмотрим пример применения этого цикла.

```
const list = ['a', 'b', 'c']

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

    console.log(list[i]) //значения, хранящиеся в элементах циклов

    console.log(i) //индексы

}
```

Как уже было сказано, прерывать выполнение такого цикла можно, используя команду `break`, а пропускать текущую итерацию и переходить сразу к следующей можно с помощью команды `continue`.

Цикл `forEach`

Этот цикл мы тоже обсуждали. Приведём пример перебора массива с его помощью.

```
const list = ['a', 'b', 'c']

list.forEach((item, index) => {

    console.log(item) //значение

    console.log(index) //индекс

})

//если индексы элементов нас не интересуют, можно обойтись и без них

list.forEach(item => console.log(item))
```

Напомним, что для прерывания такого цикла надо выбрасывать исключение, то есть, если при использовании цикла может понадобиться прервать его, лучше выбрать какой-нибудь другой цикл.

Цикл `do...while`

Это — так называемый «цикл с постусловием». Такой цикл будет выполнен как минимум один раз до проверки условия завершения цикла.

```
const list = ['a', 'b', 'c']

let i = 0

do {

    console.log(list[i]) //значение

    console.log(i) //индекс

    i = i + 1

} while (i < list.length)
```

Его можно прерывать с использованием команды `break`, можно переходить на его следующую итерацию командой `continue`.

Цикл `while`

Это — так называемый «цикл с предусловием». Если, на входе в цикл, условие продолжения цикла ложно, он не будет выполнен ни одного раза.

```
const list = ['a', 'b', 'c']

let i = 0

while (i < list.length) {

  console.log(list[i]) //значение

  console.log(i) //индекс

  i = i + 1

}
```

Цикл `for...in`

Этот цикл позволяет перебирать все перечислимые свойства объекта по их именам.

```
let object = {a: 1, b: 2, c: 'three'}

for (let property in object) {

  console.log(property) //имя свойства

  console.log(object[property]) //значение свойства

}
```

Цикл `for...of`

Цикл `for...of` совмещает в себе удобство цикла `forEach` и возможность прерывать его работу штатными средствами.

```
//перебор значений

for (const value of ['a', 'b', 'c']) {

  console.log(value) //значение

}

//перебор значений и получение индексов с помощью `entries()`

for (const [index, value] of ['a', 'b', 'c'].entries()) {

  console.log(index) //индекс

  console.log(value) //значение

}
```

Обратите внимание на то, что здесь, в заголовке цикла, используется ключевое слово `const`, а не, как можно было бы ожидать, `let`. Если внутри блока цикла переменные не нужно переназначать, то `const` нам вполне подходит.

Если сравнить циклы `for...in` и `for...of`, то окажется, что `for...in` перебирает имена свойств, а `for...of` — значения свойств.

Циклы и области видимости

С циклами и с областями видимости переменных связана одна особенность JavaScript, которая может доставить разработчику некоторые проблемы. Для того чтобы с этими проблемами разобраться, поговорим о циклах, об областях видимости, и о ключевых словах `var` и `let`.

Рассмотрим пример.

```
const operations = []

for (var i = 0; i < 5; i++) {
  operations.push(() => {
    console.log(i)
  })
}

for (const operation of operations) {
  operation()
}
```

В цикле производится 5 итераций, на каждой из которых в массив `operations` добавляется новая функция. Эта функция выводит в консоль значение счётчика цикла — `i`. После того, как функции добавлены в массив, мы этот массив перебираем и вызываем функции, являющиеся его элементами.

Выполняя подобный код можно ожидать результата, показанного ниже.

```
0
1
2
3
4
```

Но на самом деле он выводит следующее.

```
5
5
5
5
5
```

Почему это так? Всё дело в том, что в качестве счётчика цикла мы используем переменную, объявленную с использованием ключевого слова `var`.

Так как объявления подобных переменных поднимаются в верхнюю часть области видимости, вышеприведённый код аналогичен следующему.

```
var i;
```

```

const operations = []

for (i = 0; i < 5; i++) {
  operations.push(() => {
    console.log(i)
  })
}

for (const operation of operations) {
  operation()
}

```

В результате оказывается, что в цикле `for...of`, в котором мы перебираем массив, переменная `i` всё ещё видна, она равна 5, в результате, ссылаясь на `i` во всех функциях, мы выводим число 5.

Как изменить поведение программы таким образом, чтобы она делала бы то, что от неё ожидается?

Самое простое решение этой проблемы заключается в использовании ключевого слова `let`. Оно, как мы уже говорили, появилось в ES6, его использование позволяет избавиться от некоторых странностей, характерных для `var`.

В частности, в вышеприведённом примере достаточно изменить `var` на `let` и всё заработает так, как нужно.

```

const operations = []

for (let i = 0; i < 5; i++) {
  operations.push(() => {
    console.log(i)
  })
}

for (const operation of operations) {
  operation()
}

```

Теперь на каждой итерации цикла каждая функция, добавленная в массив `operations`, получает собственную копию `i`. Помните о том, что в данной ситуации нельзя использовать ключевое слово `const`, так как значение `i` в цикле меняется.

Ещё один способ решения этой проблемы, который часто применялся до появления стандарта ES6, когда ключевого слова `let` ещё не было, заключается в использовании IIFE.

При таком подходе значение `i` сохраняется в замыкании, а в массив попадает функция, возвращаемая IIFE и имеющая доступ к замыканию. Эту функцию можно выполнить тогда, когда в ней возникнет необходимость. Вот как это выглядит.

```

const operations = []

for (var i = 0; i < 5; i++) {

```

```
operations.push((j) => {  
    return () => console.log(j)  
})(i))  
}  
  
for (const operation of operations) {  
    operation()  
}
```

Часть 6: исключения, точка с запятой, шаблонные литералы

Обработка исключений

Когда при выполнении кода возникает какая-нибудь проблема, в JavaScript она выражается в виде исключения. Если не предпринять меры по обработке исключений, то, при их возникновении, выполнение программы останавливается, а в консоль выводится сообщение об ошибке.

Рассмотрим следующий фрагмент кода.

```
let obj = {value: 'message text'}  
  
let notObj  
  
let fn = (a) => a.value  
  
console.log(fn(obj)) //message text  
  
console.log('Before') //Before  
  
console.log(fn(notObj)) //ошибка, выполнение программы останавливается  
  
console.log('After')
```

Здесь у нас имеется функция, которую планируется использовать для обработки объектов, имеющих свойство `value`. Она возвращает это свойство. Если использовать эту функцию по назначению, то есть — передать ей такой объект, на работу с которым она рассчитана, при её выполнении ошибок выдано не будет. А вот если передать ей нечто неподходящее, в нашем случае — объявленную, но неинициализированную переменную, то при попытке обратиться к свойству `value` значения `undefined` произойдёт ошибка. В консоль попадёт сообщение об ошибке, выполнение программы остановится.

Вот как это выглядит при запуске данного кода в среде Node.js.

```
Командная строка

c:\js>node exceptions.js
message text
Before
c:\js\exceptions.js:3
let fn = (a) => a.value
               ^

TypeError: Cannot read property 'value' of undefined
    at fn (c:\js\exceptions.js:3:19)
    at Object.<anonymous> (c:\js\exceptions.js:8:13)
    at Module._compile (internal/modules/cjs/loader.js:702:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:713:10)
    at Module.load (internal/modules/cjs/loader.js:612:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:551:12)
    at Function.Module._load (internal/modules/cjs/loader.js:543:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:744:10)
    at startup (internal/bootstrap/node.js:240:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:564:3)

c:\js>
```

Исключение *TypeError* в *Node.js*

Если нечто подобное встретится в JS-коде веб-страницы, в консоль браузера попадёт похожее сообщение. Если такое произойдёт в реальной программе, скажем — в коде веб-сервера, подобное поведение крайне нежелательно. Хорошо было бы иметь механизм, который позволяет, не останавливая программу, перехватить ошибку, после чего принять меры по её исправлению. Такой механизм в JavaScript существует, он представлен конструкцией `try...catch`.

Конструкция `try...catch`

Конструкция `try...catch` позволяет перехватывать и обрабатывать исключения. А именно, в неё входит блок `try`, в который включают код, способный вызвать ошибку, и блок `catch`, в который передаётся управление при возникновении ошибки. В блоки `try` не включают абсолютно весь код программы. Туда помещают те его участки, которые могут вызвать ошибки времени выполнения. Например — вызовы функций, которым приходится работать с некими данными, полученными из внешних источников. Если структура таких данных отличается от той, которую ожидает функция, возможно возникновение ошибки. Вот как выглядит схема конструкции `try...catch`.

```
try {
    //строки кода, которые могут вызвать ошибку
} catch (e) {
    //обработка ошибки
}
```

Если код выполняется без ошибок — блок `catch` (обработчик исключения) не выполняется. Если же возникает ошибка — туда передаётся объект ошибки и там выполняются некие действия по борьбе с этой ошибкой.

Применим эту конструкцию в нашем примере, защитив с её помощью опасные участки программы — те, в которых вызывается функция `fn()`.

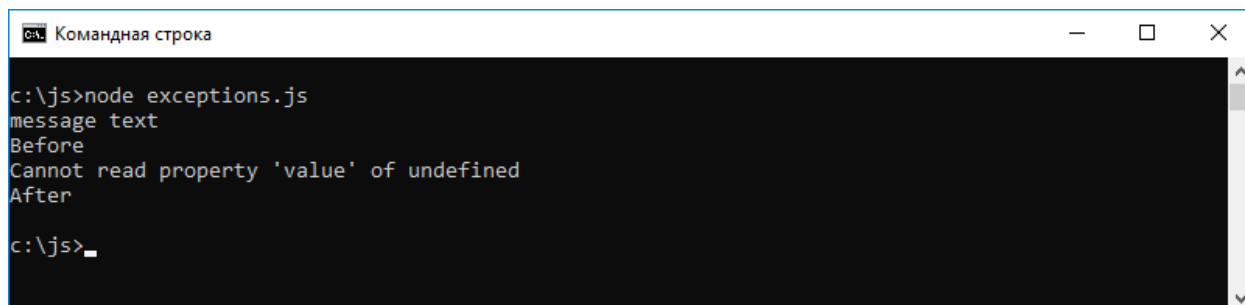
```
let obj = {value: 'message text'}

let notObj

let fn = (a) => a.value
```

```
try {  
    console.log(fn(obj))  
} catch (e) {  
    console.log(e.message)  
}  
  
console.log('Before') //Before  
  
try {  
    console.log(fn(notObj))  
} catch (e) {  
    console.log(e.message) //Cannot read property 'value' of undefined  
}  
  
console.log('After') //After
```

Посмотрим на результаты выполнения этого кода в среде Node.js.



```
Командная строка  
c:\js>node exceptions.js  
message text  
Before  
Cannot read property 'value' of undefined  
After  
c:\js>
```

Обработка ошибки в Node.js

Как видите, если сравнить этот пример с предыдущим, теперь выполняется весь код, и тот, что расположен до проблемной строки, и тот, что расположен после неё. Мы «обрабатываем» ошибку, просто выводя в консоль значения свойства `message` объекта типа [Error](#). В чём будет заключаться обработка ошибки, возникшей в реально используемом коде, зависит от ошибки.

Выше мы обсудили блок `try...catch`, но, на самом деле, эта конструкция включает в себя ещё один блок — `finally`.

Блок `finally`

Блок `finally` содержит код, который выполняется независимо от того, возникла или нет ошибка в коде, выполняющемся в блоке `try`. Вот как это выглядит.

```
try {  
    //строки кода  
} catch (e) {  
    //обработка ошибки
```

```
} finally {  
  
    //освобождение ресурсов  
  
}
```

Блок `finally` можно использовать и в том случае, если в блоке `try...catch...finally` отсутствует блок `catch`. При таком подходе он используется так же, как и в конструкции с блоком `catch`, например — для освобождения ресурсов, занятых в блоке `try`.

Вложенные блоки `try`

Блоки `try` могут быть вложены друг в друга. При этом исключение обрабатывается в ближайшем блоке `catch`.

```
try {  
  
    //строки кода  
  
    try {  
  
        //другие строки кода  
  
    } finally {  
  
        //ещё какой-то код  
  
    }  
  
} catch (e) {  
  
}
```

В данном случае, если исключение возникнет во внутреннем блоке `try`, обработано оно будет во внешнем блоке `catch`.

Самостоятельное генерирование исключений

Исключения можно генерировать самостоятельно, пользуясь инструкцией `throw`. Вот как это выглядит.

```
throw value
```

После того, как выполняется эта инструкция, управление передаётся в ближайший блок `catch`, или, если такого блока найти не удастся, выполнение программы прекращается. Значением исключения может быть всё что угодно. Например — определённый пользователем объект ошибки.

