

JavaScript, полный конспект

Ссылки

БЭМ методология [здесь](#).

Шаблоны для вёрстки [тут](#).

JS, React, Redux

[regexp101](#)

Помогает составлять регулярные выражения

[cors-anywhere.herokuapp.com](#)

Позволяет делать cors-запросы, когда браузер блокирует их.

[htmldom.dev](#)

Очень годный ресурс с готовыми решениями вопросов по DOM

[rajdee.gitbooks.io](#)

Годно про Redux на русском

HTML, CSS

[loading.io](#)

Готовые индикаторы загрузки

[www.w3schools.com](#)

Простой спиннер

[bootswatch.com](#)

Темы для Bootstrap

[heropatterns.com](#)

svg-фоны для html.

NPM

[\\$http-server](#)

Идеи для программ

Статья про то, какие первые проекты [сделать](#).

Главная страница

Перечень тестовых страниц и игр можно оформить в виде плитки, как в уроке 8 у Валака.

Simon

игра со звуком и цветом. Можно вставить фразы Глада Валакаса.

Можно сделать её проходимой: текстом рисуется картинка при каждом удачном решении. Добавить прогресс-бар.

Компонент кнопка:

В создающую функцию передаётся объект со свойствами: цвет, расположение, звук?

Калькулятор времени

Новые интервалы добавляются через клик по кнопке добавить.

Музыкальный секвенсор

Есть сетка-таблица. Значения сверху вниз – это 7 нот. Слева-направо – такты. Пользователь кликом мыши выбирает, какие ноты будут проигрываться и с какой скоростью (через интервальный таймер реализовать?). При запуске ноты проигрываются слева направо. Можно какое-нибудь соло замутить простое, типа кузнецика или из металлики.

Проигрываемые ноты могут подсвечиваться красиво.

Фоновое изображение

Скачать текстуры и придумать какую-нибудь хуйню типа простой бродилки со звуком с использованием анимированного фонового изображения. Можно даже 2 анимации с 2-мя фонами делать, чтобы был эффект 3д.

Сообщение на форуме

Сделать страницу, на которую можно отправить сообщение. Сообщение красиво оформить, будто это настоящий форум. Пользователь пишет в форме сообщение, оно появляется в импровизированной ленте. После перезагрузки всё очищается.

Тетрис

Есть несколько квадратов. Их надо перетягивать мышкой в одну линию, и тогда они загораются и со звуком исчезают. Мб случайно появляется ещё одна фигура.

Червяк ползёт

Можно сделать поле, на котором ты кликаешь мышкой и туда ползёт анимированный червяк или квадратик двигается.

Случайный спавн

Есть поле, на нём фигура. Щёлкаешь на фигуру и она появляется в случайном месте.

Полоса

Есть дистанция и полоска сверху, которая увеличивается и уменьшается. Надо кликнуть в то время, когда полоса максимальная, чтобы предмет вылетел максимально сильно и что-то толкнул или куда-то залетел. Это типа сила толчка.

Кликер

Загарается замочка жёлтым цветом – кликай, красным – не кликай или проиграешь.

И ещё такая игра как [тут](#).

Змейка

[Тут](#). Лучше сделать другую 2д версию, а потом 3д версию.

3д часы

Как в уроке Валака про трёхмерный куб. Только циферблат переворачивается как костяшка домино.

Погладь кота, сука!

Поле, на котором размещён квадрат. Наводишь мышкой на квадрат – запускается анимация движения и квадрат постоянно ускользает из-под курсора. Если упирается в стенку, то двигается в другую сторону. Вектор каждый раз должен выбираться случайно. Квадрат редко ускоряется, а потом, когда ушёл из-под курсора, плавно останавливается.

Можно повесить анимацию спящего кота в виде глаз, ушей и рта. Когда не трогаешь – спит, когда трогаешь – просыпается и уходит. Направление морды – по вектору определяется.

Как выполнить скрипт

Выполнить скрипт в node:

```
node file.js
```

Код может быть вставлен в любое место HTML-документа внутри тега <script>.

Код автоматически выполнится, когда браузер его обработает.

```
<script>
  alert( 'Привет, мир!' );
</script>
```

К веб-станице можно подключить **внешние файлы** со скриптами с помощью атрибута src.

Можно указывать относительный, абсолютный путь или полный.

Для каждого отдельного файла используется свой тег script. Для подключения нескольких скриптов используйте несколько тегов.

```
<script src="/path/to/script.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js">
</script>
```

Как правило, только простейшие скрипты помещаются в HTML. Более сложные выделяются в отдельные файлы.

Польза от отдельных файлов в том, что браузер загрузит скрипт отдельно и сможет хранить его в [кеше](#).

Другие страницы, которые подключают тот же скрипт, смогут брать его из кеша вместо повторной загрузки из сети. И таким образом файл будет загружаться с сервера только один раз.

Если атрибут src установлен, содержимое тега script будет **игнорироваться**.

В одном теге <script> нельзя использовать одновременно атрибут src и код внутри.

Вопросы

Как правильно документировать функции?

Написать про eval(). [MDN](#).

Что такое объект [Reflect](#)?

Что такое V8?

Плохо знаю темы

Aleks

Набор: CSS HTML JS, TS, фреймворк, REST API, GIT, DOCKER вполне хватит для старта.

Понимание препроцессора SCSS будет большим плюсом для написание стилей в JSS, Styled components. Чаще всего это работа с темами, передача проперти, и наследование.

В сборке использовать обязательно

1. ES LINT
2. TS LINT
3. Commit LINT

Написание тестов

Генераторы

Документирование кода

EventMixin

TypeScript

Event Loop

Task queue

слайдер, галерея, поиск, фильтр, сортировка, калькулятор цен.

Верстка на Grid, Flexbox

Умение работать с макетами из Figma

Webpack

Babel

Redux

sagas,

websockets

js как сделать уведомление <https://habr.com/ru/company/ruvds/blog/350486/>

Html

Гриды

Флоаты

Семантика для HTML

Словарь

JavaScript — мультипарадигменный язык программирования. Поддерживает стили:

- объектно-ориентированный
- императивный
- функциональный.

Является реализацией спецификации ECMAScript.

Основные архитектурные черты:

- динамическая типизация (переменная связывается с типом в момент присваивания, а не объявления);
- слабая типизация (допустимо преобразование значения одного типа в значение другого типа),
- автоматическое управление памятью,
- использование прототипов,
- функции как объекты первого класса.

Инструкции – это команды, которые выполняют действия.

Инструкция – это код, представляющий собой команду.

Инструкции – исполняются.

Например, for.

Выражение – это фрагмент кода, который становится значением.

Выражение – код, выполнение которого возвращает значение.

Выражения – вычисляются

Например, $x = (2 + 3) \cdot 2 + 3$ – это выражение.

Переменная – это «именованное хранилище» для данных. Данные сохраняются в области памяти, связанной с переменной. Мы можем получить к ней доступ, используя имя переменной.

Это поименованная область памяти, адрес которой можно использовать для осуществления доступа к данным. Данные, находящиеся в переменной (то есть по данному адресу памяти), называются значением этой переменной.

Ещё переменная определяется как имя, с которым может быть связано значение, или даже как место (location) для хранения значения.

Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Переменные создаются через let или const и оператор присваивания «=».

Кеширование – это когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его.

JSON (JavaScript Object Notation) – это общий формат для представления значений и объектов. JSON является синтаксисом для сериализации объектов, массивов, чисел, строк, логических значений и значения null.

Шаблон проектирования или **паттерн** – повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

#? Какие бывают?

Синтаксический сахар – синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового.

Объект (словарь, ассоциативный массив) – абстрактный тип данных, коллекция пар «ключ-значение», где каждый ключ уникален. Это набор свойств, и каждое свойство состоит из имени и ассоциированного с этим именем значения. Свойство объекта можно понимать как переменную, закрепленную за объектом. Если значением свойства является функция, то её можно назвать методом объекта.

Метод – это то же самое свойство, только функция. Различие свойства/метода это не более чем условность.

this – ключевое слово для доступа к информации внутри объекта. Значение this – это объект «перед точкой», который использовался для вызова метода. Значение this вычисляется во время выполнения кода и зависит от контекста. Оно может использоваться в любой функции.

Задействование метода – это использование метода одного объекта в рамках (в контексте) другого объекта.

Массив – это структура данных, содержащая упорядоченный набор элементов и предоставляющая возможность произвольного доступа к своим элементам.

Псевдомассивы – объекты, имеющие индексированные свойства и length. Они также могут иметь другие свойства и методы, но у них нет встроенных методов массивов.

Метод Array.from(obj[, mapFn, thisArg]) создаёт настоящий Array из итерируемого объекта или псевдомассива.

Очередь – один из самых распространённых вариантов применения массива. Это упорядоченная коллекция элементов, поддерживающая два вида операций:

- push добавляет элемент в конец.
- shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Стек (стопка) – абстрактный тип данных, представляющий собой список элементов, организованных по принципу «последним пришёл — первым вышел» (*LIFO, last in — first out*).

Возможны три операции со стеком: добавление элемента (проталкивание, *push*), удаление элемента (*pop*) и чтение головного элемента (*peek*).

Стек вызовов (call stack) – цепочки функций, вызывающих друг друга. Стек вызовов – это очерёдность выполнения цепи запущенных программ.

Каждый внутренний вызов добавляет текущую функцию внутрь стека — и так до самой глубокой функции. Затем, когда происходит возврат, начинается раскрутка стека — из него по очереди (в обратном порядке, ведь это стек) извлекаются функции и продолжают своё выполнение с того места, где внутренняя функция вернула результат.

Двусторонняя очередь – структура данных, которая может работать и как очередь, и как стек.

Перебираемые (или итерируемые) объекты – это концепция, которая позволяет использовать такой объект в цикле for..of. Это не только массивы, но и, например, строки. Это объекты, которые реализуют метод Symbol.iterator.

Псевдомассивы – это объекты, у которых есть индексы и свойство length.

Мар – это коллекция пар ключ/значение, как и Object.

Отличие в том, что Мар позволяет использовать ключи любого типа, включая функции, объекты и примитивы. В отличие от объектов, ключи не приводятся к строкам. [MND](#).

Set – это множество: массив уникальных элементов.

Объекты Set позволяют сохранять уникальные значения любого типа: как примитивы, так и другие типы объектов.

WeakMap – это Мар-подобная коллекция пар ключ-значение, позволяющая использовать в качестве ключей только объекты, а значения могут быть произвольных типов. Пары автоматически удаляются, как только ключи (объекты) становятся недостижимы иными путями.

WeakMap не поддерживает перебор. Также, нет способа взять все ключи или значения из него, потому что соответствующих методов тоже нет.

WeakSet аналогичен Set, но добавлять в WeakSet можно только объекты (не примитивные значения).

Объект присутствует в множестве только до тех пор, пока доступен где-то ещё, или сборщик мусора удалит его. Ссылки на объекты в WeakSet являются слабыми: если на объект, хранимый в WeakSet нет ни одной внешней ссылки, то сборщик мусора удалит этот объект.

Деструктурирующее присваивание – специальный синтаксис, позволяющий извлекать части из составных данных. Можно использовать как синтаксический сахар для объявления переменных, обмена значениями и т.д.

Общее для всех языков

Программы, которые выполняют интерпретацию кода, называют рантаймом или средой исполнения или интерпретатором. Node.js – это один из многих интерпретаторов.

Среда выполнения

Среда исполнения

Рантайм

Интерпретатор – программа, которая выполняет интерпретацию кода. Например, это Node.js или браузер.

Окружение – это дополнительная функциональность (к базовым языковым), которую предоставляет каждая среда исполнения. Окружение предоставляет свои объекты и дополнительные функции. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее.

Лексическое окружение (скопировал из функций)

это структура данных, которая хранит сведения о соответствии идентификаторов и переменных.

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый **лексическим окружением** LexicalEnvironment. Он состоит из двух частей:

1. Запись окружения (environment record) — объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).
2. Ссылка на внешнее окружение (reference to the outer environment) — ссылка, позволяющая обращаться к внешнему (родительскому) лексическому окружению. То есть к тому, что соответствует коду снаружи (снаружи от текущих фигурных скобок).

Получается, что есть два лексических окружения:

Первое – локальное в **Environment Record**.

Второе – внешнее в **[[Environment]]**, ссылается на внешнее окружение, в котором функция была создана.

Классификация языков:

Высокоуровневый / низкоуровневый

Компилируемый / интерпретируемый

Высокоуровневый язык

Язык, разработанный для быстроты и удобства использования программистом. Основная черта высокоуровневых языков – абстракция, то есть введение смысловых конструкций, кратко описывающих такие структуры данных и операции над ними, описания которых на низкоуровневых языках сложны для понимания. Работают совместно с трансляторами, которые переводят языковые конструкции в низкоуровневый код.

Низкоуровневый язык

Язык, близкий к программированию непосредственно в машинных кодах. Для обозначения машинных команд обычно применяется мнемоническое обозначение. Это позволяет запоминать команды не в виде последовательности двоичных нулей и единиц, а в виде осмысленных сокращений слов человеческого языка.

Компилируемый язык

Компилятор – это программа, которая переводит текст, написанный на языке программирования, в набор машинных кодов. Соответственно, компилируемый язык программирования – это язык, код которого должен пройти стадию компиляции – быть переведённым в машинный код (инструкции целевой машины), который непонятен людям.

Интерпретируемый язык

Интерпретация — построчный анализ, обработка и выполнение исходного кода программы или запроса (в отличие от компиляции, где весь текст программы перед запуском анализируется и транслируется в машинный код, без её выполнения). Интерпретируемый язык – язык, в котором инструкции не исполняются целевой машиной, ачитываются и исполняются другой программой-интерпретатором (которая обычно написана на языке целевой машины).

Динамическая типизация — приём в языках программирования, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной.

Динамическое приведение типов – преобразование значения одного типа в значение другого типа.

Это когда операторы и функции автоматически приводят переданные им значения к нужному типу.

Из-за того, что JavaScript умеет сам изменять тип операндов, он называется языком с динамическим приведением типов, или языком со слабой типизацией.

Полифил – это эмуляция метода, который существует в современной спецификации JavaScript, но ещё не поддерживается текущим движком JavaScript.

Веб компоненты – специальные компоненты, которые не зависят от фреймворка или библиотеки и способны работать с любой из них. Похожи на инкапсулированные куски логики, которые могут быть интегрированы в любой проект. Подробнее: [Web Components](#).

API - application programming interface, программный интерфейс приложения. Это описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой. Часто реализуется отдельной программной библиотекой. API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

Функции

См. также раздел «Теория по функциям».

Функция – это фрагмент программного кода для выполнения определённых задач, к которому можно обратиться из другого места программы. Функция – способ группировки команд. Обычные значения, такие как строки или числа представляют собой *данные*. Функции можно воспринимать как «действия».

Словарь в [MDN](#)

Call-back функция – функция, передающаяся как параметр в другую функцию, которой делегирована обработка какой-то задачи.

Детерминированность – функция называется детерминированной, когда для одних и тех же входных аргументов она возвращает один и тот же результат. Например, функция, переворачивающая строку, детерминированная. Функция, возвращающая случайное число, не является детерминированной.

Побочный эффект – любые действия, изменяющие среду выполнения. Это любые файловые операции, такие как запись в файл, отправка или приём данных по сети, даже вывод в консоль или чтение файла. Кроме того, побочными эффектами считаются обращения к глобальным переменным (как на чтение, так и запись) и изменение входных аргументов в случае, когда они передаются по ссылке.

Чистая функция – это детерминированная функция без побочных эффектов.

Стек вызовов (call stack) – LIFO-стек, хранящий информацию для возврата управления из функций в программу (или подпрограмму, при вложенных или рекурсивных вызовах). При вызове подпрограммы или возникновении прерывания, в стек заносится адрес возврата — адрес в памяти следующей инструкции приостановленной программы и управление передается подпрограмме или подпрограмме-обработчику.

По простому – это структура данных, которая хранит информацию о том, где мы сейчас находимся. Как только мы начинаем выполнять функцию, мы кладём её в стек. Когда из функции выходим, она убирается из стека. Т.е. элементы размещаются только сверху.

Рекурсия – это термин в программировании, означающий вызов функцией самой себя.

Отличие рекурсии от итеративного процесса в том, что в рекурсии подвызовы складываются в стек, а в итеративный вариант использует один контекст, в котором будут последовательно меняться значения переменных и результата

Глубина рекурсии – общее количество вложенных вызовов (включая первый).

Рекурсивная структура данных (рекурсивно определяемая) – это структура, которая повторяет саму себя в своих частях. Например, деревья и HTML теги.

Область видимости (scope)

Это место, откуда мы имеем доступ к переменным или функциям. Кроме того, области видимости – это набор правил, по которым происходит поиск переменных. Сначала переменные ищутся в локальной области видимости, затем во внешней, пока не доходит до глобальной.

В JS есть 3 типа областей видимости:

Глобальная

Переменные и функции, объявленные в этой области, становятся глобальными, появляются в пространстве имён и доступны из любого места в коде.

Функциональная (локальная)

Переменные, объявленные внутри функции, доступны только внутри этой функции и всей вложенным в неё функциям. За её пределами за обращение к таким переменным получится ошибка.

Блочная

Такая область видимости находится внутри фигурных скобок так называемого блока. Например, внутри if, for и т.д. Это не относится к переменным var. Появилась в ES6.

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this и прочую служебную информацию.

Лексическое окружение — это структура данных, которая хранит сведения о соответствии идентификаторов и переменных. Здесь «идентификатор» — это имя переменной или функции, а «переменная» — это ссылка на объект или значение примитивного типа. Не путать с обычным окружением.

Понятие «лексическое окружение» или «статическое окружение» в JavaScript относится к возможности доступа к переменным, функциям и объектам на основе их расположения в исходном коде.

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый **лексическим окружением** LexicalEnvironment.

Объект лексического окружения состоит из двух частей:

1. Запись окружения (environment record) — объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).

2. Ссылка на внешнее окружение (reference to the outer environment) — ссылка, позволяющая обращаться к внешнему (родительскому) лексическому окружению. То есть к тому, что соответствует коду снаружи (снаружи от текущих фигурных скобок).