О точках с запятой

Использовать точки с запятой в JavaScript-коде необязательно. Некоторые программисты обходятся без них, полагаясь на автоматическую систему их расстановки, и ставя их только там, где это совершенно необходимо. Некоторые предпочитают ставить их везде, где это возможно. Автор этого материала относит себя к той категории программистов, которые стремятся обходиться без точек с запятой. Он говорит, что решил обходиться без них осенью 2017 года, настроив Prettier так, чтобы он удалял их везде, где без их явной вставки можно обойтись. По его мнению код без точек с запятой выглядит естественнее и его легче читать.

Пожалуй, можно сказать, что сообщество JS-разработчиков разделено, по отношению к точкам с запятой, на два лагеря. При этом существуют и руководства по стилю JavaScript, которые предписывают явную расстановку точек с запятой, и руководства, которые рекомендуют обходиться без них.

Всё это возможно из-за того, что в JavaScript существует система автоподстановки точек с запятой (Automatic Semicolon Insertion, ASI). Однако, то, что в JS коде, во многих ситуациях, можно обойтись без этих символов, и то, что точки с запятой расставляются автоматически, при подготовке кода к выполнению, не означает, что

программисту не нужно знать правила, по которым это происходит. Незнание этих правил приводит к появлению ошибок.

Правила автоподстановки точек с запятой

Парсер JavaScript-кода автоматически добавляет точки с запятой при разборе текста программы в следующих ситуациях:

1. Когда следующая строка начинается с кода, который прерывает текущий код (код некоей команды может располагаться на нескольких строках).
2. Когда следующая строка начинается с символа `}`, который закрывает текущий блок.
3. Когда обнаружен конец файла с кодом программы.
4. В строке с командой `return`.
5. В строке с командой `break`.
6. В строке с командой `throw`.
7. В строке с командой `continue`.

Примеры кода, который работает не так, как ожидается

Вот некоторые примеры, иллюстрирующие вышеприведённые правила. Например, как вы думаете, что будет выведено в результате выполнения следующего фрагмента кода?

```
const hey = 'hey'

const you = 'hey'

const heyYou = hey + ' ' + you

['h', 'e', 'y'].forEach((letter) => console.log(letter))
```

При попытке выполнения этого кода будет выдана ошибка `Uncaught TypeError: Cannot read property 'forEach' of undefined` система, основываясь на правиле №1, пытается интерпретировать код следующим образом.

```
const hey = 'hey';

const you = 'hey';

const heyYou = hey + ' ' + you['h', 'e', 'y'].forEach((letter) =>
console.log(letter))
```

Проблему можно решить, самостоятельно поставив точку с запятой после предпоследней строки первого примера.

Вот ещё один фрагмент кода.

```
(1 + 2).toString()
```

Результатом его выполнения станет вывод строки `"3"`. А что произойдёт, если нечто подобное появится в следующем фрагменте кода?

```
const a = 1

const b = 2

const c = a + b

(a + b).toString()
```


В данной ситуации появится ошибка `TypeError: b is not a function` так как вышеприведённый код будет интерпретирован следующим образом.

```
const a = 1

const b = 2

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

Взглянем теперь на пример, основанный на правиле №4.

```
((() => {

    return

    {

        color: 'white'

    }

}))()
```

Можно подумать, что это IIFE вернёт объект, содержащий свойство `color`, но на самом деле это не так. Вместо этого функция вернёт значение `undefined` так как система добавляет точку с запятой после команды `return`.

Для того чтобы решить подобную проблему, открывающую фигурную скобку объектного литерала нужно поместить в той же строке, где находится команда `return`.

```
((() => {

    return {

        color: 'white'

    }

}))()
```

Если взглянуть на следующий фрагмент кода, то можно подумать, что он выведет в окне сообщения 0.

```
1 + 1

-1 + 1 === 0 ? alert(0) : alert(2)
```

Но он выводит 2, так как, в соответствии с правилом №1, этот код представляется следующим образом.

```
1 + 1 -1 + 1 === 0 ? alert(0) : alert(2)
```

В вопросе использования точек с запятой в JavaScript стоит проявлять осторожность. Вы можете встретить как горячих сторонников точек с запятой, так и их противников. На самом деле, решая, нужны ли в вашем коде точки с запятой, можно положиться на тот факт, что JS поддерживает их автоматическую подстановку, но при этом каждый должен сам для себя решить — нужны ли они в его коде или нет. Главное — последовательно и разумно применять выбранный подход. В том, что касается расстановки точек с запятой и структуры кода, можно порекомендовать придерживаться следующих правил:

- Пользуясь командой `return`, располагайте то, что она должна вернуть из функции, в той же строке, в которой находится эта команда. То же самое касается команд `break`, `throw`, `continue`.

- Уделяйте особое внимание ситуациям, когда новая строка кода начинается со скобки, так как эта строка может быть автоматически объединена с предыдущей и представлена системой как попытка вызова функции или попытка доступа к элементу массива.

В целом же можно сказать, что, ставите ли вы точки с запятой самостоятельно, или полагаетесь на их автоматическую расстановку, тестируйте код для того, чтобы убедиться, что работает он именно так, как ожидается.

Кавычки и шаблонные литералы

Поговорим об особенностях использования кавычек в JavaScript. А именно, речь идёт о следующих допустимых в JS-программах типах кавычек:

- Одинарные кавычки.
- Двойные кавычки.
- Обратные кавычки.

Одинарные и двойные кавычки, в целом, можно считать одинаковыми.

```
const test = 'test'
```

```
const bike = "bike"
```

Разницы между ними практически нет. Пожалуй, единственное заметное различие заключается в том, что в строках, заключённых в одинарные кавычки, нужно экранировать символ одинарной кавычки, а в строках, заключённых в двойные — символ двойной.

```
const test = 'test'
```

```
const test = 'te\'st'
```

```
const test = 'te"st'
```

```
const test = "te\'st"
```

```
const test = "te'st"
```

В разных руководствах по стилю можно найти как рекомендацию по использованию одинарных кавычек, так и рекомендацию по использованию двойных кавычек. Автор этого материала говорит, что в JS-коде стремится использовать исключительно одинарные кавычки, используя двойные только в HTML-коде.

Обратные кавычки появились в JavaScript с выходом стандарта ES6 в 2015 году. Они, помимо других новых возможностей, позволяют удобно описывать многострочные строки. Такие строки можно задавать и используя обычные кавычки — с применением escape-последовательности `\n`. Выглядит это так.

```
const multilineString = 'A string\nnon multiple lines'
```

Обратные кавычки (обычно кнопка для их ввода находится левее цифровой клавиши 1 на клавиатуре) позволяют обойтись без `\n`.

```
const multilineString = `A string
```

```
on multiple lines`
```

Но этим возможности обратных кавычек не ограничены. Так, если строка описана с использованием обратных кавычек, в неё, используя конструкцию `${}`, можно подставлять значения, являющиеся результатом вычисления JS-выражений.

```
const multilineString = `A string
```

```
on ${1+1} lines`
```

Такие строки называют шаблонными литералами.

Шаблонные литералы отличаются следующими особенностями:

- Они поддерживают многострочный текст.
- Они дают возможность интерполировать строки, в них можно использовать встроенные выражения.
- Они позволяют работать с тегированными шаблонами, давая возможность создавать собственные предметно-ориентированные языки (DSL, Domain-Specific Language).

Поговорим об этих возможностях.

Многострочный текст

Задавая, с помощью обратных кавычек, многострочные тексты, нужно помнить о том, что пробелы в таких текстах так же важны, как и другие символы. Например, рассмотрим следующий многострочный текст.

```
const string = `First
                  Second`
```

Его вывод даст примерно следующее.

```
First
    Second
```

То есть оказывается, что когда этот текст вводился в редакторе, то, возможно, программист ожидал, что слова `First` и `Second`, при выводе, окажутся строго друг под другом, но на самом деле это не так. Для того чтобы обойти эту проблему, можно начинать многострочный текст с перевода строки, и, сразу после закрывающей обратной кавычки, вызывать метод `trim()`, который удалит пробельные символы, находящиеся в начале или в конце строки. К таким символам, в частности, относятся пробелы и знаки табуляции. Удалены будут и символы конца строки. Выглядит это так.

```
const string = `
First
Second`.trim()
```

Интерполяция

Под интерполяцией здесь понимается преобразование переменных и выражений в строки. Делается это с использованием конструкции `${}`.

```
const variable = 'test'

const string = `something ${ variable }` //something test
```

В блок `${}` можно добавлять всё что угодно — даже выражения.

```
const string = `something ${1 + 2 + 3}`

const string2 = `something ${foo() ? 'x' : 'y' }`
```

В константу `string` попадёт текст `something 6`, в константу `string2` будет записан либо текст `something x`, либо текст `something y`. Это зависит от того, истинное или ложное значение вернёт функция `foo()` (здесь применяется тернарный оператор, который, если то, что находится до знака вопроса, является истинным, возвращает то, что идёт после знака вопроса, в противном случае возвращая то, что идёт после двоеточия).

Тегированные шаблоны

Тегированные шаблоны применяются во множестве популярных библиотек. Среди них — [Styled Components](#), [Apollo](#), [GraphQL](#).

То, что выводят такие шаблоны, подчиняется некоей логике, задаваемой с помощью функции. Вот немного переработанный пример, приведённый в одной из наших [публикаций](#), иллюстрирующий работу с тегированными шаблонными строками.

```
const esth = 8

function helper(strs, ...keys) {
  const str1 = strs[0] //ES
  const str2 = strs[1] //is
  let additionalPart = ''
  if (keys[0] == 8) { //8
    additionalPart = 'awesome'
  }
  else {
    additionalPart = 'good'
  }

  return `${str1}${keys[0]}${str2}${additionalPart}`
}

const es = helper`ES ${esth} is `
console.log(es) //ES 8 is awesome.
```

Здесь, если в константе `esth` записано число 8, в `es` попадёт строка `ES 8 is awesome`. В противном случае там окажется другая строка. Например, если в `esth` будет число 6, то она будет выглядеть как `ES 6 is good`.

В [Styled Components](#) тегированные шаблоны используются для определения CSS-строк.

```
const Button = styled.button`
  font-size: 1.5em;
  background-color: black;
  color: white;
`;
```

В [Apollo](#) они применяются для определения GraphQL-запросов.

```
const query = gql`
  query {
    ...
  }
`
```

```
}
```

Зная то, как устроены тегированные шаблоны, несложно понять, что `styled.button` и `gql` из предыдущих примеров — это просто функции.

```
function gql(literals, ...expressions) {  
  
}
```

Например, функция `gql()` возвращает строку, которая может быть результатом любых вычислений. Параметр `literals` этой функции представляет собой массив, содержащий разбитое на части содержимое шаблонного литерала, `expressions` содержит результаты вычисления выражений.

Разберём следующую строку.

```
const string = helper`something ${1 + 2 + 3} `
```

В функцию `helper` попадёт массив `literals`, содержащий два элемента. В первом будет текст `something` с пробелом после него, во втором — пустая строка — то есть то, что находится между выражением `${1 + 2 + 3}` и концом строки. В массиве `expressions` будет один элемент — `6`.

Вот более сложный пример.

```
const string = helper`something  
  
another ${'x'}  
  
new line ${1 + 2 + 3}  
  
test`
```

Здесь в функцию `helper`, в качестве первого параметра попадёт следующий массив.

```
[ 'something\nanother ', '\nnew line ', '\ntest' ]
```

Второй массив будет выглядеть так.

```
[ 'x', 6 ]
```

Часть 7: строгий режим, ключевое слово `this`, события, модули, математические вычисления

Строгий режим

Строгий режим (`strict mode`) появился в стандарте ES5. В этом режиме меняется семантика языка, он нацелен на то, чтобы улучшить поведение JavaScript, что приводит к тому, что код в этом режиме ведёт себя не так, как обычный. Фактически, речь идёт о том, что в этом режиме устраняются недостатки, неоднозначности языка, устаревшие возможности, которые сохраняются в нём из соображений совместимости.

Включение строгого режима

Для того чтобы использовать в некоем коде строгий режим, его нужно явным образом включить. То есть, речь не идёт о том, что этот режим применяется по умолчанию. Такой подход нарушил бы работу бесчисленного количества существующих программ, опирающихся на механизмы языка, присутствовавшие в нём с самого начала, с 1996 года. На самом деле, значительные усилия тех, кто разрабатывает стандарты JavaScript, направлены именно на обеспечение совместимости, на то, чтобы код, написанный в расчёте на старые версии

стандартов, можно было бы выполнять на сегодняшних JS-движках. Такой подход можно считать одним из залогов успеха JavaScript как языка для веб-разработки.

Для того чтобы включить строгий режим, используется особая директива, которая выглядит так.

```
'use strict'
```

К тому же эффекту приведёт и директива, записанная в виде `"use strict"`, и та же директива, после которой поставлена точка с запятой (`'use strict';` и `"use strict";`)

Эту директиву (именно так — вместе с кавычками), для того, чтобы весь код в некоем файле выполнялся бы в строгом режиме, помещают в начале этого файла.

```
'use strict'
```

```
const name = 'Flavio'
```

```
const hello = () => 'hey'
```

```
//...
```

Строгий режим может быть включён и на уровне отдельной функции. Для этого соответствующую директиву надо поместить в начале кода тела функции.

```
function hello() {  
    'use strict'  
    return 'hey'  
}
```

Подобное может оказаться полезным в том случае, если строгий режим нужно использовать в существующей кодовой базе и при этом включение его на уровне файла оказывается нецелесообразным по причине нехватки времени на тщательное тестирование кода всего этого файла.

Надо отметить, что, если строгий режим включён, выключить его во время выполнения программы нельзя.

Рассмотрим некоторые особенности строгого режима.

Борьба со случайной инициализацией глобальных переменных

Мы уже говорили о том, что если случайно назначить некое значение необъявленной переменной, даже если сделать это в коде функции, такая переменная по умолчанию будет сделана глобальной (принадлежащей глобальному объекту). Это может привести к неожиданностям.

Например, следующий код приводит к созданию именно такой переменной.

```
;(function() {  
    variable = 'hey'  
})()
```

Переменная `variable` будет доступна в глобальной области видимости после выполнения IIFE.

Если включить на уровне этой функции строгий режим, тот же самый код вызовет ошибку.

```
;(function() {  
    'use strict'
```

```
variable = 'hey'

}) ()
```

Ошибки, возникающие при выполнении операций присваивания значений

JavaScript, в обычном режиме, никак не сообщает о некоторых ошибках, возникающих в ходе выполнения операций присваивания значений.

Например, в JS имеется значение [undefined](#), которое представляет собой одно из примитивных значений языка и представлено свойством глобального объекта `undefined`. В обычном JS вполне возможна такая команда.

```
undefined = 1
```

Выглядит это как запись единицы в некую переменную с именем `undefined`, а на самом деле это — попытка записи нового значения в свойство глобального объекта, которое, кстати, в соответствии со стандартом, нельзя перезаписывать. В обычном режиме, хотя такая команда и возможна, она ни к чему не приведёт — то есть, и значение `undefined` изменено не будет, и сообщение об ошибке не появится. В строгом режиме подобное вызовет ошибку. Для того чтобы увидеть это сообщение об ошибке, а заодно убедиться в том, что значение `undefined` не переопределяется в обычном режиме, попробуйте выполнить следующий код в браузере или в Node.js.

```
undefined = 1

console.log('This is ' + undefined)

;(() => {

    'use strict'

    undefined = 1

}) ()
```

Такое же поведение системы характерно и при работе с такими сущностями, как значения `Infinity` и `NaN`, а также с другими подобными. Строгий режим позволяет всего этого избежать.

В JavaScript можно задавать свойства объектов с использованием метода [Object.defineProperty\(\)](#). В частности, с помощью этого метода можно задавать свойства, которые нельзя менять.

```
const car = {}

Object.defineProperty(car, 'color', {

    value: 'blue',

    writable: false

})

console.log(car.color)

car.color = 'yellow'

console.log(car.color)
```

Обратите внимание на атрибут `writable: false`, использованный при настройке свойства `color`.

Вышеприведённый код, выполненный в обычном режиме, не приведёт ни к изменению свойства объекта `color`, ни к выводу ошибки. Попытка поменять это свойство в строгом режиме окончится выдачей ошибки.

```
;(() => {
    'use strict'

    car.color = 'red'
})()
```

То же самое относится и к геттерам. Этот код выполнится, хотя и безрезультатно.

```
const car = {
    get color() {
        return 'blue'
    }
}

console.log(car.color)

car.color = 'red'

console.log(car.color)
```

А попытка выполнить то же самое в строгом режиме вызовет ошибку, сообщающая о попытке установки свойства объекта, у которого есть лишь геттер.

```
;(() => {
    'use strict'

    car.color = 'yellow'
})()
```

В JavaScript есть метод [Object.preventExtensions\(\)](#), делающий объект нерасширяемым, то есть таким, к которому нельзя добавить новые свойства. При работе с такими объектами в обычном режиме проявляются те же особенности языка, которые мы рассматривали выше.

```
const car = { color: 'blue' }

Object.preventExtensions(car)

console.log(car.model)

car.model = 'Fiesta'

console.log(car.model)
```

Здесь обе попытки вывести свойство объекта `model` приведут к появлению в консоли значения `undefined`. Такого свойства в объекте не было, попытка создать его после того, как объект был сделан нерасширяемым, ни к чему не привела. То же действие в строгом режиме приводит к выдаче сообщения об ошибке.

```
;(() => {
    'use strict'

    car.owner = 'Flavio'
```



```
}  
  
) ()
```

В эту же категорию действий, не приводящих к каким-то изменениям, возможно, ожидаемым программистом, но и не вызывающих ошибок, попадают операции, в ходе выполнения которых делаются попытки назначить некие свойства примитивным значениям. Например, такой код, в обычном режиме, не вызовет ошибки, но и не даст никаких результатов.

```
let one = 1  
  
one.prop = 2  
  
console.log(one.prop)
```

То же самое в строгом режиме приведёт к появлению сообщения об ошибке, указывающем на то, что у числа 1 нельзя создать свойство `prop`. Похожим образом система ведёт себя и при работе с другими примитивными типами данных.

Ошибки, связанные с удалением сущностей

В обычном режиме, если попытаться удалить, помощью оператора [delete](#), свойство объекта, которое удалить нельзя, `delete` просто возвратит `false` и всё тихо закончится неудачей.

```
delete Object.prototype
```

В строгом режиме здесь будет выдана ошибка.

Аргументы функций с одинаковыми именами

Функции могут иметь параметры с одинаковыми именами, ошибок это не вызывает (хотя подобное выглядит как ошибка того, кто такую функцию создал).

```
;(function(a, a, b) {  
  
    console.log(a, b)  
  
})(1, 2, 3) //2 3
```

Этот код в обычном режиме выводит в консоль 2 3. В строгом режиме подобное вызовет ошибку.

Кстати, если при объявлении стрелочной функции её параметры будут иметь одинаковые имена, это, и в обычном режиме, приведёт к выводу сообщения об ошибке.

Восьмеричные значения

В обычном JavaScript можно пользоваться восьмеричными значениями, добавляя в начало числа 0.

```
;( () => {  
  
    console.log(010)  
  
}) () //8
```

Здесь в консоль попадёт десятичное представление восьмеричного числа 10, то есть 8. Этот 0 перед числом может быть поставлен случайно. В строгом режиме работать с восьмеричными числами, заданными в таком формате, нельзя. Но если нужно и пользоваться строгим режимом и работать с восьмеричными числами, записывать их можно в формате 0oXX. Следующий код тоже выведет 8.

```
;( () => {  
  
    'use strict'
```

```
console.log(0o10)

}) () //8
```

Оператор with

Оператор [with](#), использование которого может привести к путанице, в строгом режиме запрещён.

Изменение поведения кода в строгом режиме не ограничиваются теми, которые мы обсудили выше. В частности, в этом режиме иначе ведёт себя ключевое слово `this`, с которым мы уже сталкивались, и о котором сейчас мы поговорим подробнее.

Особенности ключевого слова this

Ключевое слово `this`, или контекст выполнения, позволяет описать окружение, в котором производится выполнение JS-кода. Его значение зависит от места его использования и от того, включён или нет строгий режим.

Ключевое слово this в строгом режиме

В строгом режиме значение `this`, передаваемое в функции, не приводится к объекту. Это преобразование не только требует ресурсов, но и даёт функциям доступ к глобальному объекту в том случае, если они вызываются с `this`, установленным в `undefined` или `null`. Такое поведение означает, что функция может получить несанкционированный доступ к глобальному объекту. В строгом режиме преобразования `this`, передаваемого функции, не производится. Для того чтобы увидеть разницу между поведением `this` в функциях в разных режимах — попробуйте этот код с использованием директивы `'use strict'` и без неё.

```
;(function() {

    console.log(this)

}) ()
```

Ключевое слово this в методах объектов

Метод — это функция, ссылка на которую записана в свойство объекта. Ключевое слово `this` в такой функции ссылается на этот объект. Это утверждение можно проиллюстрировать следующим примером.

```
const car = {

    maker: 'Ford',

    model: 'Fiesta',

    drive() {

        console.log(`Driving a ${this.maker} ${this.model} car!`)

    }

}

car.drive()

//Driving a Ford Fiesta car!
```

В данном случае мы применяем обычную функцию (а не стрелочную — это важно), ключевое слово `this`, используемое в которой, автоматически привязывается к содержащему эту функцию объекту.

Обратите внимание на то, что вышеприведённый способ объявления метода объекта аналогичен такому:

```
const car = {

    maker: 'Ford',
```

```
model: 'Fiesta',

drive: function() {

    console.log(`Driving a ${this.make} ${this.model} car!`)

}

}
```

То же самое поведение ключевого слова `this` в методе объекта можно наблюдать и при использовании следующей конструкции.

```
const car = {

    make: 'Ford',

    model: 'Fiesta'

}

car.drive = function() {

    console.log(`Driving a ${this.make} ${this.model} car!`)

}

car.drive()

//Driving a Ford Fiesta car!
```

Ключевое слово `this` и стрелочные функции

Попробуем переписать вышеприведённый пример с использованием, в качестве метода объекта, стрелочной функции.

```
const car = {

    make: 'Ford',

    model: 'Fiesta',

    drive: () => {

        console.log(`Driving a ${this.make} ${this.model} car!`)

    }

}

car.drive()

//Driving a undefined undefined car!
```

Как видно, тут, вместо названий производителя автомобиля и его модели выводятся значения `undefined`. Дело в том, что, как мы уже говорили, `this` в стрелочной функции содержит ссылку на контекст, включающий в себя функцию.

К стрелочной функции нельзя привязать `this`, а к обычной функции можно

Привязка this

В JavaScript существует такое понятие, как привязка `this`. Сделать это можно разными способами. Например, при объявлении функции привязать её ключевое слово `this` можно к некоему объекту с использованием метода `bind()`.

```
const car = {  
  maker: 'Ford',  
  model: 'Fiesta'  
}  
  
const drive = function() {  
  console.log(`Driving a ${this.maker} ${this.model} car!`)  
}.bind(car)  
  
drive()  
  
//Driving a Ford Fiesta car!
```

С помощью этого же метода к методу одного объекта, в качестве `this`, можно привязать другой объект.

```
const car = {  
  maker: 'Ford',  
  model: 'Fiesta',  
  drive() {  
    console.log(`Driving a ${this.maker} ${this.model} car!`)  
  }  
}  
  
const anotherCar = {  
  maker: 'Audi',  
  model: 'A4'  
}  
  
car.drive.bind(anotherCar)()  
  
//Driving a Audi A4 car!
```

Привязку `this` можно организовать и на этапе вызова функции, используя методы `call()` и `apply()`.

```
const car = {  
  maker: 'Ford',  
  model: 'Fiesta'  
}  
  
const drive = function(kmh) {
```

```

    console.log(`Driving a ${this.make} ${this.model} car at ${kmh} km/h!`)
  }

  drive.call(car, 100)

  //Driving a Ford Fiesta car at 100 km/h!

  drive.apply(car, [100])

  //Driving a Ford Fiesta car at 100 km/h!

```

К `this` привязывается то, что передаётся этим методам в качестве первого аргумента. Разница между этими методами заключается в том, что `apply()`, в качестве второго аргумента, принимает массив с аргументами, передаваемыми функции, а `call()` принимает список аргументов.

О привязке `this` в обработчиках событий браузера

В коллбэках обработчиков событий `this` указывает на HTML-элемент, с которым произошло то или иное событие. Для того чтобы привязать к коллбэку, в качестве `this`, что-то другое, можно воспользоваться методом `bind()`. Вот пример, иллюстрирующий это.