«Переменная» — это просто свойство специального внутреннего объекта: Environment Record.

Получается, что есть два лексических окружения:

Первое — локальное в Environment Record.

Второе — внешнее в [[Environment]], ссылается на внешнее окружение, в котором функция была создана.

Замыкание — запоминаение функцией части окружения, где она была задана. Функция замыкает в себе идентификаторы (все, что мы определяем) из лексической области видимости.

В JavaScript, все функции изначально являются замыканиями. Они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]] и все они могут получить доступ к внешним переменным. Есть только одно исключение: [Синтаксис "new Function"](#).

IIFE immediately-invoked function expressions — означает функцию, запускаемую сразу после её объявления. [MDN](#).

Всплытие, поднятие, hoisting — это способность функций, объявленных через function declaration, быть вызванными в коде ранее, ещё до того, как их объявили.

То же самое относится к переменной, объявленной через var. Такое объявление переменных «всплывает», но присваивания значений — нет. JavaScript "поднимает" только объявление, но не инициализацию. Если вы используете переменную, объявленную и проинициализированную после её использования, то значение будет undefined.

Это помещение в память объявлений функций до выполнения кода: объявления переменных и функций попадают в память в процессе фазы компиляции, но остаются в коде на том месте, где вы их объявили.

[MDN](#)

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. В браузере он называется window, в Node.js — global, в другой среде исполнения может называться иначе. `globalThis` был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении.

Интроспекция (type introspection) — возможность запросить тип и структуру объекта во время выполнения программы. Интроспекция может использоваться для реализации ad-hoc-полиморфизма.

Минификатор — перед отправкой JavaScript-кода на реальные работающие проекты код сжимается с помощью минификатора — специальной программы, которая уменьшает размер кода, удаляя комментарии, лишние пробелы, и, что самое главное, локальным переменным даются укороченные имена.

Декораторы — это, по сути, «обёртки», которые дают нам возможность изменить поведение функции, не изменяя её код. Это приём программирования, который позволяет взять существующую функцию и изменить/расширить её поведение.

Из википедии:

Декоратор — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Шаблон Декоратор предоставляет гибкую альтернативу практике создания подклассов с целью расширения функциональности.

Заимствование метода — использование метода одной структуры данных в контексте другой структуры. В JS можно делать с помощью методов .apply и .call.

Шаблон проектирования или паттерн — повторяемая [архитектурная конструкция](#), представляющая собой решение проблемы [проектирования](#) в рамках некоторого часто возникающего контекста.

[Прототипы, классы, ООП](#)

Прототипное наследование — это возможность языка, которая позволяет создавать одни объекты на основе других.

Унаследованные свойства – свойства, унаследованные от прототипа. В JavaScript все объекты имеют скрытое свойство `[[Prototype]]`, которое является либо другим объектом, либо `null`. Объект, на который ссылается `[[Prototype]]`, называется «прототипом». Если мы хотим прочитать свойство `obj` или вызвать метод, которого не существует у `obj`, тогда JavaScript попытается найти его в прототипе.

Класс в объектно-ориентированном программировании – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

Статические свойства – это общие свойства для всего класса. Они хранятся в конструкторе класса и доступны через конструктор класса. К ним нельзя обратиться через инстанс, а только через класс.

В классе такие свойства и методы обозначаются ключевым словом `static`. Статические методы, часто используются для создания служебных функций для приложения.

Статическими их можно назвать только в контексте того, что если использовать функции как конструкторы, то в создаваемых объектах это свойство будет отсутствовать, и доступ к ним будет осуществляться только по имени класса.

Статичный метод – это метод, который не использует личные свойства объекта. Например, если метод печатает одну и ту же строку, а не подставляет какое-нибудь свойство этого объекта.

Инкапсуляция – объединение и сокрытие от прямого обращения данных (в т.ч. методов) в рамках одной структуры. Функции называют «методами», а данные – «свойствами».

Интерфейс – набор функций (имена и их сигнатуры, то есть количество и типы входящих параметров, а также возвращаемое значение), не зависящих от конкретной реализации. Интерфейс определяет способ взаимодействия с системой.

Интерфейсные функции – функции, которые использует пользователь.

Внутренние (вспомогательные) функции – функции, которые используются исключительно внутри абстракции и к которым у пользователя нет доступа.

Примесь – это класс, методы которого предназначены для использования в других классах, но без наследования от примеси. Примесь только содержит в себе методы, которые реализуют определённое поведение. Метод, хранящиеся в примеси, используются другими объектами, чтобы добавить функциональность другим классам. Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть добавлены в прототип любого класса. Это не наследование, а просто копирование методов.

ООП – в современном мире под ООП имеют в виду полиморфизм, наследование и инкапсуляцию.

Хорошая статья на [вики](#).

Полиморфизм – способность функции обрабатывать данные разных типов. Виды полиморфизма:

- параметрический полиморфизм – исполнение одного и того же кода для разных типов данных.
- Ad-hoc-полиморфизм – обработка аргументов в зависимости от их типа, т.е. подразумевается исполнение разного кода для разных типов аргументов (например, через ветвление).

Наследование – способность объекта использовать методы родительского класса.

Инкапсуляция – объединение и сокрытие от прямого обращения данных (в т.ч. функций) в рамках одной структуры. Функции называют «методами», а данные – «свойствами».

Утиная типизация (англ. Duck typing) в ООП-языках – определение факта реализации определённого интерфейса объектом без явного указания или наследования этого интерфейса, а просто по реализации полного набора его методов.

Смысл утиной типизации – в проверке отдельных необходимых методов и свойств.

Например, мы можем проверить, что объект – массив, не вызывая `Array.isArray`, а просто уточнив наличие важного для нас метода, например `splice`.

«Если это выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка (какая разница, что это на самом деле)»

В объектно-ориентированном программировании **свойства и методы разделены на 2 группы**:

Внутренний интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса. Они не доступны для пользователя, но могут использоваться в доступных для пользователя методах в качестве вспомогательных механизмов.

Внешний интерфейс – методы и свойства, доступные снаружи класса.

В JavaScript есть **два типа полей** (свойств и методов) объекта:

Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.

Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Можно сказать, что внутренний интерфейс реализуется через приватные и защищённые поля. Для скрытия внутреннего интерфейса мы используем защищённые или приватные свойства:

Защищённые поля имеют префикс `_`. Это хорошо известное соглашение, не поддерживаемое на уровне языка. Программисты должны обращаться к полю, начинающемуся с `_`, только из его класса и классов, унаследованных от него.

Приватные поля имеют префикс `#`. JavaScript гарантирует, что мы можем получить доступ к таким полям только внутри класса. В настоящее время приватные поля не очень хорошо поддерживаются в браузерах, но можно использовать полифил.

Обработка ошибок

Исключения – это ошибки, которые возникают в синтаксически корректном коде. Их также называют «ошибками во время выполнения».

Исключение – это сигнал, указывающий на возникновение какой-либо исключительной ситуации или ошибки.

throw (возбуждение исключения)

это способ просигнализировать о такой ошибке или исключительной ситуации.

catch (перехватить исключение)

значит обработать его, т.е. предпринять действия, необходимые или подходящие для восстановления после исключения.

Асинхронное программирование

В синхронном коде каждая операция ожидает окончания предыдущей. Поэтому вся программа может зависнуть, пока одна из команд выполняется очень долго.

Асинхронные функции при своей работе не блокируют программу во время ввода-вывода информации.

Например, при копировании или чтении файлов.

Реализовать АФ можно путём использования:

1. callback функций
2. promise
3. связки async/await

Callback – это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback – функция обратного вызова).

Чтобы работать с полученной информацией, АФ всегда принимает функцию обратного вызова колбек (callback). Callback вызывается, когда операция асинхронной функции закончится.

Коллбэк – это обычная функция, которая передаётся в другую, асинхронную функцию, как аргумент. Вызвана она будет только в том случае, когда произойдёт некое событие.

Колбек имеет следующую сигнатуру: `callback(error, result)`.

Технологии

AJAX, Asynchronous Javascript and XML, асинхронный JavaScript и XML – подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с

сервером. В результате при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее. По-русски иногда произносится транслитом как «аякс».

В классической модели веб-приложения:

1. Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
2. Браузер формирует и отправляет запрос серверу.
3. В ответ сервер генерирует совершенно новую веб-страницу и отправляет её браузеру и т. д., после чего браузер полностью перезагружает всю страницу.

При использовании AJAX:

1. Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
2. Скрипт (на языке JavaScript) определяет, какая информация необходима для обновления страницы.
3. Браузер отправляет соответствующий запрос на сервер.
4. Сервер возвращает только ту часть документа, на которую пришёл запрос.
4. Скрипт вносит изменения с учётом полученной информации (без полной перезагрузки страницы)

HTML

DOM

DOM – это Document Object Model, объектная модель документа. Программный интерфейс (API), который представляет всё содержимое страницы в виде объектов, которые можно менять. Позволяет программам и скриптам получить доступ к содержимому HTML, XHTML- и XML-документов и изменять его. Точка входа – `document`.

BOM

BOM – это Browser Object Model, объектная модель браузера. Она предоставляет инструменты для работы с браузером (со всем, кроме документа). Это набор глобальных объектов, управляющих поведением именно браузера. Доступны через `window`.