```

<!DOCTYPE html>

<html>

  <body>

    <button id="el">Element (this)</button>

    <button id="win">Window (this)</button>

    <script>

      const el = document.getElementById("el")

      el.addEventListener('click', function () {

        alert(this) //object HTMLButtonElement

      })

      const win = document.getElementById("win")

      win.addEventListener('click', function () {

        alert(this) //object Window

      }).bind(this))

    </script>

  </body>

</html>

```

События

JavaScript в браузере использует событийную модель программирования. Те или иные действия выполняются кодом в ответ на происходящие события. В этом разделе мы поговорим о событиях и о том, как их обрабатывать.

В качестве события может выступать, например, завершение загрузки DOM, получение данных, выполненное в результате асинхронного запроса, щелчок мышью по элементу страницы, прокрутка страницы, ввод некоего символа с клавиатуры. На самом деле, существует множество событий, обрабатывая которые, JS-код страницы позволяет решать широкий спектр задач по взаимодействию приложения с пользователями, с элементами страницы, с окружением, в котором работает код.

Обработчики событий

Реагировать на события можно с помощью обработчиков событий (event handler), которые представляют собой функции, вызываемые в тот момент, когда происходят события.

При необходимости для обработки одного и того же события можно зарегистрировать несколько обработчиков, которые будут вызываться в том случае, если это событие произойдет. Регистрировать обработчики событий можно различными способами. Рассмотрим три таких способа.

Встроенные обработчики событий

В наши дни встроенные обработчики событий используются редко из-за их ограниченности. Раньше они применялись гораздо чаще. Для задания такого обработчика события его код добавляется в HTML-разметку элемента в виде особого атрибута. В следующем примере такой вот простейший обработчик события `onclick`, возникающего при щелчке по кнопке, назначен кнопке с надписью `Button 1`.

```
<!DOCTYPE html>

<html>

  <body>

    <button onclick="alert('Button 1!')">Button 1</button>

    <button onclick="doSomething()">Button 2</button>

    <script>

      function doSomething() {

        const str = 'Button 2!'

        console.log(str)

        alert(str)

      }

    </script>

  </body>

</html>
```

В HTML-коде кнопки `Button` 2 применяется похожий подход, но здесь указывается функция, код которой выполняется в ответ на нажатие кнопки. Этот код выполняет вывод заданной строки в консоль и выводит окно с тем же текстом.

Назначение обработчика свойству HTML-элемента

Этот метод назначения обработчиков событий подходит для случаев, когда у некоего события элемента должен быть лишь один обработчик. Заключается он в назначении функции соответствующему свойству элемента.

Например, у объекта `window` есть событие `onload`, которое вызывается после загрузки HTML-кода страницы и всех дополнительных ресурсов, необходимых ей, например — стилей и изображений. Если назначить этому событию обработчик, то при его вызове можно быть уверенным в том, что браузер загрузил всё содержимое страницы, с которым теперь можно работать программно, не опасаясь того, что какие-то элементы страницы ещё не загружены.

```
window.onload = () => {  
    alert('Hi!') //страница полностью загружена  
}
```

Такой подход часто используют при обработке XHR-запросов. Так, в ходе настройки запроса можно задать обработчик его события [onreadystatechange](#), который будет вызван при изменении состояния его свойства `readyState`. Вот пример использования этого подхода для загрузки JSON-данных из общедоступного API.

```
<!DOCTYPE html>  
  
<html>  
  
    <body>  
  
  
  
  
  
  
  
  
        <button onclick="loadData()">Start</button>  
  
  
  
  
  
  
  
  
        <script>  
  
  
  
  
  
  
  
  
            function loadData () {  
  
                const xhr = new XMLHttpRequest()  
  
                const method = 'GET'  
  
                const url = 'https://jsonplaceholder.typicode.com/todos/1'  
  
                xhr.open(method, url, true)  
  
                xhr.onreadystatechange = function () {  
  
                    if(xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200)  
  
                    {  
  
                        console.log(xhr.responseText)  
  
                    }  
  
                }  
  
            }  
  
        }  
  
    }  
  
}
```

```
        xhr.send()

    }

</script>

</body>

</html>
```

Проверить, назначен ли обработчик некоему событию, можно так.

```
if (window.onload){}
```

Использование метода `addEventListener()`

Метод `addEventListener()`, который нам уже встречался, представляет собой современный механизм назначения обработчиков событий. Он позволяет регистрировать несколько обработчиков для одного события.

```
window.addEventListener('load', () => {

    //загрузка завершена

})
```

Обратите внимание на то, что браузер IE8 (и его более старые версии) не поддерживает метод `addEventListener()`. Тут используется похожий метод `attachEvent()`. Это нужно учитывать в том случае, если ваша программа должна поддерживать устаревшие браузеры.

О назначении обработчиков событий различным элементам

Подключать обработчики событий к объекту `window` можно для обработки «глобальных» событий, таких, как нажатия кнопок на клавиатуре. В то же время, отдельным HTML-элементам назначают обработчики событий, которые реагируют на то, что происходит с этими элементами, например — на щелчки по ним мышью. Поэтому метод `addEventListener()` используют и с объектом `window`, и с обычными элементами.

Объект `Event`

В качестве первого параметра обработчик события может принимать объект события — `Event`. Набор свойств этого объекта зависит от события, которое он описывает. Вот, например, код, который демонстрирует обработку событий нажатия клавиш клавиатуры с использованием события `keydown` объекта `window`.

```
<!DOCTYPE html>

<html>

    <body>

        <script>

            window.addEventListener('keydown', event => {

                //нажата клавиша на клавиатуре

                console.log(event.type, event.key)

            })

            window.addEventListener('mousedown', event => {

                //нажата кнопка мыши
```



```

        //0 - левая кнопка, 2 - правая

        console.log(event.type, event.button, event.clientX, event.clientY)

    })

</script>

</body>

</html>

```

Как видно, здесь, для вывода в консоль сведений о нажатой клавише, используется свойство объекта `key`. Здесь же используется и свойство `type`, указывающее на тип события. В этом примере, на самом деле, мы работаем с объектом [KeyboardEvent](#), который используется для описаний событий, связанных с клавиатурой. Этот объект является наследником объекта [Event](#). Объекты, предназначенные для обработки разнообразных событий, расширяют возможности стандартного объекта события.

В этом же примере, для обработки событий, связанных с мышью, используется объект [MouseEvent](#). В обработчике события `mousedown` мы выводим в консоль тип события, номер кнопки (свойство `button`) и координаты указателя в момент щелчка (свойства `clientX` и `clientY`).

Объект [DragEvent](#) применяется при обработке событий, возникающих при перетаскивании элементов страницы.

Среди свойств объекта `Event`, доступных и в других объектах событий, можно отметить уже упомянутое свойство `type` и свойство `target`, указывающее на DOM-элемент, на котором произошло событие. У объекта `Event` есть и методы. Например — метод `createEvent()` позволяет создавать новые события.

Всплытие событий

Рассмотрим следующий пример.

```

<!DOCTYPE html>

<html>

    <head>

        <style>

            #container {

                height: 100px;

                width: 200px;

                background-color: blue;

            }

            #child {

                height: 50px;

                width: 100px;

                background-color: green;

            }

```

```
</style>

</head>

<body>

<div id="container">

  <div id="child">

    </div>

  </div>

<script>

  const contDiv = document.getElementById('container')

  contDiv.addEventListener('click', event => {

    console.log('container')

  })

  const chDiv = document.getElementById('child')

  chDiv.addEventListener('click', event => {

    console.log('child')

  })

  window.addEventListener('click', event => {

    console.log('window')

  })

</script>

</body>

</html>
```

Если открыть загрузить страницу с таким кодом в браузер, открыть консоль и последовательно щёлкнуть мышью сначала в свободной области страницы, потом — по синему прямоугольнику, а потом — по зелёному, то в консоль попадёт следующее:

```
window
```

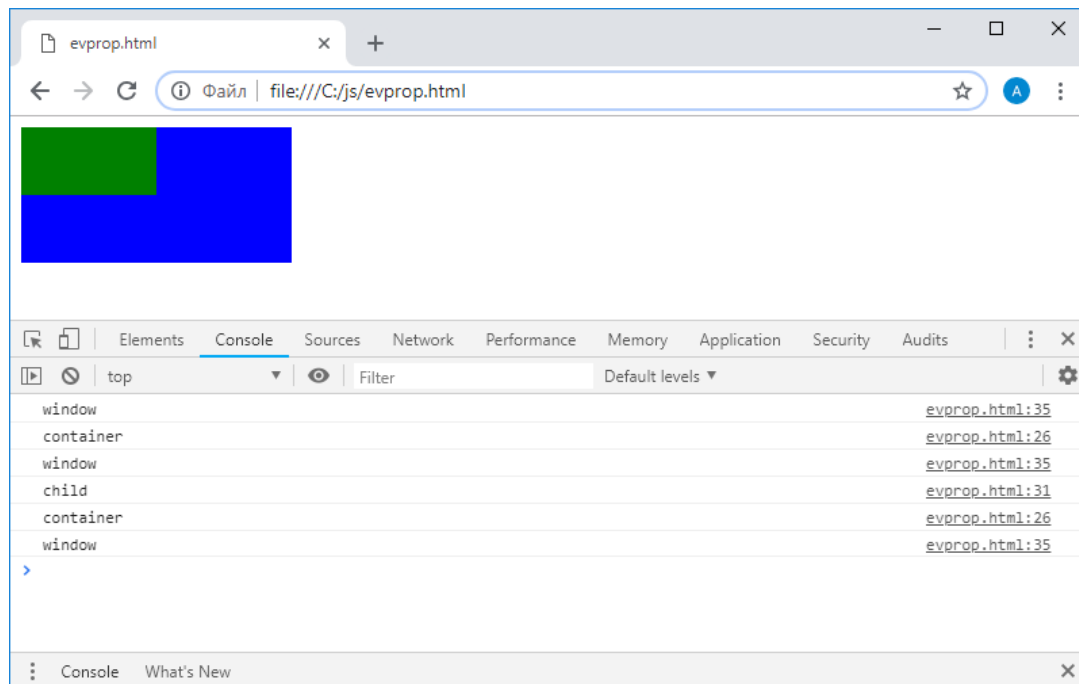
```
container
```

```
window
```

```
child
```

```
container
```

window



Всплытие событий

То, что здесь можно наблюдать, называется всплытием события (event bubbling). А именно, событие, возникающее у дочернего элемента, распространяется на родительский элемент. Этот процесс продолжается до тех пор, пока событие не достигнет самого «верхнего» элемента. Если у элементов, по которым проходит всплывающее событие, определены соответствующие обработчики, они будут вызваны в соответствии с порядком распространения события.

Всплытие событий можно останавливать, пользуясь методом `stopPropagation()` объекта события. Например, если нужно, чтобы, после щелчка мышью по элементу `child`, соответствующее событие не распространялось бы дальше, нам нужно переписать код, в котором мы назначаем ему обработчик события `click`, следующим образом.

```
chDiv.addEventListener('click', event => {  
    console.log('child')  
    event.stopPropagation()  
})
```

Если теперь выполнить ту же последовательность действий, которую мы выполняли выше, то есть — щёлкнуть в свободной области окна, потом — по элементу `container`, а потом — по `child`, то в консоль будет выведено следующее.

```
window  
container  
window  
child
```

Часто используемые события

Рассмотрим некоторые события, обработка которых нужна чаще всего.

Событие `load`

Событие `load` вызывается у объекта `window` при завершении загрузки страницы. Подобные события есть и у других элементов, например, у HTML-элемента `body`.

События мыши

Событие `click` вызывается по щелчку кнопкой мыши. Событие `dblclick` — по двойному щелчку. При этом если заданы обработчики событий `click` и `dblclick`, сначала вызывается обработчик события `click`, а потом — `dblclick`. События `mousedown`, `mousemove`, `mouseup`, можно использовать для обработки перетаскивания объектов по странице. При этом надо отметить, что если элементу назначен обработчик события `mousemove`, этот обработчик будет вызываться, в ходе перемещения указателя мыши над элементом, очень много раз. В подобных ситуациях, если в соответствующем обработчике выполняются какие-то достаточно тяжёлые вычисления, есть смысл ограничить частоту выполнения этих вычислений. Мы поговорим об этом ниже.

События клавиатуры

Событие `keydown` вызывается при нажатии на клавишу клавиатуры. Если клавишу удерживают нажатой, оно продолжает вызываться. Когда клавишу отпускают — вызывается событие `keyup`.

Событие `scroll`

Событие `scroll` вызывается для объекта `window` при прокрутке страницы. В обработчике события можно, для того, чтобы узнать позицию прокрутки, обратиться к свойству `window.scrollY`.

Это событие, так же, как и событие `mousemove`, вызывается в ходе выполнения соответствующей операции много раз.

Ограничение частоты выполнения вычислений в обработчиках событий

События `mousemove` и `scroll` дают сведения о координатах мыши и о позиции прокрутки. Выполнение в обработчиках таких событий каких-то серьёзных вычислений может привести к замедлению работы программы. В подобной ситуации есть смысл задуматься об ограничении частоты выполнения таких вычислений. Этот приём называют «троттлингом» (throttling), его реализации можно найти в специализированных библиотеках вроде [Lodash](#). Сущность этого приёма заключается в создании механизма, который позволяет ограничить частоту выполнения неких действий, которые, без ограничения, выполнялись бы слишком часто. Рассмотрим собственную реализацию этого механизма.