API

application programming interface, программный интерфейс приложения. Это описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой. Часто реализуется отдельной программной библиотекой. API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

: псевдокласс

Псевдокласс в CSS — это ключевое слово, добавленное к селектору, которое определяет его особое состояние. Например, `:hover` может быть использован для изменения цвета кнопки при наведении курсора на неё. Также называются «Селекторы состояния элементов». [MDN](#)

:: псевдоэлемент

Псевдоэлемент в CSS — это ключевое слово, добавляемое к селектору, которое позволяет стилизовать определённую часть выбранного элемента. Например, псевдоэлемент `::first-line` может быть использован для изменения шрифта первой строки абзаца.

Список псевдоэлементов в документации. [MDN](#)

Модальное окно

Всплывающее окно поверх основного содержимого, требующее действий от пользователя.

Это небольшое окно с сообщением называется модальным окном. Понятие модальное означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK»:

```
alert("Hello");
```

`Alert`, `prompt` и `confirm` – все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

Тормозят ли они таймер#?

Fetch

Современный метод для выполнения AJAX запросов. Именно с помощью `fetch` происходит общение с сервером и другими сайтами.

Вводные данные

Типы данных (кратко)

Типизация – это классификация информации.

Тип данных – это конкретный класс информации. Например, `String` в JavaScript — это тип данных.

Понятие «тип данных» всегда привязано к конкретному языку и может быть чем угодно в зависимости от предпочтений разработчиков языка.

Сложное определение:

Тип данных – это множество значений и операций над этими значениями. Это возможные значения, характеристики и набор операций для некоторого атрибута.

Множество – это совокупность объектов, которые называются элементами этого множества, обладающих общим для всех их характеристическим свойством.

Есть восемь основных типов данных в JavaScript, о них подробнее написано ниже.

1. Number
2. BigInt
3. String
4. Boolean (логический тип)
5. Значение «null»
6. Значение «undefined»
7. Object
8. Symbol

Структура данных

Структура данных – это конкретный способ хранения и организации данных.

Использование структуры данных, подходящей под решаемую задачу, позволяет упростить код, устранивая запутанную логику.

Некоторые структуры данных:

1. Массив
2. Стек
3. Очередь
4. Связанные списки
5. Графы
6. Деревья
7. Префиксные деревья
8. Хэш таблицы
9. Куча
- 10.Матрица

Инструкции

Инструкции – это синтаксические конструкции и команды, которые выполняют действия.

Инструкция – это код, представляющий собой команду.

Инструкции – исполняются.

К инструкциям относятся:

```
for  
while  
break  
return  
if  
и другие
```

Обычно каждую инструкцию пишут на новой строке, чтобы код было легче читать и завершают точкой с запятой. Иногда JS автоматом проставляет точку с запятой, но это не всегда работает.

Некоторые длинные выражения можно переносить на другую строку для удобности, точка с запятой не нужна и JS её ставить не будет. Например, при использовании .map .filter и т.д.

Выражения

О выражениях и операторах можно подробнее прочитать здесь: [выражения и операторы](#).

Выражение – это фрагмент кода, который становится значением.

Например, $x = (2 + 3)$. $2 + 3$ – это выражение.

Выражение – код, выполнение которого возвращает значение.

Выражения – вычисляются

К выражениям относятся:

- Вызов функции
- Арифметические и логические операции
- Тернарный оператор
- и другие

Комментарии

```
// Однострочный комментарий.  
  
/*  
многострочный  
комментарий.  
*/
```

В большинстве редакторов строку кода можно закомментировать, нажав комбинацию клавиш **Ctrl+/** для однострочного комментария и что-то вроде **Ctrl+Shift+/** – для многострочных комментариев.

Вложенные комментарии в JS не поддерживаются.

Строгий режим - use strict

[MDN](#)

Директива выглядит как строка: "use strict". Её надо прописывать в самом начале документа или функции. иначе строгий режим может не включиться. Над "use strict" могут быть записаны только комментарии. Нет никакого способа отменить use strict.

Директива "use strict" переключает движок в «современный» режим, изменяя поведение некоторых встроенных функций.

Некоторые конструкции языка (например, классы) включают строгий режим по умолчанию.

Что меняется?

Во-первых, строгий режим делает невозможным **случайное создание глобальных переменных**. Т.е. нельзя что-то присвоить в переменную, пока она не объявлена через let, const или var. В обычном JavaScript во время такого присваивания создаётся новое свойство глобального объекта и выполнение продолжается. Присваивания, которые могут случайно создать глобальную переменную, в строгом режиме выбрасывают исключение:

```
"use strict";
```

```
// Предполагая, что не существует глобальной переменной  
mistypeVaraible = 17; // mistypedVaraible, эта строка выбросит ReferenceError
```

Во-вторых, строгий режим влечёт ошибку, если что-то присваивается в **«запрещённое» имя** переменной. Например, нельзя что-то присваивать так:

```
// Присваивание значения глобальной переменной, защищенной от записи  
var undefined = 5; // выдаст TypeError  
var Infinity = 5; // выдаст TypeError  
  
// Задание нового свойства нерасширяемому объекту  
var fixed = {};  
Object.preventExtensions(fixed);  
fixed.newProp = "ohai"; // выдаст TypeError
```

В-третьих, в строгом режиме попытки **удалить неудаляемые свойства** будут вызывать исключения (в то время как прежде такая попытка просто не имела бы эффекта):

```
"use strict";  
delete Object.prototype; // выдаст TypeError
```

В-пятых, строгий режим требует, чтобы **имена аргументов** в объявлении функций встречались **только один раз**. В обычном коде последний повторённый аргумент скрывает предыдущие аргументы с таким же именем.

```
function sum(a, a, c) { // !!! синтаксическая ошибка  
  "use strict";  
  return a + a + c; // ошибка, если код был запущен  
}
```

...и другие штуки. Смотри в [MDN](#).

Сборка мусора

Учебник [тут](#).

В интерпретаторе JavaScript есть фоновый процесс, который называется [сборщик мусора](#). Он следит за всеми объектами и удаляет те, которые стали недостижимы.

Основной концепцией управления памятью в JavaScript является принцип *доступности*.

1. Существует **базовое множество** достижимых значений, которые не могут быть удалены.

Например:

- Локальные переменные и параметры текущей функции.
- Переменные и параметры других функций в текущей цепочке вложенных вызовов.
- Глобальные переменные.
- (некоторые другие внутренние значения)

Эти значения мы будем называть **корнями**.

2. Любое другое значение считается достижимым, если оно **доступно из корня по ссылке или по цепочке ссылок**.

Например, если в локальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы.

Взаимодействие: alert, prompt, confirm

Это функции интерфейса браузера.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

alert

Показывает сообщение. Отобразит окно в браузере и приостановит дальнейшее выполнение скриптов до тех пор, пока пользователь не нажмёт кнопку «OK».

Это окно с сообщением называется *модальным окном*. Понятие *модальное* означает, что пользователь не может взаимодействовать с интерфейсом остальной страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

```
alert(message);
```

prompt

Показывает сообщение и запрашивает ввод текста от пользователя. Возвращает напечатанный текст в поле ввода или null, если была нажата кнопка «Отмена» или Esc с клавиатуры.

Функция prompt принимает два аргумента:

```
result = prompt(title, [default]);
```

title

Текст для отображения в окне.

default

Необязательный второй параметр, устанавливает начальное значение в поле для текста в окне.

Отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

Пользователь может напечатать что-либо в поле ввода и нажать OK. Он также может отменить ввод нажатием на кнопку «Отмена» или нажав на клавишу Esc.

Вызов prompt вернёт текст, указанный в поле для ввода, или null, если ввод отменён пользователем.

Для IE: всегда устанавливайте значение по умолчанию

Второй параметр является необязательным, но если не указать его, то Internet Explorer установит значение "undefined" в поле для ввода.

```
let test = prompt("Test", ''); // <-- для IE
```

confirm

Показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена.

Если нажата кнопка OK, то вернёт true.

Если нажата кнопка «Отмена» или Esc, то вернёт false.

```
result = confirm(question);
```

Например:

```
let isBoss = confirm("Ты здесь главный?");  
alert( isBoss ); // true, если нажата OK
```

Полифили и babel

Когда мы используем современные возможности JavaScript, некоторые движки могут не поддерживать их.

[Babel](#) – это [транспилер](#). Он переписывает современный JavaScript-код в предыдущий стандарт.

У Babel две части:

1. [транспилер](#), который переписывает код в старый стандарт и после этого код отправляется на сайт.

Современные сборщики проектов типа [webpack](#) или [brunch](#) предоставляют возможность запускать транспилер автоматически после каждого изменения кода, что позволяет экономить время.

2. полифил. Термин «полифил» означает, что скрипт «заполняет» пробелы и добавляет современные функции.

Два интересных хранилища полифилов:

[core.js](#) поддерживает много функций, можно подключать только нужные.

[polyfill.io](#) – сервис, который автоматически создаёт скрипт с полифилом в зависимости от необходимых функций и браузера пользователя.

Таким образом, чтобы современные функции поддерживались в старых движках, нам надо установить транспилер и добавить полифил.

В больших проектах лучше использовать сборщик проектов + транспайлер (например, webpack + babel), для автоматического конвентирования файлов, т.к. это будет быстрее, чем вручную копировать полифили: не надо смотреть поддерживаемые фичи браузера, не надо искать нужные полифили.

Стандартные встроенные объекты

Глобальные функции

[eval\(\)](#)

[uneval\(\)](#)

[isFinite\(\)](#)

[isNaN\(\)](#)

[parseFloat\(\)](#)

[parseInt\(\)](#)

[decodeURI\(\)](#)

[decodeURIComponent\(\)](#)

[encodeURI\(\)](#)

[encodeURIComponent\(\)](#)

Числа и даты

[Number](#)

[Math](#)

[Date](#)

Обработка текста

Объекты для манипулирования текстом.

[String](#)

[RegExp](#)

Структурированные данные

Буферы данных и Объектная нотация JavaScript.

[ArrayBuffer \(en-US\)](#)

[DataView \(en-US\)](#)

[JSON](#)

Объекты управляющих абстракций

[Promise](#)

Прочее

[Аргументы функции \(arguments\)](#)

</>

Всякая хуйня, которую я забываю

[isNaN](#)

Функция определяет, является ли литерал или переменная нечисловым значением.

Значение NaN не равно ничему, в т.ч. самому себе.

```
isNaN(NaN)  
// true
```

Number.isNaN()

Метод определяет, является ли переданное значение NaN. Это более надёжная версия оригинальной глобальной функции isNaN()

```
Number.isNaN(value)
```

В отличие от глобальной функции isNaN(), не имеет проблемы принудительного преобразования параметра в число. Это значит, что в него безопасно передавать значения, которые обычно превращаются в NaN, но на самом деле NaN не являются.

Object.is()

Определяет, являются ли два значения одинаковыми значениями. Ведёт себя так же, как и тройное равно, но со специальной обработкой:

- работает с NaN
- 0 и -0 не равны друг другу

Как обычно два отдельных объекта не равны.

```
Object.is(value1, value2);
```

Array.isArray()

Метод возвращает true, если объект является массивом и false, если он массивом не является.

```
Array.isArray(obj)
```

isFinite()

Глобальная функция определяет, является ли переданное значение конечным числом. Предварительно приводит значение к числу.

NaN, положительной или отрицательной бесконечностью, метод вернёт false, иначе возвращается true.

```
isFinite(0);  
isFinite("0"); // true
```

Number.isFinite()

Метод Number.isFinite() определяет, является ли переданное значение конечным числом.

В отличии от глобальной функции isFinite(), этот метод принудительно не преобразует параметр в число. Это означает, что он возвращает true только для конечных значений числового типа.

```
isFinite(0); // true  
isFinite("0"); // false
```

parseInt()

Принимает строку в качестве аргумента и возвращает целое число.

```
parseInt(string, radix);
```

radix – система счисления в диапазоне между 2 и 36.

Всегда указывайте этот параметр.

parseFloat()

принимает строку в качестве аргумента и возвращает число с плавающей точкой.

```
parseFloat(строка)
```

in operator

возвращает true, если свойство содержится в указанном объекте или в его цепочке прототипов.

```
prop in object
```

</>

Переменные

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

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

Объявление переменных

Переменная создаётся (определяется, объявляется) с помощью инструкций:

let – собственно, переменная;

const – константа, которую нельзя изменить, попытка сделать это приведёт к ошибке;

var – старый способ, подробнее написано дальше.

Можно объявить несколько переменных в одной строке через запятую:

```
// можно, но не рекомендуется
let user = 'John', age = 25, message = 'Hello';

// вот так норм
let user = 'John',
    age = 25,
    message = 'Hello';
```

Область видимости

Простыми словами, **область видимости** – это зона доступности переменных.

Виды:

Глобальная

Переменные и функции, объявленные в этой области, доступны из любого места в коде.

Функциональная (локальная)

Переменные, объявленные внутри функции, доступны только внутри этой функции и всей вложенным в неё функциям. За её пределами за обращение к таким переменным получится ошибка.

Блочная

Такая область видимости находится внутри фигурных скобок блока. Например, внутри if, for и т.д. Это не относится к переменным var. Появилась в ES6.

Кроме того, области видимости – это набор правил, по которым происходит поиск переменных. Сначала переменные ищутся в локальной области видимости, затем во внешней, пока не доходит до глобальной.

Область видимости цикла for

Любые переменные (объявленные с помощью const или let), созданные при инициализации (в круглых скобках) цикла имеют локальную область видимости: они видны только в пределах цикла и существуют только в период выполнения цикла. После завершения цикла эти локальные переменные удаляются из памяти. Попытка обратиться за пределами цикла к локальным переменным цикла вызовет ошибку ReferenceError.

Область видимости блока if.. else

Переменные в этих блоках локальные.

В примере ниже в блоках объявляются константы, которые нельзя вызвать вне этих блоков:

```
if ('condition') {  
    const a = true;  
} else {  
    const a = false;  
}  
  
a;  
// ReferenceError: a is not defined
```

Область видимости функций

Переменные, объявленные внутри функции, видны только внутри этой функции.

В то же время, у функции есть доступ к внешним переменным. Внешняя переменная используется, только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Внешняя будет проигнорирована.

Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.

Ограничения на имена

В JavaScript есть ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы \$ и _.
2. Первый символ не должен быть цифрой.
3. Нельзя использовать дефис
4. Нельзя использовать зарезервированные слова

Существует [список зарезервированных слов](#), которые нельзя использовать в качестве имён переменных, потому что они используются самим языком. Например: let, class, return и function зарезервированы.

Регистр имеет значение.

Нелатинские буквы разрешены, но их не рекомендуются использовать.

var, особенности

Существует 2 основных отличия var от let/const:

var становится свойством глобального объекта

К ним можно обращаться через window.

var не имеют блочной (локальной) области видимости, они всегда в глобальной. Var выходит за пределы блоков if, for и подобных. Это происходит потому, что на заре развития JavaScript блоки кода не имели лексического окружения. Исключения – они «блочные» в теле функции, либо, если переменная глобальная, в скрипте.

Эти особенности, как правило, не очень хорошо влияют на код. Блочная область видимости – это удобно. Поэтому много лет назад let и const были введены в стандарт и сейчас являются основным способом объявления переменных.

var игнорирует блоки типа if, for и мы получили глобальную переменную:

```
if (true) {  
    var test = true;  
}  
console.log(test); // true, переменная существует вне блока if
```

Аналогично для циклов:

```
for (var i = 0; i < 10; i++) {  
    // ...  
}  
alert(i); // 10, переменная i доступна вне цикла, т.к. является глобальной переменной
```

var в функциях не видны вне этих функций:

```
function foo() {  
    var a = 123;  
}  
  
console.log(a);  
// ReferenceError: a is not defined
```

var всплывает

Т.е. обрабатываются в начале запуска функции или скрипта. Важный момент: переменная всплынет, но её значение будет undefined. Всплывают только сами объявления переменных, но не их присвоенные значения. В ситуации с let или const произошла бы ошибка.

```
console.log(a);  
// undefined  
  
var a = 123;
```

Типы данных (подробно)

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число. Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

Динамическая типизация — приём в языках программирования, при котором переменная связывается с типом в момент присваивания значения, а не в момент объявления переменной.

Каждому типу данных принадлежит свой диапазон значений. Для number это 1, 2, NaN, infinity, для Bool – true, false и так далее.

Типы данных в JS

Есть восемь основных типов данных в JavaScript:

1. Number

2. BigInt
3. String
4. Boolean (логический тип)
5. Значение «null»
6. Значение «undefined»
7. Object
8. Symbol

1 – Number

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой. Существует множество операций для чисел: умножение *, деление /, сложение +, вычитание - и так далее.

Кроме обычных чисел, существуют «специальные числовые значения»: Infinity, -Infinity и NaN.

Специальные числовые значения относятся к типу «число», хотя это не числа в привычном значении этого слова.

Infinity представляет собой математическую бесконечность. Это особое значение, которое больше любого числа. Мы можем получить его в результате деления на ноль или задать его явно: let a = Infinity.

NaN

Означает вычислительную ошибку. Это результат неправильной или неопределенной математической операции, например при делении строки на 2. Любая математическая операция, в которой участвует NaN возвращает NaN. Математические операции в JavaScript «безопасны». Скрипт никогда не остановится с фатальной ошибкой, в худшем случае мы получим NaN как результат выполнения.

NaN не равно ничему, в т.ч. самому себе. Для проверки можно использовать:

```
isNaN(значение)  
Object.is(NaN, NaN)
```

2 – BigInt

В JavaScript тип «number» не может содержать числа больше, чем 2^{53} (или меньше, чем -2^{53} для отрицательных). Это техническое ограничение вызвано их внутренним представлением. 2^{53} – это достаточно большое число, состоящее из 16 цифр, поэтому чаще всего проблем не возникает. Но иногда нам нужны действительно гигантские числа, например в криптографии или при использовании метки времени («timestamp») с микросекундами.