```
let cached = null

window.addEventListener('mousemove', event => {

  if (!cached) {

    setTimeout(() => {

      //предыдущее событие доступно через переменную cached

      console.log(cached.clientX, cached.clientY)

      cached = null

    }, 100)

  }

  cached = event

})
```

Теперь вычисления, выполняемые при возникновении события `mousemove`, производятся не чаще одного раза за 100 миллисекунд.

ES-модули

В стандарте ES6 появилась новая возможность, получившая название ES-модули. Потребность в стандартизации этой возможности назрела уже давно, что выражается в том, что и разработчики клиентских частей веб-проектов, и серверные программисты, пишущие для среды Node.js, уже давно нечто подобное используют.

Модуль представляет собой файл, содержащий некий код. Из этого файла можно экспортировать, делать общедоступными, функции и переменные. Ими можно воспользоваться, просто подключив модуль к некоему файлу с кодом, при этом внутренние механизмы модуля извне недоступны.

В Node.js в качестве системы модулей долгое время использовался и продолжает использоваться стандарт CommonJS. В браузерах, до появления ES-модулей, применялись различные библиотеки и системы сборки проектов, имитирующие возможность работы с модулями. Теперь же, после стандартизации, браузеры постепенно вводят поддержку ES-модулей, что позволяет говорить о том, что, для поддержки модулей, уже довольно скоро дополнительных средств не понадобится. В частности, по информации ресурса caniuse.com, в конце ноября 2018 года уровень поддержки ES-модулей браузерами немного превышает 80%.

Работа по внедрению ES-модулей [ведётся](#) и в Node.js.

Синтаксис ES-модулей

В Node.js для подключения ES-модулей применяется такая запись.

```
import package from 'module-name'
```

При работе с CommonJS-модулями то же самое выглядит так.

```
const package = require('module-name')
```

Как уже было сказано, модуль представляет собой JavaScript-файл, который что-то экспортирует. Делается это с помощью ключевого слова `export`. Например, напишем модуль, который экспортирует функцию, преобразующую переданную ей строку к верхнему регистру, и дадим файлу с ним имя `uppercase.js`. Пусть его текст будет таким.

```
export default str => str.toUpperCase()
```

Здесь в модуле задана команда экспорта по умолчанию, поэтому экспортироваться может анонимная функция. В противном случае экспортируемым сущностям надо давать имена.

Теперь этот модуль можно импортировать в некий код (в другой модуль, например) и воспользоваться там его возможностями.

Загрузить модуль на HTML-страницу можно, используя тег `<script>` с атрибутом `type="module"`.

```
<script type="module" src="index.js"></script>
```

Обратите внимание на то, что такой способ импорта модулей работает как [отложенная](#) (defer) загрузка скрипта. Кроме того, важно учитывать то, что в нашем примере в модуле `uppercase.js` используется экспорт по умолчанию, поэтому, при его импорте, ему можно назначить любое желаемое имя. Вот как это выглядит в коде веб-страницы. Для того чтобы у вас этот пример заработал, вам понадобится локальный веб-сервер. Например, если вы пользуетесь редактором VSCode, можно воспользоваться его расширением Live Server (идентификатор — `ritwickdey.liveserver`).

```
<!DOCTYPE html>
```

```
<html>
```

```
  <head>
```

```
</head>

<body>

  <script type="module">

    import toUpperCase from './uppercase.js'

    console.log(toUpperCase('hello'))

  </script>

</body>

</html>
```

После загрузки этой страницы в консоль попадёт текст HELLO.

Модули можно импортировать и с использованием абсолютного URL.

```
import toUpperCase from 'https://flavio-es-modules-
example.glitch.me/uppercase.js'
```

Пути в командах импорта модулей могут быть либо абсолютными, либо относительными, начинающимися с ./ или ../

Другие возможности импорта и экспорта

Выше мы приводили пример модуля, использующего экспорт по умолчанию.

```
export default str => str.toUpperCase()
```

Однако из модуля можно экспортировать и несколько сущностей.

```
const a = 1

const b = 2

const c = 3

export { a, b, c }
```

Если этот код поместить в файл module.js, то импортировать всё, что он экспортирует, и воспользоваться всем этим можно следующим образом.

```
<html>

  <head>

  </head>

  <body>

    <script type="module">

      import * as m from './module.js'

      console.log(m.a, m.b, m.c)

    </script>

  </body>
```

```
</html>
```

В консоль будет выведено 1 2 3.

Импортировать из модуля можно только то, что нужно, пользуясь конструкциями следующего вида.

```
import { a } from './module.js'
```

```
import { a, b } from './module.js'
```

Пользоваться тем, что импортировано, можно напрямую.

```
console.log(a)
```

Импортируемые сущности можно переименовывать

```
import { a, b as two } from './module.js'
```

Если в модуле используется и экспорт по умолчанию, и другие команды экспорта, то одной командой можно импортировать и то, что экспортировано по умолчанию, и другие сущности. Перепишем наш модуль module.js.

```
const a = 1
```

```
const b = 2
```

```
const c = 3
```

```
export { a, b, c }
```

```
export default () => console.log('hi')
```

Вот как его импортировать и использовать.

```
import sayHi, { a } from './module.js'
```

```
console.log(a)
```

```
sayHi()
```

Пример работы с модулем можно посмотреть [здесь](#).

CORS

При загрузке модулей используется [CORS](#). Это означает, что для успешной загрузки модулей с других доменов у них должен быть задан заголовок CORS, разрешающий межсайтовую загрузку скриптов (наподобие Access-Control-Allow-Origin: *).

Атрибут nomodule

Если браузер не поддерживает работу с модулями, то, работая над страницей, можно предусмотреть резервный механизм в виде загрузки скрипта с использованием тега `<script>`, в котором задан атрибут `nomodule`. Браузер, поддерживающий модули, этот тег проигнорирует.

```
<script type="module" src="module.js"></script>
```

```
<script nomodule src="fallback.js"></script>
```

О модулях ES6 и WebPack

Модули ES6 — это замечательная технология, привлекающая тем, что она входит в стандарт ECMAScript. Но, пользуясь модулями, нужно стремиться к тому, чтобы количество загружаемых файлов было бы небольшим,

иначе это может сказаться на производительности загрузки страницы. В этой связи стоит сказать, что [бандлер](#) WebPack, используемый для упаковки кода приложений, по видимому, ещё долго не потеряет актуальности.

Модули CommonJS

Как уже было сказано, в Node.js используется система модулей CommonJS. Эта система позволяет разработчику создавать небольшие самостоятельные фрагменты кода, подходящие для использования во множестве проектов и поддающиеся автономному тестированию. На основе CommonJS создана огромнейшая экосистема модулей npm.

Давайте напишем CommonJS-модуль, основываясь на примере, который мы уже рассматривали. А именно, поместим в файл `up-node.js` следующий код.

```
exports.uppercase = str => str.toUpperCase()
```

Для того чтобы воспользоваться этим модулем в некоей программе, его нужно подключить.

```
const up = require('./up-node.js')  
  
console.log(up.uppercase('hello'))
```

После выполнения этого кода в консоль попадёт HELLO.

Обычно пакеты, загружаемые из npm, импортируют так, как показано ниже.

```
const package = require('module-name')
```

Модули CommonJS загружаются синхронно и обрабатываются в том порядке, в котором осуществляется обнаружение в коде соответствующих команд. Эта система не используется в клиентском коде.

Из CommonJS-модуля можно экспортировать несколько сущностей.

```
exports.a = 1  
  
exports.b = 2  
  
exports.c = 3
```

Импортировать их можно следующим образом, используя возможности по деструктурирующему присваиванию.

```
const { a, b, c } = require('./up-node.js')
```

Математические вычисления

Математические вычисления часто встречаются в любых программах, и JavaScript — не исключение. Поговорим об арифметических операторах языка и об объекте `Math`, содержащем встроенные реализации некоторых полезных функций. Для того чтобы поэкспериментировать с тем, о чём мы будем тут говорить, удобно пользоваться JS-консолью браузера.

Арифметические операторы

Сложение (+)

Оператор `+` выполняет сложение чисел и конкатенацию строк.

Вот примеры его использования с числами

```
const three = 1 + 2 //3  
  
const four = three + 1 //4
```

Вот как он ведёт себя со строками, преобразуя, при необходимости, другие типы данных к строковому типу.


```
'three' + 1 // three1
```

Вычитание (-)

```
const two = 4 - 2 //2
```

Деление (/)

При работе с обычными числами оператор деления ведёт себя вполне ожидаемым образом.

```
20 / 5 //4
```

```
20 / 7 //2.857142857142857
```

Если поделить число на 0, это не вызовет ошибку. Результатом выполнения такой операции будет значение `Infinity` (для положительного делимого) или `-Infinity` (для отрицательного делимого).

```
1 / 0 //Infinity
```

```
-1 / 0 //-Infinity
```

Остаток от деления (%)

Оператор `%` возвращает остаток от деления, в некоторых ситуациях это может оказаться полезным.

```
20 % 5 //0
```

```
20 % 7 //6
```

Остатком от деления на 0 является особое значение `NaN` (Not a Number — не число).

```
1 % 0 //NaN
```

```
-1 % 0 //NaN
```

Умножение (*)

```
1 * 2 //2
```

```
-1 * 2 //-2
```

Возведение в степень (**)

Этот оператор возводит первый операнд в степень, заданную вторым операндом.

```
1 ** 2 //1
```

```
2 ** 1 //2
```

```
2 ** 2 //4
```

```
2 ** 8 //256
```

```
8 ** 2 //64
```

Унарные операторы

Инкремент (++)

Унарный оператор `++` можно использовать для прибавления 1 к некоему значению. Его можно размещать до инкрементируемого значения или после него.

Если он будет поставлен перед переменной — он сначала увеличивает хранящееся в ней число на 1, после его возвращает данное число.

```
let x = 0
```

```
++x //1
```

```
x //1
```

Если же поставить его после переменной — то он сначала вернёт её предыдущее значение, а потом уже увеличит.

```
let x = 0
```

```
x++ //0
```

```
x //1
```

Декремент (--)

Унарный оператор -- похож на вышерассмотренный ++ с той разницей, что он не увеличивает значения переменных на 1, а уменьшает их.

```
let x = 0
```

```
x-- //0
```

```
x //-1
```

```
--x //-2
```

Унарный оператор (-)

Такой оператор позволяет делать положительные числа отрицательными и наоборот.

```
let x = 2
```

```
-x //-2
```

```
x //2
```

Унарный оператор (+)

Этот оператор, если операнд не является числом, пытается преобразовать его к числу. Если оператор уже является числом — ничего не происходит.

```
let x = 2
```

```
+x //2
```

```
x = '2'
```

```
+x //2
```

```
x = '2a'
```

```
+x //NaN
```

Оператор присваивания и его разновидности

В JavaScript, помимо обычного оператора присваивания (=), есть несколько его разновидностей, которые упрощают выполнение часто встречающихся операций. Вот, например, оператор +=.

```
let x = 2
```

```
x += 3
```

```
x //5
```

Его можно прочитать так: «Прибавить к значению переменной, расположенной слева, то, что находится справа, и записать результат сложения в ту же переменную». Фактически, вышеприведённый пример можно переписать следующим образом.

```
let x = 2
```

```
x = x + 3
```

```
x // 5
```

По такому же принципу работают и другие подобные операторы:

- `--`
- `*=`
- `/=`
- `%=`
- `**=`

Приоритет операторов

При работе со сложными выражениями нужно учитывать приоритет операторов. Например, рассмотрим следующее выражение.

```
const a = 1 * 2 + 5 / 2 % 2
```

Результатом его вычисления будет `2.5`. Для того чтобы понять как получен результат этого выражения, нужно учитывать приоритет операторов. Следующий список операторов начинается с тех, которые имеют наивысший приоритет.

- `-` `+` `++` `--` — унарные операторы, операторы инкремента и декремента.
- `/` `%` — умножение, деление, получение остатка от деления.
- `+` `-` — сложение и вычитание.
- `=` `+=` `-=` `*=` `/=` `%=` `**=` — операторы присваивания.

Операторы, обладающие одинаковым приоритетом, выполняются в порядке их обнаружения в выражении. Если пошагово расписать вычисление вышеприведённого выражения, то получится следующее.

```
const a = 1 * 2 + 5 / 2 % 2
```

```
const a = 2 + 2.5 % 2
```

```
const a = 2 + 0.5
```

```
const a = 2.5
```

Для переопределения порядка выполнения операторов соответствующие части выражения можно включить в круглые скобки. Рассмотрим следующее выражение.

```
const a = 1 * (2 + 5) / 2 % 2
```

Результатом его вычисления будет `1.5`.

Объект Math

Объект `Math` содержит свойства и методы, предназначенные для упрощения математических вычислений. Подробности о нём можно почитать [здесь](#). Этот объект используется самостоятельно, без создания экземпляров.

Среди его свойств можно, например, отметить `Math.E` — константу, содержащую число e , и `Math.PI` — константу, содержащую число π .

```
Math.E // 2.718281828459045
```

```
Math.PI // 3.141592653589793
```

Вот список некоторых полезных методов этого объекта.

- `Math.abs()` — возвращает абсолютное значение числа.
- `Math.ceil()` — округляет число, возвращая наименьшее целое число, большее либо равное указанному.
- `Math.cos()` — возвращает косинус угла, выраженного в радианах.
- `Math.floor()` — округляет число, возвращая наибольшее целое число, меньшее либо равное указанному.
- `Math.max()` — возвращает максимальное из переданных ему чисел.
- `Math.min()` — возвращает минимальное из переданных ему чисел.
- `Math.random()` — возвращает псевдослучайное число из диапазона `[0, 1)` (не включая 1).
- `Math.round()` — округляет число до ближайшего целого числа.
- `Math.sqrt()` — возвращает квадратный корень из числа.

Сравнение значений

Для сравнения значений в JavaScript используются операторы сравнения, с некоторыми из которых мы уже встречались.

- `==` — оператор нестрогого равенства. Перед сравнением значений выполняет преобразование типов.
- `!=` — оператор нестрогого неравенства.

Эти операторы рекомендуется использовать очень осторожно, так как они способны приводить к неожиданным результатам сравнений. Лучше всего, если на то нет особых причин, использовать соответствующие операторы, выполняющие строгую проверку.

- `===` — оператор строгого равенства.
- `!==` — оператор строгого неравенства.

Вот ещё некоторые операторы сравнения.

- `<` — оператор «меньше».
- `<` — оператор «больше».
- `<=` — оператор «меньше или равно».
- `>=` — оператор «больше или равно».

Вот один из примеров, иллюстрирующих особенности работы операторов нестрогого и строгого равенства.

```
1 === true //false
```

```
1 == true //true
```

В первом случае значения сравниваются без приведения типов, в результате оказывается, что число `1` не равно `true`. Во втором случае число `1` приводится к `true` и выражение говорит нам о том, что `1` и `true` равны.

Таймеры и асинхронное программирование

В соответствующих разделах руководства, перевод которого мы публикуем, поднимаются темы использования таймеров и асинхронного программирования. Эти темы были рассмотрены в ранее опубликованном нами переводе курса по Node.js. Для того чтобы с ними ознакомиться, рекомендуем почитать следующие материалы:

- [Руководство по Node.js, часть 6: цикл событий, стек вызовов, таймеры](#)
- [Руководство по Node.js, часть 7: асинхронное программирование](#)

Часть 8: обзор возможностей стандарта ES6

О стандарте ES6

Стандарт ES6, который правильнее было бы называть ES2015 или ECMAScript 2015 (это — его официальные наименования, хотя все называют его ES6), появился через 4 года после выхода предыдущего стандарта — ES5.1. На разработку всего того, что вошло в стандарт ES5.1, ушло около десяти лет. В наши дни всё то, что появилось в этом стандарте, превратилось в привычные инструменты JS-разработчика. Надо отметить, что ES6 внёс в язык серьёзные изменения (сохраняя обратную совместимость с его предыдущими версиями). Для того чтобы оценить масштаб этих изменений, можно отметить, что размер документа, описывающего стандарт ES5, составляет примерно 250 страниц, а стандарт ES6 описывается в документе, состоящем уже из приблизительно 600 страниц.

В перечень наиболее важных новшеств стандарта ES2015 можно включить следующие:

- Стрелочные функции.
- Промисы.
- Генераторы
- Ключевые слова `let` и `const`.
- Классы.
- Модули.
- Поддержка шаблонных литералов.
- Поддержка параметров функций, задаваемых по умолчанию.
- Оператор `spread`.
- Деструктурирующее присваивание.
- Расширение возможностей объектных литералов.
- Цикл `for...of`.
- Поддержка структур данных `Map` и `Set`.

Рассмотрим эти возможности.

Стрелочные функции

Стрелочные функции изменили внешний вид и особенности работы JavaScript-кода. С точки зрения внешнего вида их использование делает объявления функций короче и проще. Вот объявление обычной функции.

```
const foo = function foo() {  
  
    //...  
  
}
```

А вот практически такая же (хотя и не полностью аналогичная вышеобъявленной) стрелочная функция.

```
const foo = () => {  
  
    //...  
  
}
```

Если тело стрелочной функции состоит лишь из одной строки, результат выполнения которой нужно из этой функции вернуть, то записывается она ещё короче.

```
const foo = () => doSomething()
```

Если стрелочная функция принимает лишь один параметр, записать её можно следующим образом.

```
const foo = param => doSomething(param)
```

Надо отметить, что с появлением стрелочных функций обычные функции никуда не делись, их всё так же можно использовать в коде, работают они так же, как и прежде.

Особенности ключевого слова `this` в стрелочных функциях

У стрелочных функций нет собственного значения `this`, они наследуют его из контекста выполнения.

Это устраняет проблему, для решения которой при использовании обычных функций приходилось, для сохранения контекста, использовать конструкции наподобие `var that = this`. Однако, как было показано в предыдущих частях руководства, это изменение серьёзно сказывается на особенностях работы со стрелочными функциями и на сфере их применения.

Промисы

Промисы позволяют избавиться от широко известной проблемы, называемой «адом коллбэков», хотя их использование подразумевает применение достаточно сложных структур. Эта проблема была решена в стандарте ES2017 с появлением конструкции `async/await`, которая основана на промисах.

JavaScript-разработчики использовали промисы и до появления стандарта ES2015, применяя для этого различные библиотеки (например — `jQuery`, `q`, `deferred.js`, `vow`). Это говорит о важности и востребованности данного механизма. Разные библиотеки реализуют его по-разному, появление стандарта в этой области можно считать весьма позитивным фактом.

Вот код, написанный с использованием функций обратного вызова (коллбэков).

```
setTimeout(function() {  
  console.log('I promised to run after 1s')  
  setTimeout(function() {  
    console.log('I promised to run after 2s')  
  }, 1000)  
}, 1000)
```

С использованием промисов это можно переписать следующим образом.

```
const wait = () => new Promise((resolve, reject) => {  
  setTimeout(resolve, 1000)  
})  
  
wait().then(() => {  
  console.log('I promised to run after 1s')  
  return wait()  
})  
  
.then(() => console.log('I promised to run after 2s'))
```

Генераторы

Генераторы — это особые функции, которые могут приостанавливать собственное выполнение и возобновлять его. Это позволяет, пока генератор находится в состоянии ожидания, выполняться другому коду.

Генератор самостоятельно принимает решение о том, что ему нужно приостановиться и позволить другому коду, «ожидающему» своей очереди, выполниться. При этом у генератора есть возможность продолжить своё выполнение после того, как та операция, результатов выполнения которой он ждёт, окажется выполненной.

Всё это делается благодаря единственному простому ключевому слову `yield`. Когда в генераторе встречается это ключевое слово — его выполнение приостанавливается.

Генератор может содержать множество строк с этим ключевым словом, приостанавливая собственное выполнение несколько раз. Генераторы объявляют с использованием конструкции `*function`. Эту звёздочку перед словом `function` не стоит принимать за нечто вроде оператора разыменования указателя, применяемого в языках наподобие C, C++ или Go.

Генераторы знаменуют своим появлением новую парадигму программирования на JavaScript. В частности, они дают возможность двустороннего обмена данными между генератором и другим кодом, позволяют создавать долгоживущие циклы `while`, которые не «подвешивают» программу.

Рассмотрим пример, иллюстрирующий особенности работы генераторов. Вот сам генератор.

```
function *calculator(input) {  
    var doubleThat = 2 * (yield (input / 2))  
    var another = yield (doubleThat)  
    return (input * doubleThat * another)  
}
```

Такой командой мы его инициализируем.

```
const calc = calculator(10)
```

Затем мы обращаемся к его итератору.

```
calc.next()
```

Эта команда запускает итератор, она возвращает такой объект.

```
{  
    done: false  
    value: 5  
}
```

Здесь происходит следующее. В коде выполняется функция, использующая значение `input`, переданное конструктору генератора. Код генератора выполняется до тех пор, пока в нём не встретится ключевое слово `yield`. В этот момент он возвращает результат деления `input` на 2, что, так как `input` равняется 10, даёт число 5. Это число мы получаем благодаря итератору, и, вместе с ним, указание на то, что работа генератора пока не завершена (свойство `done` в объекте, возвращённом итератором, установлено в значение `false`), то есть, функция пока лишь приостановлена.

При следующем вызове итератора мы передаём в генератор число 7.

```
calc.next(7)
```

В ответ на это итератор возвращает нам следующий объект.

```
{  
  
  done: false  
  
  value: 14  
  
}
```

Здесь число 7 было использовано при вычислении значения `doubleThat`.

На первый взгляд может показаться, что код `input / 2` представляет собой нечто вроде аргумента некоей функции, но это — лишь значение, возвращаемое на первой итерации. Здесь мы это значение пропускаем и используем новое входное значение 7, умножая его на 2. После этого мы доходим до второго ключевого слова `yield`, в результате значение, полученное на второй итерации, равняется 14.

На следующей итерации, которая является последней, мы передаём в генератор число 100.

```
calc.next(100)
```

В ответ получаем следующий объект.

```
{  
  
  done: true  
  
  value: 14000  
  
}
```

Итерация завершена (в генераторе больше не встречается ключевое слово `yield`), в объекте возвращается результат вычисления выражения `(input * doubleThat * another)`, то есть — $10 * 14 * 100$ и указание на завершение работы итератора (`done: true`)

Ключевые слова `let` и `const`

В JavaScript для объявления переменных всегда использовалось ключевое слово `var`. Такие переменные имеют функциональную область видимости. Ключевые слова `let` и `const` позволяют, соответственно, объявлять переменные и константы, обладающие блочной областью видимости.

Это означает, что, например, переменная, объявленная с помощью ключевого слова `let` в цикле, внутри блока `if` или внутри обычного блока кода, ограниченного фигурными скобками, за пределы этого блока не выйдет. Переменные же, объявленные с помощью `var`, в таких блоках не удерживаются, становясь доступными в функции, на уровне которой они объявлены.

Ключевое слово `const` работает так же как и `let`, но с его помощью объявляют константы, которые являются иммутабельными.

В современном JS-коде ключевое слово `var` используется редко. Оно уступило место ключевым словам `let` и `const`. При этом, что может показаться необычным, ключевое слово `const` используется сегодня весьма широко, что говорит о популярности идей иммутабельности сущностей в современном программировании.

Классы

Сложилось так, что JavaScript был единственным чрезвычайно широко распространённым языком, использующим модель прототипного наследования. Программисты, переходящие на JS с языков, реализующих механизм наследования, основанный на классах, чувствовали себя в такой среде неуютно. Стандарт ES2015 ввёл в JavaScript поддержку классов. Это, по сути, «синтаксический сахар» вокруг внутренних механизмов JS, использующих прототипы. Однако это влияет на то, как именно пишут JS-приложения.

Теперь механизмы наследования в JavaScript выглядят как аналогичные механизмы в других объектно-ориентированных языках.

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
    hello() {  
        return 'Hello, I am ' + this.name + '.'  
    }  
}  
  
class Actor extends Person {  
    hello() {  
        return super.hello() + ' I am an actor.'  
    }  
}  
  
var tomCruise = new Actor('Tom Cruise')  
  
console.log(tomCruise.hello())
```

Эта программа выводит в консоль текст `Hello, I am Tom Cruise. I am an actor.`

В JS-классах нельзя объявлять переменные экземпляров, их нужно инициализировать в конструкторах.