Тип BigInt был добавлен в JavaScript, чтобы дать возможность работать с целыми числами произвольной длины. Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

```
const bigInt = 1234567890123456789012345678901234567890n;
```

3 – String

Строка (string) в JavaScript должна быть заключена в кавычки. В JavaScript существует три типа кавычек.

1. Двойные: "Привет".
2. Одинарные: 'Привет'.
3. Обратные: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные кавычки же имеют «расширенную функциональность». Они позволяют встраивать выражения в строку, заключая их в \${...}. Это называется «интерполяция».

4 – Boolean (логический тип)

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь). Такой тип, как правило, используется для хранения значений да/нет.

5 – Значение «null»

Специальное значение null не относится ни к одному из типов, описанных выше.

Оно формирует отдельный тип, который содержит только значение: null.

В JavaScript null – специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

Можно присваивать его переменной, значение которой пока неизвестно, чтобы застолбить её имя на будущее.

Не путать с неприсвоенным значением!

typeof (null) => object (по историческим причинам).

6 – Значение «undefined»

Тип данных undefined также состоит только из одного значения: undefined. Оно формирует тип из самого себя также, как и null.

Оно означает, что «значение не было присвоено». Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined.

Технически мы можем присвоить значение undefined любой переменной, но так делать не рекомендуется.

Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined – для проверок, была ли переменная назначена.

7 – Object

Тип object – особенный.

Все остальные типы называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё). Объекты же используются для хранения коллекций данных или более сложных объектов.

8 – Symbol

Тип symbol (символ) используется для создания уникальных идентификаторов объектов.

typeof

Оператор «typeof» позволяет определить тип передаваемого операнда. Название типа возвращается в виде строки. У него есть два синтаксиса:

Синтаксис оператора
typeof x

Синтаксис функции
typeof(x)

```
typeof undefined // "undefined"
typeof 0          // "number"
typeof 10n       // "bigint"
typeof true      // "boolean"
typeof "foo"     // "string"
typeof Symbol("id") // "symbol"
typeof null      // "object"  (1)
typeof alert      // "function" (2)
```

Последние три строки нуждаются в пояснении:

1. Результатом вызова typeof null является "object". Это неверно. Это официально признанная ошибка в typeof, сохранённая для совместимости. Конечно, null не является объектом. Это специальное значение с отдельным типом. Повторимся, это ошибка в языке.
2. Вызов typeof alert возвращает "function", потому что alert является функцией. Мы изучим функции в следующих главах, где заодно увидим, что в JavaScript нет специального типа «функция». Функции относятся к

объектному типу. Но `typeof` обрабатывает их особым образом, возвращая "function". Формально это неверно, но очень удобно на практике.

instanceof

Оператор `instanceof` проверяет, принадлежит ли объект к определённому классу. Другими словами, «`object instanceof constructor`» проверяет, присутствует ли объект `constructor.prototype` в цепочке прототипов `object`.

`Instanceof` проверяет принадлежность к конструктору (к структуре данных).

`Typeof` проверяет принадлежность к типу данных.

Преобразование типов

Динамическое приведение типов – преобразование значения одного типа в значение другого типа.

Это когда операторы и функции автоматически приводят переданные им значения к нужному типу.

Из-за того, что JavaScript умеет сам изменять тип операндов, он называется языком с динамическим приведением типов, или языком со слабой типизацией.

Если сложить число и строку, js преобразует число в строку и осуществит конкатенацию.

При умножении js будет преобразовывать строку к числу и умножать числа. Если строка не является числом (является буквами), то в результате будет `Nan`.

Здесь написано только про примитивы. Про преобразование объектов см. раздел про объекты.

Хорошая задача на проверку преобразований [тут](#).

Строковое преобразование

Строковое преобразование – это представление чего-либо в виде строки. Некоторые функции делают это автоматом. Например, `alert(value)` преобразует значение к строке. При этом `console.log` к строке не преобразует.

`String(value)`

Функция выполняет явное преобразование значения к строке.

Преобразование у примитивов происходит очевидным образом: `false` становится "false", `null` становится "null" и т.д.

У объектов немного иначе ([дописать](#))

При сложении чего-либо со строкой, это становится строкой. Исключение – унарный `«+»` перед строкой.

Пример посложнее:

```
alert(2 + 2 + '1'); // будет "41", а не "221"
```

Сложение и преобразование строк — это особенность бинарного плюса `+`. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа.

Несколько хитрых задач:

```
" -9" + 5
// " -95"
// любое сложение со строкой превращает число 5 в строку и добавляет к строке.
```

```
" -9" - 5
// -14

" \t \n" - 2
// -2
// пробельные символы, такие как \t и \n, по краям строки игнорируются при преобразовании в число. Это аналогично пустой строке, а пустая строка '' = 0
```

Численное преобразование

Численное преобразование происходит в математических функциях и выражениях.

Явно преобразовать к числу можно с помощью функции Number(x) или +унарного плюса.

Number(value)

Функция явно преобразовывает value к числу:

```
let str = "123";
let num = Number(str); // становится числом 123
console.log(typeof num); // number
```

Правила численного преобразования:

```
Number(NaN);           // NaN
Number(undefined);     // NaN
Number(null);          // 0

Number('');            // 0
Number('str');         // NaN
Number('19');          // 19
Number(Infinity);     // Infinity

Number(true);          // 0
Number(false);         // 1

Number({});            // NaN
Number([]);             // 0

Number([3]);            // 3
Number([1, 2]);         // NaN
```

Бинарный +

Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам и конкатенируются (присоединяются) друг к другу.

```
console.log( 2 * 3 + '2');
// "62", число преобразуется к строке

alert(2 + 2 + '1' );
// "41", а не "221"
```

Унарный +

Всё преобразует к числу, в т.ч. булевые типы. Это равносильно использованию функции преобразования Number() (о ней ниже).

```
const a = '123';
console.log(typeof +a);
// number

let a = 'abc';
console.log(+a)           // NaN
console.log(typeof(+a))   // number

console.log(+true);
console.log(+false);
// 1, 0

console.log( +{} );
// NaN

console.log( +[1, 2, 3] );
```

```
// NaN
```

Остальные математические операторы

Почти все математические операторы, кроме сложения, выполняют преобразование данных к типу number. Они работают только с числами и всегда преобразуют операнды в числа:

```
alert( "6" / "2" );
// 3, Строки преобразуются в числа

alert( 6 - '2' );
// 4, '2' приводится к числу
```

Неожиданные приколы численного преобразования только **со строкой**:

Пустая строка преобразуется в 0.

Если в строке только пробелы и цифры подряд – преобразуется в соответствующее число.

Если есть пробелы между цифрами – NaN.

Если есть буквы между цифрами – NaN.

Массивы реализуют только преобразование `toString`.

Они не имеют ни `Symbol.toPrimitive`, ни функционирующего `valueOf`.

Таким образом, [] становится пустой строкой, [1] становится "1",

а [1, 2] становится "1,2" – нет, NaN.

```
console.log( [] + 1 );      // "1"
console.log( [1] + 1 );     // "11"
console.log( [1,2] + 1 );   // "1,21"
```

Поэтому в примере выше массив с одним элементом преобразуется в число.

Логическое преобразование

Самое простое. Происходит в логических операторах, но также может быть выполнено явно:

- с помощью функции `Boolean(value)`
- двойным отрицанием «!!»

```
// функция Boolean()
Boolean("string"); // true
Boolean(null);    // false

// двойное отрицание
!!"string";       // true
!!null;           // false
```

Правило преобразования

Есть несколько значений, которые называют ложными

к `false` преобразуются:
0
пустая строка ""
null
`undefined`
`Nan`

Все остальные значения становятся `true`.
Важно: строчка с нулём "0" – это `true`.

Преобразование объектов (кратко)

Подробно о преобразовании объектов написано в разделе про объекты.

Бывают операции, при которых объект должен быть преобразован в примитив. Например:

Строковое преобразование – если объект выводится через `alert(obj)`.

Численное преобразование – при арифметических операциях и при сравнении с примитивом.

Логическое преобразование – при `if(obj)` и других логических операциях.

Алгоритм преобразований объектов к примитивам следующий. В процессе преобразования движок JavaScript пытается найти и вызвать три следующих метода объекта:

1. Вызывает `obj[Symbol.toPrimitive](hint)` – метод с символьным ключом `Symbol.toPrimitive` (системный символ), если такой метод существует, и передаёт ему хинт.

2. Иначе:

- Если хинт равен "number" или "default", пытается вызвать `obj.valueOf()`, а если его нет, то `obj.toString()`, если он существует.

- Если хинт равен "string", пытается вызвать `obj.toString()`, а если его нет, то `obj.valueOf()`, если он существует.

На практике довольно часто достаточно реализовать только `obj.toString()` как «универсальный» метод для всех типов преобразований, возвращающий «читаемое» представление объекта, достаточное для логирования или отладки.

Нет обязательного требования, чтобы `toString()` возвращал именно строку, или чтобы метод `Symbol.toPrimitive` возвращал именно число для хинта «number». Единственное обязательное требование: эти методы должны возвращать примитив, а не объект. По историческим причинам, если `toString` или `valueOf` вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало).

Метод `Symbol.toPrimitive`, напротив, обязан возвращать примитив, иначе будет ошибка.