Конструктор класса

У классов есть специальный метод, `constructor`, который вызывается при создании экземпляра класса с использованием ключевого слова `new`.

Ключевое слово `super`

Ключевое слово `super` позволяет обращаться к родительскому классу из классов-потомков.

Геттеры и сеттеры

Геттер для свойства можно задать следующим образом.

```
class Person {  
    get fullName() {  
        return `${this.firstName} ${this.lastName}`  
    }  
}
```

Сеттер можно описать так, как показано ниже.

```
class Person {  
    set age(years) {
```

```
    this.theAge = years
  }
}
```

С геттерами и сеттерами работают так, как будто они представляют собой не функции, а обычные свойства объектов.

Модули

До появления стандарта ES2015 существовало несколько конкурирующих подходов к работе с модулями. В частности, речь идёт о технологиях RequireJS и CommonJS. Такая ситуация приводила к разногласиям в сообществе JS-разработчиков.

В наши дни, благодаря стандартизации модулей в ES2015, ситуация постепенно нормализуется.

Импорт модулей

Модули импортируют с использованием конструкции вида `import...from...`. Вот несколько примеров.

```
import * as something from 'mymodule'

import React from 'react'

import { React, Component } from 'react'

import React as MyLibrary from 'react'
```

Экспорт модулей

Внутренние механизмы модуля закрыты от внешнего мира, но из модуля можно экспортировать всё то, что он может предложить другим модулям. Делается это с помощью ключевого слова `export`.

```
export var foo = 2

export function bar() { /* ... */ }
```

Шаблонные литералы

Шаблонные литералы представляют собой новый способ описания строк в JavaScript. Вот как это выглядит.

```
const aString = `A string`
```

Кроме того, использование синтаксиса шаблонных литералов позволяет внедрять в строки выражения, интерполировать их. Делается это с помощью конструкции вида `${a_variable}`. Вот простой пример её использования

```
const v = 'test'

const str = `something ${v}` //something test
```

Вот пример посложнее, иллюстрирующий возможность вычисления любых выражений и подстановки их результатов в строку.

```
const str = `something ${1 + 2 + 3}`

const str2 = `something ${foo() ? 'x' : 'y' }`
```

Благодаря использованию шаблонных литералов гораздо легче стало объявлять многострочные строки.

```
const str3 = `Hey

this
```

```
string
```

```
is awesome!`
```

Сравните это с тем, что приходилось делать для описания многострочных строк при использовании возможностей, имевшихся в языке до ES2015.

```
var str = 'One\n' +  
  
'Two\n' +  
  
'Three'
```

Параметры функций, задаваемые по умолчанию

Теперь функции поддерживают параметры, используемые по умолчанию — в том случае, если при вызове функций им не передаются соответствующие аргументы.

```
const foo = function(index = 0, testing = true) { /* ... */ }  
  
foo()
```

Оператор spread

Оператор `spread` (оператор расширения) позволяет «раскрывать» массивы, объекты или строки. Этот оператор выглядит как три точки (`...`). Сначала рассмотрим его на примере массива.

```
const a = [1, 2, 3]
```

Вот как на основании этого массива создать новый массив.

```
const b = [...a, 4, 5, 6]
```

Вот как создать копию массива.

```
const c = [...a]
```

Этот оператор работает и с объектами. Например — вот как с его помощью можно клонировать объект.

```
const newObj = { ...oldObj }
```

Применив оператор `spread` к строке, можно преобразовать её в массив, в каждом элементе которого содержится один символ из этой строки.

```
const hey = 'hey'  
  
const arrayized = [...hey] // ['h', 'e', 'y']
```

Этот оператор, помимо вышеописанных вариантов его применения, удобно использовать при вызове функций, ожидающих обычный список аргументов, с передачей им массива с этими аргументами.

```
const f = (foo, bar) => {}  
  
const a = [1, 2]  
  
f(...a)
```

Раньше это делалось с использованием конструкции вида `f.apply(null, a)`, но такой код и писать сложнее, и читается он хуже.

Деструктурирующее присваивание

Техника деструктурирующего присваивания позволяет, например, взять объект, извлечь из него некоторые значения и поместить их в именованные переменные или константы.

```
const person = {  
  firstName: 'Tom',  
  lastName: 'Cruise',  
  actor: true,  
  age: 54,  
}  
  
const {firstName: name, age} = person
```

Здесь из объекта извлекаются свойства `firstName` и `age`. Свойство `age` записывается в объявляемую тут же константу с таким же именем, а свойство `firstName`, после извлечения, попадает в константу `name`.

Деструктурирующее присваивание подходит и для работы с массивами.

```
const a = [1,2,3,4,5]  
  
const [first, second, , , fifth] = a
```

В константы `first`, `second` и `fifth` попадут, соответственно, первый, второй и пятый элементы массива.

Расширение возможностей объектных литералов

В ES2015 значительно расширены возможности описания объектов с помощью объектных литералов.

Упрощение включения в объекты переменных

Раньше, чтобы назначить какую-нибудь переменную свойством объекта, надо было пользоваться следующей конструкцией.

```
const something = 'y'  
  
const x = {  
  something: something  
}
```

Теперь то же самое можно сделать так.

```
const something = 'y'  
  
const x = {  
  something  
}
```

Прототипы

Прототип объекта теперь можно задать с помощью следующей конструкции.

```
const anObject = { y: 'y' }  
  
const x = {
```

```
__proto__: anObject
}
```

Ключевое слово `super`

С использованием ключевого слова `super` объекты могут обращаться к объектам-прототипам. Например — для вызова их методов, имеющих такие же имена, как методы самих этих объектов.

```
const anObject = { y: 'y', test: () => 'zoo' }

const x = {
  __proto__: anObject,
  test() {
    return super.test() + 'x'
  }
}

x.test() //zoox
```

Вычисляемые имена свойств

Вычисляемые имена свойств формируются на этапе создания объекта.

```
const x = {
  ['a' + '_' + 'b']: 'z'
}

x.a_b //z
```

Цикл `for...of`

В 2009 году, в стандарте ES5, появились циклы `forEach()`. Это — полезная конструкция, к минусам которой относится тот факт, что такие циклы очень неудобно прерывать. Классический цикл `for` в ситуациях, когда выполнение цикла нужно прервать до его обычного завершения, оказывается гораздо более адекватным выбором.

В ES2015 появился цикл `for...of`, который, с одной стороны, отличается краткостью синтаксиса и удобством `forEach`, а с другой — поддерживает возможности по досрочному выходу из цикла.

Вот пара примеров цикла `for...of`.

```
//перебор значений элементов массива

for (const v of ['a', 'b', 'c']) {
  console.log(v);
}

//перебор значений элементов массива с выводом их индексов благодаря
использованию метода entries()

for (const [i, v] of ['a', 'b', 'c'].entries()) {
  console.log(i, v);
}
```

```
}
```

Структуры данных Map и Set

В ES2015 появились структуры данных Map и Set (а также их «слабые» варианты WeakMap и WeakSet, использование которых позволяет улучшить работу «сборщика мусора» — механизма, ответственного за управление памятью в JS-движках). Это — весьма популярные структуры данных, которые, до появления их официальной реализации, приходилось имитировать имеющимися средствами языка.

Часть 9: обзор возможностей стандартов ES7, ES8 и ES9

Стандарт ES7

Стандарт ES7, который, в соответствии с официальной терминологией, называется ES2016, вышел летом 2016 года. Он, в сравнении с ES6, принёс в язык не так много нового. В частности, речь идёт о следующем:

- Метод `Array.prototype.includes()`.
- Оператор возведения в степень.

Метод `Array.prototype.includes()`

Метод `Array.prototype.includes()` предназначен для проверки наличия в массиве некоего элемента. Находя в массиве искомое, он возвращает `true`, не находя — `false`. До ES7 для выполнения той же операции служил метод `indexOf()`, который возвращает, в случае нахождения элемента, первый индекс, по которому его можно обнаружить в массиве. Если же `indexOf()` элемента не находит — он возвращает число `-1`.

В соответствии с правилами преобразования типов JavaScript число `-1` преобразуется в `true`. Как результат, для проверки результатов работы `indexOf()` следовало пользоваться не особенно удобной конструкцией следующего вида.

```
if ([1,2].indexOf(3) === -1) {  
    console.log('Not found')  
}
```

Если в подобной ситуации, полагая, что `indexOf()`, не находя элемента, возвращает `false`, воспользоваться чем-то вроде показанного ниже, код будет работать неправильно.

```
if (![1,2].indexOf(3)) { //неправильно  
    console.log('Not found')  
}
```

В данном случае оказывается, что конструкция `![1,2].indexOf(3)` даёт `false`.

С использованием метода `includes()` подобные сравнения выглядят гораздо логичнее.

```
if (![1,2].includes(3)) {  
    console.log('Not found')  
}
```

В данном случае конструкция `[1,2].includes(3)` возвращает `false`, это значение оператор `!` превращает в `true` и в консоль попадает сообщение о том, что искомый элемент в массиве не найден.

Оператор возведения в степень

Оператор возведения в степень выполняет ту же функцию, что и метод `Math.pow()`, но пользоваться им удобнее, чем библиотечной функцией, так как он является частью языка.

```
Math.pow(4, 2) == 4 ** 2 //true
```

Этот оператор можно считать приятным дополнением JS, которое пригодится в приложениях, выполняющих некие вычисления. Похожий оператор существует и в других языках программирования.

Стандарт ES8

Стандарта ES8 (ES2017) вышел в 2017 году. Он, как и ES7, внёс в язык не особенно много нового. А именно, речь идёт о следующих возможностях:

- Дополнение строк до заданной длины.
- Метод `Object.values()`.
- Метод `Object.entries()`.
- Метод `Object.getOwnPropertyDescriptors()`.
- Завершающие запятые в параметрах функций.
- Асинхронные функции.
- Работа с разделяемой памятью и атомарные операции.

Дополнение строк до заданной длины

В ES8 появились два новых метода объекта `String` — `padStart()` и `padEnd()`.

Метод `padStart()` заполняет текущую строку другой строкой до тех пор, пока итоговая строка не достигнет нужной длины. Заполнение происходит в начале строки (слева). Вот как пользоваться этим методом.

```
str.padStart(targetLength [, padString])
```

Здесь `str` — это текущая строка, `targetLength` — длина итоговой строки (если она меньше длины текущей строки — эта строка будет возвращена без изменений), `padString` — необязательный параметр — строка, используемая для заполнения текущей строки. Если параметр `padString` не задан — для дополнения текущей строки до заданной длины используется символ пробела.

Метод `padEnd()` аналогичен `padStart()`, но заполнение строки происходит справа.

Рассмотрим примеры использования этих методов.

```
const str = 'test'.padStart(10)

const str1 = 'test'.padEnd(10, '*')
```

```
console.log(`'${str}'`) // '      test'

console.log(`'${str1}'`) //'test*****'
```

Здесь, при использовании `padStart()` с указанием лишь желаемой длины итоговой строки, в начало исходной строки были добавлены пробелы. При использовании `padEnd()` с указанием длины итоговой строки и строки для её заполнения в конец исходной строки были добавлены символы `*`.

Метод `Object.values()`

Этот метод возвращает массив, содержащий значения собственных свойств объекта, то есть таких свойств, которые содержит сам объект, а не тех, которые доступны ему через цепочку прототипов.

Вот как им пользоваться.

```
const person = { name: 'Fred', age: 87 }

const personValues = Object.values(person)

console.log(personValues) // ['Fred', 87]
```

Этот метод применим и к массивам.

Метод `Object.entries()`

Этот метод возвращает массив, каждый элемент которого также является массивом, содержащим, в формате `[key, value]`, ключи и значения собственных свойств объекта.

```
const person = { name: 'Fred', age: 87 }

const personValues = Object.entries(person)

console.log(personValues) // [['name', 'Fred'], ['age', 87]]
```

При применении этого метода к массивам в качестве ключей выводятся индексы элементов, а в качестве значений выводится то, что хранится в массиве по соответствующим индексам.

Метод `getOwnPropertyDescriptors()`

Этот метод возвращает сведения обо всех собственных свойствах объекта. Со свойствами объектов ассоциированы наборы атрибутов (дескрипторы). В частности, речь идёт о следующих атрибутах:

- `value` — значение свойства объекта.
- `writable` — содержит `true` если свойство можно менять.
- `get` — содержит функцию-геттер, связанную со свойством, или, если такой функции нет — `undefined`.
- `set` — содержит функцию-сеттер для свойства или `undefined`.
- `configurable` — если тут будет `false` — свойство нельзя удалять, нельзя менять его атрибуты за исключением значения.
- `enumerable` — если в этом свойстве будет содержаться `true` — свойство является перечислимым.