Операторы в JS

Подробнее о выражениях и операторах можно прочитать в документации тут: [Выражения и операторы](#). По ссылке описаны и математические, и унарные операторы, `instanceof`, `yeld`, `in` и ещё куча всего.

Операторы всегда возвращают какое-то значение. Например, операторы сравнения возвращают булевые значения.

Приоритет операторов

Приоритет

Приоритет операторов определяет порядок, в котором операторы выполняются. Операторы с более высоким приоритетом выполняются первыми.

Ассоциативность

Ассоциативность определяет порядок, в котором обрабатываются операторы с одинаковым приоритетом

Левая ассоциативность – операторы с одинаковым приоритетом выполняются слева направо.

Операторы присваивания являются право-ассоциативными:

```
a = b = 5;
```

Вот [таблица](#) приоритета операторов.

Нет необходимости всё запоминать, обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных.

Приоритет	Тип оператора	Ассоциативность	Оператор
20	группировка	не определено	(...)
17	Постфиксный инкремент	не определено	... ++
	Постфиксный декремент		... --

16	Логическое отрицание	правая	! ...
	Унарный +		+ ...
	Префиксный инкремент		++ ...
	Префиксный декремент		-- ...
15	Возведение в степень	правая	... ** ...
14	Умножение	левая	
	Деление		
	Остаток		
13	Сложение	левая	
	Вычитание		
11	Меньше	левая	
	Меньше или равно		
	Больше		
	Больше или равно		
10	Равно	левая	... == ...
	Не равно		... != ...
	Строго равно		... === ...
	Строго не равно		... !== ...
6	Логическое «И»	левая	... && ...
5	Логическое «ИЛИ»	левая
	Нулевое слияние		... ?? ...
4	Условный	правая	... ? ... : ...
3	Присваивание	правая	=
1	Запятая / Последовательность	левая	... , ...

Операторы математические

+ - сложение, вычитание
 * умножение
 / деление
 % остаток от деления
 ** возведение в степень

Операнд – объект, который участвует в операции.

Унарные операции требуют только один операнд. Например, -3.

Бинарные операции требуют наличие двух операндов. Если пропустить хотя бы один (например «3 +), то программа завершится с синтаксической ошибкой.

Тернарные – с тремя операндами. Причем, операторы могут выглядеть одинаково, но обозначать разные операции.

Коммутативность – свойство операции, когда изменения порядка operandов не влияет на результат. Бинарная операция считается коммутативной, если поменяв местами operandы получается тот же самый результат: $(3 + 2) = (2 + 3)$.

Инкремент и декремент

Это унарные операции, увеличивающие или уменьшающие переменную на 1: «`n++`», «`n--`».

У них 2 формы (в зависимости от того, ставятся они до переменной или после): `++`префиксная и постфиксная`++`.

Префиксная нотация:

сначала происходит изменение переменной, а потом её возврат.

Постфиксная нотация наоборот:

сначала возврат (старого значения), а потом изменение переменной.

```
let a = 1;
console.log(a++); // 1
console.log(a); // 2

let b = 1;
console.log(++b); // 2
console.log(b); // 2
```

Важно:

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа `5++`, приведёт к ошибке.

Операторы `++/--` могут также использоваться внутри выражений. Их приоритет выше, чем у арифметических операций. Лучше так не делать и использовать стиль «одна строка – одно действие».

```
let counter = 1;
alert( 2 * ++counter ); // 4

let counter = 1;
alert( 2 * counter++ ); // 2, потому что counter++ возвращает "старое" значение
```

Дополнительно про бинарный и унарный «+» написано выше в «численном преобразовании»

Конкатенация, бинарный +

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет. Это называется конкатенацией.

Если хотя бы один из operandов является строкой, то второй будет также преобразован к строке.

Тем не менее, помните, что операции выполняются слева направо. Если перед строкой идут два числа, то числа будут сложены перед преобразованием в строку:

```
alert(2 + 2 + '1' ); // будет "41", а не "221"
```

Сложение и преобразование строк – это особенность бинарного плюса +.

Другие арифметические операторы работают только с числами и всегда преобразуют operandы в числа.

Например, вычитание и деление:

```
alert( 2 - '1' ); // 1
alert( '6' / '2' ); // 3
```

Унарный +

Унарный, то есть применённый к одному значению, плюс + ничего не делает с числами. Но если operand не число, унарный плюс преобразует его в число. Это то же самое, что и `Number()`, только короче.

Присваивание, =

Операторы присваивания присваивают значение своему левому операнду, зависящее от значения их правого операнда. Таким образом, возможно присваивание по цепочке:

```
let a, b, c;  
a = b = c = 2 + 2;
```

Такое присваивание работает справа-налево. Сначала вычисляется самое правое выражение $2 + 2$, и затем оно присваивается переменным слева: c , b и a . В конце у всех переменных будет одно значение.

Оператор " $=$ " возвращает значение.

Все операторы возвращают значение. Вызов $x = value$ записывает $value$ в x и возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения, но писать самим в таком стиле не рекомендуется:

```
let a = 1;  
let b = 2;  
  
let c = 3 - (a = b + 1);  
  
alert( a ); // 3  
alert( c ); // 0
```

Деструктурирующее присваивание

Подробнее в разделе про дестракчеринг.

Деструктурирующее присваивание позволяет присваивать свойства массива или объекта переменным с использованием синтаксиса, похожего на объявление литералов массива или объекта:

```
[a, b] = [1, 2]  
{a, b} = {a:1, b:2}
```

Операторы сравнения и равенства

Документация [здесь](#), тут есть, что полезного почитать.

Операторы сравнения всегда возвращают логическое значение: `true` или `false`.

Некоторые особенности сравнения кратко (подробнее о них написано ниже):

- `NaN` не равно ничему, в том числе и `NaN`.
 - `null` и `undefined` равны между собой только в абстрактном (не строгом) сравнении.
 - нули с положительным и отрицательным знаком равны.
 - сравнение объекта истинно лишь в том случае, если оба операнда ссылаются на один и тот же объект в памяти.
- Все приколы происходят с операторами равенства и при сравнении разных типов данных между собой.

```
NaN === NaN // false  
  
null == undefined // true  
null === undefined // false  
  
-0 === 0 // true
```

С `NaN` корректно работают:

```
Object.is(NaN, NaN) => true  
isNaN(NaN) => true
```

Равенство и алгоритмы сравнения

Строгое равенство

Оператор строгого равенства (`==`) проверяет равенство и значения, и типа данных операндов. Если `a` и `b` принадлежат к разным типам данных, то результат автоматически `false`.

Оператор «строго не равно» (`!=`) возвращает `true`, если операнды принадлежат к разным типам или не равны.

Такой алгоритм сравнения называется Строгий Алгоритм Эквивалентного Сравнения.

```
1 === 1      // true
1 === '1'    // false
0 === false  // false
```

Нестрогое равенство

Оператор нестрогого равенства (`==`) сначала приводит операнды к одному типу и только затем применяет строгое сравнение.

Оператор строго не равно (`!=`) сначала приводит операнды к одному типу и возвращает `true` в том случае, если они не равны по значению.

Такой алгоритм сравнения называется [Абстрактный Алгоритм Эквивалентного Сравнения](#).

```
1 == 1      // истина
"1" == 1    // истина
0 == false  // истина
```

Операторы «больше, меньше» (и равенства тоже) возвращают логическое значение: `true` или `false`.

С ними всё понятно, смысла останавливаться нет.

```
Больше >
Больше или равно >=
Меньше(<)
Меньше или равно (<=)
```

Сравнение объектов

Если оба операнда являются объектами, то JavaScript сравнивает внутренние ссылки, которые равны в том случае, если они ссылаются на один и тот же объект в памяти.

```
const a = {};
const b = {};
console.log(a == b);
// false

const a = {};
const b = a;
console.log(a == b);
// true
```

Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок сравнения.

Для сравнения строк используется кодировка Unicode, а не настоящий алфавит. Поэтому заглавные буквы (большие, типа "A") меньше, чем строчные (маленькие, типа "a"), потому что строчные буквы имеют больший код в таблице Unicode. О таблице Unicode больше написано в файле про типы данных.

Алгоритм сравнения строк:

- Сравниваются первые символы. Чей первый символ больше, та строка и больше.
- Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
- Большой считается более длинная строка. Если обе строки заканчиваются одновременно, то они равны.

```
console.log('A' < 'a');
// true

console.log('abc' === 'abc');
// true

console.log('a' < 'abc');
// true
```

Сравнение разных типов между собой

При сравнении разных типов, они преобразуются к числу.

Например, логическое значение true становится 1, а false – 0.

Это не работает с null и undefined, о них написано ниже.

```
string и number:
Number(string)

что-то и Boolean:
Number(Boolean)

Object и number или Object и string:
valueOf() и toString()
```

На практике:

```
alert( '2' > 1 );      // true, строка '2' становится числом 2
alert( '01' == 1 );    // true, строка '01' становится числом 1

alert( true == 1 );   // true
alert( false == 0 );  // true
```

Сравнение с null и undefined

Поведение null и undefined особое.

```
console.log(null === undefined); // false
console.log(null == undefined); // true

console.log(Number(null))       // 0
console.log(Number(undefined)) // NaN
```

Тут небольшая хуйня.