Вот как пользоваться этим методом.

```
Object.getPrototypeOfDescriptors(obj)
```

Он принимает объект, сведения о свойствах которого нужно узнать, и возвращает объект, содержащий эти сведения.

```
const person = { name: 'Fred', age: 87 }

const propDescr = Object.getPrototypeOfDescriptors(person)

console.log(propDescr)

/*
{ name:
  { value: 'Fred',
    writable: true,
    enumerable: true,
    configurable: true },
  age:
```



```
{ value: 87,  
  writable: true,  
  enumerable: true,  
  configurable: true } }  
*/
```

Зачем нужен этот метод? Дело в том, что он позволяет создавать мелкие копии объектов, копируя, помимо других свойств, геттеры и сеттеры. Этого нельзя было сделать, пользуясь для копирования объектов методом `Object.assign()`, который появился в стандарте ES6.

В следующем примере имеется объект с сеттером, который выводит, с помощью `console.log()` то, что пытаются записать в его соответствующее свойство.

```
const person1 = {  
  set name(newName) {  
    console.log(newName)  
  }  
}
```

```
person1.name = 'x' // x
```

Попробуем скопировать этот объект, воспользовавшись методом `assign()`.

```
const person2 = {}  
Object.assign(person2, person1)
```

```
person2.name = 'x' // в консоль ничего не попадает, сеттер не скопирован
```

Как видно, такой подход не работает. Свойство `name`, которое в исходном объекте было сеттером, теперь представлено в виде обычного свойства.

Теперь выполним копирование объекта с использованием методов `Object.defineProperties()` (он появился в ES5.1) и `Object.getOwnPropertyDescriptors()`.

```
const person3 = {}  
Object.defineProperties(person3,  
  Object.getOwnPropertyDescriptors(person1))
```

```
person3.name = 'x' //x
```

Здесь в копии объекта сеттер остался.

Надо отметить, что ограничения, характерные для `Object.assign()`, свойственны и для метода `Object.create()` при использовании его для клонирования объектов.

Завершающие запятые в параметрах функций

Эта возможность позволяет оставлять запятую в конце списка параметров или аргументов, соответственно, при объявлении и при вызове функций.

```
const doSomething = (  
  var1,  
  var2,  
) => {  
  //...  
}  
  
doSomething(  
  'test1',  
  'test2',  
)
```

Это повышает удобство работы с системами контроля версий. А именно, речь идёт о том, что, при добавлении новых параметров в функцию, не приходится менять существующий код только ради вставки запятой.

Асинхронные функции

В стандарте ES2017 появилась конструкция `async/await`, которую можно считать важнейшим новшеством этой версии языка.

Асинхронные функции представляют собой комбинацию промисов и генераторов, они упрощают конструкции, для описания которых раньше требовался большой объём шаблонного кода и неудобные в работе цепочки промисов. Фактически, речь идёт о высокоуровневой абстракции над промисами.

Когда в стандарте ES2015 появились промисы, они призваны были решить существующие проблемы с асинхронным кодом, что они и сделали. Но за те два года, которые разделяют стандарты ES2015 и ES2017, стало ясно, что промисы нельзя считать окончательным решением этих проблем.

В частности, промисы были нацелены на решение проблемы «ада коллбэков», но, решив эту проблему, они сами показали себя не с лучшей стороны из-за усложнения кода, в котором они используются. Собственно говоря, конструкция `async/await` решает проблему промисов и повышает удобство работы с асинхронным кодом.

Рассмотрим пример.

```
function doSomethingAsync() {  
  return new Promise((resolve) => {  
    setTimeout(() => resolve('I did something'), 3000)  
  })  
}  
  
async function doSomething() {  
  console.log(await doSomethingAsync())  
}
```

```
console.log('Before')

doSomething()

console.log('After')
```

Этот код выведет в консоль следующее.

Before

After

I did something

Как видно, после вызова `doSomething()` программа продолжает выполняться, после `Before` в консоль тут же выводится `After`, а после того, как пройдут три секунды, выводится `I did something`.

Последовательный вызов асинхронных функций

При необходимости асинхронные функции могут формировать нечто вроде цепочек вызовов. Такие конструкции отличаются лучшей читабельностью, чем нечто подобное, основанное исключительно на промисах. Это можно видеть на следующем примере.

```
function promiseToDoSomething() {

  return new Promise((resolve)=>{

    setTimeout(() => resolve('I did something'), 10000)

  })

}

async function watchOverSomeoneDoingSomething() {

  const something = await promiseToDoSomething()

  return something + ' and I watched'

}

async function watchOverSomeoneWatchingSomeoneDoingSomething() {

  const something = await watchOverSomeoneDoingSomething()

  return something + ' and I watched as well'

}

watchOverSomeoneWatchingSomeoneDoingSomething().then((res) => {

  console.log(res) // I did something and I watched and I watched as well

})
```

Разделяемая память и атомарные операции

Здесь речь идёт об объекте [SharedArrayBuffer](#), который позволяет описывать разделяемые области памяти, и об объекте [Atomics](#), который содержит набор атомарных операций в виде статических методов. Подробности о возможностях, которые дают программисту эти объекты, можно почитать [здесь](#).

Стандарт ES9

ES9 (ES2018) — это самая свежая на момент публикации данного материала версия стандарта. Вот её основные возможности:

- Применение операторов `spread` и `rest` к объектам.
- Асинхронные итераторы.
- Метод `Promise.prototype.finally()`.
- Улучшения регулярных выражений.

Применение операторов `spread` и `rest` к объектам

Мы уже говорили об операторах `rest` и `spread`, которые появились в ES6 и могут быть использованы для работы с массивами. Оба они выглядят как три точки. Оператор `rest`, в следующем примере деструктурирования массива, позволяет поместить его первый и второй элементы в константы `first` и `second`, а все остальные — в константу `others`.

```
const numbers = [1, 2, 3, 4, 5]

const [first, second, ...others] = numbers

console.log(first) //1
console.log(second) //2
console.log(others) //[ 3, 4, 5 ]
```

Оператор `spread` позволяет передавать массивы в функции, ожидающие обычные списки параметров.

```
const numbers = [1, 2, 3, 4, 5]

const sum = (a, b, c, d, e) => a + b + c + d + e

const res = sum(...numbers)

console.log(res) //15
```

Теперь, используя тот же подход, можно работать и с объектами. Вот пример использования оператора `rest` в операции деструктурирующего присваивания.

```
const { first, second, ...others } =

  { first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }

console.log(first) //1
console.log(second) //2
console.log(others) //{ third: 3, fourth: 4, fifth: 5 }
```

Вот оператор `spread`, применяемый при создании нового объекта на основе существующего. Этот пример продолжает предыдущий.

```
const items = { first, second, ...others }

console.log(items) //{ first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

Асинхронные итераторы

Новая конструкция `for-await-of` позволяет вызывать асинхронные функции, возвращающие промисы, в циклах. Такие циклы ожидают разрешения промиса перед переходом к следующему шагу. Вот как это выглядит.

```
for await (const line of readLines(filePath)) {
```

```
    console.log(line)
}
```

При этом надо отметить, что подобные циклы нужно использовать в асинхронных функциях — так же, как это делается при работе с конструкцией `async/await`.

Метод `Promise.prototype.finally()`

Если промис успешно разрешается — осуществляется вызов очередного метода `then()`. Если что-то идёт не так — вызывается метод `catch()`. Метод `finally()` позволяет выполнять некий код независимо от того, что происходило до этого.

```
fetch('file.json')
    .then(data => data.json())
    .catch(error => console.error(error))
    .finally(() => console.log('finished'))
```

Улучшения регулярных выражений

В регулярных выражениях появилась возможность ретроспективной проверки строк (`?<=`). Это позволяет искать в строках некие конструкции, перед которыми есть какие-то другие конструкции.

Возможность опережающих проверок, использующая конструкцию `?=`, имела в регулярных выражениях, реализованных в JavaScript, и до стандарта ES2018. Такие проверки позволяют узнать, следует ли за неким фрагментом строки другой фрагмент.

```
const r = /Roger(?= Waters)/
const res1 = r.test('Roger is my dog')
const res2 = r.test('Roger is my dog and Roger Waters is a famous musician')
console.log(res1) //false
console.log(res2) //true
```

Конструкция `?!` выполняет обратную операцию — совпадение будет найдено только в том случае, если за заданной строкой не идёт другая строка.

```
const r = /Roger(?! Waters)/g
const res1 = r.test('Roger is my dog')
const res2 = r.test('Roger is my dog and Roger Waters is a famous musician')
console.log(res1) //true
console.log(res2) //false
```

При ретроспективной проверке, как уже было сказано, используется конструкция `?<=`.

```
const r = /(?!<=Roger) Waters/
const res1 = r.test('Pink Waters is my dog')
const res2 = r.test('Roger is my dog and Roger Waters is a famous musician')
console.log(res1) //false
```

```
console.log(res2) //true
```

Операцию, обратную описанной, можно выполнить с помощью конструкции `?<!`.

```
const r = /(?!Roger) Waters/
```

```
const res1 = r.test('Pink Waters is my dog')
```

```
const res2 = r.test('Roger is my dog and Roger Waters is a famous musician')
```

```
console.log(res1) //true
```

```
console.log(res2) //false
```

Управляющие последовательности Unicode в регулярных выражениях

В регулярных выражениях можно использовать класс `\d`, соответствующий любой цифре, класс `\s`, соответствующий любому пробельному символу, класс `\w`, который соответствует любому буквенно-цифровому символу, и так далее. Возможность, о которой идёт речь, расширяет набор классов, которыми можно пользоваться в регулярных выражениях, позволяя работать с Unicode-последовательностями. Речь идёт о классе `\p{}` и об обратном ему классе `\P{}`.

В Unicode каждый символ имеет набор свойств. Эти свойства указываются в фигурных скобках группы `\p{}`. Так, например, свойство `Script` определяет семейство языков, к которому принадлежит символ, свойство `ASCII`, логическое, принимает значение `true` для ASCII-символов, и так далее. Например, выясним, содержат ли некие строки исключительно ASCII-символы.

```
console.log(r.test('abc')) //true
```

```
console.log(r.test('ABC@')) //true
```

```
console.log(r.test('ABCЖ')) //false
```

Свойство `ASCII_Hex_Digit` принимает значение `true` только для символов, которые можно использовать для записи шестнадцатеричных чисел.

```
const r = /^ \p{ASCII_Hex_Digit}+$/u
```

```
console.log(r.test('0123456789ABCDEF')) //true
```

```
console.log(r.test('H')) //false
```

Существует и множество других подобных свойств, которые используются так же, как вышеописанные. Среди них — `Uppercase`, `Lowercase`, `White_Space`, `Alphabetic`, `Emoji`.

Вот, например, как с помощью свойства `Script` определить, какой алфавит используется в строке. Здесь мы проверяем строку на использование греческого алфавита.

```
const r = /^ \p{Script=Greek}+$/u
```

```
console.log(r.test('ελληνικά')) //true
```

```
console.log(r.test('hey')) //false
```

Подробности об этих свойствах можно почитать [здесь](#).

Именованные группы

Захваченным группам символов в ES2018 можно давать имена. Вот как это выглядит.

```
const re = /(?!<year>\d{4})-(?!<month>\d{2})-(?!<day>\d{2})/

const result = re.exec('2015-01-02')

console.log(result)

/*
[ '2015-01-02',
  '2015',
  '01',
  '02',
  index: 0,
  input: '2015-01-02',
  groups: { year: '2015', month: '01', day: '02' } ]
*/
```

Без использования именованных групп те же данные были бы доступны лишь как элементы массива.

```
const re = /(\d{4})-(\d{2})-(\d{2})/

const result = re.exec('2015-01-02')

console.log(result)

/*
[ '2015-01-02',
  '2015',
  '01',
  '02',
  index: 0,
  input: '2015-01-02',
  groups: undefined ]
*/
```

Флаг регулярных выражений *s*

Использование флага *s* приводит к тому, что символ *.* (точка) будет, кроме прочих, соответствовать и символу новой строки. Без использования этого флага точка соответствует любому символу за исключением символа новой строки.

```
console.log(/hi.welcome/.test('hi\nwelcome')) // false
console.log(/hi.welcome/s.test('hi\nwelcome')) // true
```