При строгом равенстве они очевидно неравны друг другу, потому что это два разных типа данных.

При нестрогом сравнении они равны. По идеи, этого не должно происходить, потому что они сначала должны преобразоваться к number, и Number(null) – это 0, а Number(undefined) – это NaN. Тем не менее, они равны при нестрогом сравнении и это специальное правило языка.

На этом шизофрения не заканчивается:

```
console.log( Number(null)); // 0

console.log( null >= 0 );  // true
console.log( null > 0 );  // false
console.log( null == 0 ); // false
```

Причина в том, что нестрогое равенство и сравнения $>$ $<$ $>=$ $<=$ работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому $null >= 0$ истинно, а $null > 0$ ложно. С другой стороны, для нестрогого равенства $==$ значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому $null == 0$ ложно.

Операторы логические

В учебнике learn.js много интересного написано по этой теме [здесь](#). С этими операторами много приколов на практике, постарался выписать всё интересное из учебника.

В JavaScript есть три логических оператора со следующим приоритетом:

!	не (отрицание)	16
&&	и (конъюнкция)	6
	или (дизъюнкция)	5

Важно:

- при использовании логических операторов, каждый operand конвертируется в логическое значение.
- у логических операторов «и» и «или» минимум два, а максимум сколько угодно operandов.
- в js нельзя записать операцию таким образом: $8 < x < 20$. Нужно разбить его на два логических выражения и соединить логическим оператором: $(8 < x) \&\& (x < 20)$
- приоритет оператора $\&\&$ больше, чем у $||$
- важные задачи на эту тему [здесь](#). Они ёбнутые, надо их иногда пересматривать.

Обо всех кратко

$||$ - возвращает первый true-operand или любой последний

$\&\&$ - возвращает первый false-operand или любой последний

$!$ – не, унарный оператор. Конвертирует в булево значение и возвращает «обратное» булево значение.

Оператор «или»

Оператор «или» возвращает первый истинный operand, либо просто любой последний:

```
result = value1 || value2 || value3;
```

Можно передать больше двух operandов.

Сокращённые вычисления

Когда найден первый истинный operand, дальнейшие значения не вычисляются. Так можно делать сокращённое вычисление. Operandами могут быть как отдельные значения, так и произвольные выражения. В примере ниже значение переменной «x» не изменится, потому что вычисления туда просто не дойдут:

```
let x;
true || (x = 1);
console.log(x); // undefined, потому что (x = 1) не вычисляется
```

Оператор «и»

Оператор «и» $\&\&$ возвращает первый ложный operand или просто любой последний, если ложных нет.

```
console.log( 1 && null && 2 )
// null
```

```
console.log(1 && 2 && 3 && 4)
// 4
```

Задачи

Что выведет код ниже?

```
console.log( console.log(1) && console.log(2) );
// 1
// undefined
```

console.log выполняет печать на экран, но, как функция, возвращает undefined. Первая функция отработает и сразу же вернёт undefined. Оператор «и» вернёт этот undefined, потому что логическое значение undefined – false.

Что выведет код?

```
console.log(2 && 3 && 4 || 1)
// 4

console.log(1 || 2 && 3 && 4)
// 1
```

Приоритет оператора «и» больше, чем «или», так что он выполняется раньше. «И» возвращает первое ложное значение или просто последнее. Но тут это не важно. Скобкам нужно подождать, пока отработает всё выражение. После того, как отработал «и», запускается «или» и возвращает первое истинное значение. Поэтому в первой строчке результат – 4, а во второй – 1.

Оператор «отрицание» !

Оператор принимает один аргумент, затем:

1. сначала приводит аргумент к логическому типу true/false;
2. возвращает противоположное значение.

```
console.log( !true ); // false
console.log( !0 );    // true
```

Двойное «не» можно использовать для преобразования значений к логическому типу:

```
console.log( !!"non-empty string" ); // true
console.log( !!null );              // false
```

Оператор нулевого слияния ??

Оператор нулевого слияния (??) – это логический оператор, который возвращает значение правого операнда когда значение левого операнда равно [null](#) или [undefined](#), в противном случае будет возвращено значение левого операнда.

Относительно новый оператор (октябрь 2020), страница в [MDN](#).

Это быстрый способ выбрать первое «определенное» значение из списка.

Если переменная «a» имеет значение «null» или «undefined», значит она не определена и ей будет присвоено значение «b»:

```
a ?? b
```

Живой пример:

```
let a = null;
console.log(a ?? 12); // 12

let user;
alert(user ?? "Аноним"); // Аноним
```

```
let user = "Иван";
alert(user ?? "Аноним"); // Иван

let firstName = null;
let lastName = null;
let nickName = "Суперкодер";

alert(firstName ?? lastName ?? nickName ?? "Аноним"); // Суперкодер
// показывает первое определённое значение
```

Для понимания, реалиовать оператор «??» самостоятельно можно было бы так:

```
result = (a !== null && a !== undefined) ? a : b;
```

Он похож на оператор «или», который возвращает первое истинное значение, но они не идентичны. Важное различие между ними заключается в том, что:

|| возвращает первое **истинное** значение.

?? возвращает первое **определенное** значение.

Получается, что такие значения, как ноль, false или пустая строка тольковались бы как «не присвоенные», что было бы ошибкой в данном контексте.

```
let height = 0;

alert(height || 100); // 100
alert(height ?? 100); // 0
```

Приоритет

У этого оператора приоритет 5, согласно [таблице](#). Он ниже, чем у оператора ||.

Важно, что в русскоязычной версии таблицы его пока нет (ноябрь 2020).

Совместное использование с операторами || и &&

По соображениям безопасности JavaScript запрещает использование оператора ?? вместе с && и ||, если только приоритет явно не указан в круглых скобках:

```
let x = 1 && 2 ?? 3; // синтаксическая ошибка

let x = (1 && 2) ?? 3; // 2, без ошибок
```

Операторы побитовые

[Побитовые операторы](#) на MDN.

Они используются редко. Чтобы понять их, нам нужно углубиться в низкоуровневое представление чисел, и было бы неоптимально делать это прямо сейчас, тем более что они нам не понадобятся в ближайшее время.

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления.

Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования.

Поддерживаются следующие побитовые операторы:

- AND(и) (&)
- OR(или) (|)
- XOR(побитовое исключающее или) (^)
- NOT(не) (~)

- LEFT SHIFT(левый сдвиг) (<<)
- RIGHT SHIFT(правый сдвиг) (>>)
- ZERO-FILL RIGHT SHIFT(правый сдвиг с заполнением нулями) (>>>)

Оператор запятая/последовательность

Изредка используется для написания более короткого кода, поэтому его нужно просто знать.

Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего.

Например:

```
let a = (1 + 2, 3 + 4);

console.log(a);
// 7 (результат 3 + 4)
```

Первое выражение $1 + 2$ выполняется, но результат отбрасывается. Затем идёт $3 + 4$, выражение выполняется и возвращается результат.

Обратите внимание, что оператор «,» имеет очень низкий приоритет, ниже `=`, поэтому скобки важны в приведённом примере. Без них в `a = 1 + 2, 3 + 4` выполнится так: $(a = 1 + 2), 3 + 4$.

Иногда его используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

```
// три операции в одной строке
for (a = 1, b = 3, c = a * b; a < 10; a++) {
  ...
}
```

Управление потоком выполнения

`if, else`

`if` – это оператор, инструкция. Вычисляет выражение в скобках и преобразует результат к логическому типу. Если предикат – истина, то выполняется соответствующий блок кода.

Значения, которые всегда `false`

Есть несколько значений, которые называют ложными. К `false` всегда преобразуются:

```
0
"" (пустая строка)
NaN
null
undefined
```

Остальные значения становятся `true`, поэтому их называют «правдивыми» («truthy»).

В действии

```
if (year < 2015) {
  result = 'мало';
} else if (year > 2015) {
  result = 'много';
} else {
  result = 'верно!'
}
```

? , тернарный оператор

Используется, когда надо что-то присвоить переменной в зависимости от какого-то условия. В отличии от if-else, является выражением, а не инструкцией. Следовательно, она вычисляет и возвращает значение, можно сохранить результат вычисления этого выражения в константе. Условная конструкция if в JavaScript выражением не является. If – это инструкция, она выполняет действие, ничего не вычисляя и не возвращая. Поэтому нельзя с помощью конструкции if присваивать значения. Смысл оператора вопросительный знак ? – вернуть то или иное значение, в зависимости от условия и использовать его надо именно для этого. Не надо использовать его для ветвления кода, для этого существует if.

Общая форма выглядит так:

```
predicate ? <expression on true> : <expression on false>
result = a > 0 ? 'yes' : 'no';
```

Сочетается с функциями-однострочниками, когда надо вернуть то или иное значение в зависимости от логического условия:

```
const example = (a) => a > 0 ? 'yes' : 'no';
const abs = number => (number >= 0 ? number : -number);
```

Можно разбивать на несколько строк:

```
let welcome = (age < 18) ?
  function() { alert("Привет!"); } :
  function() { alert("Здравствуйте!"); };
```

Тернарные операторы могут быть вложенными:

```
let age = prompt('Возраст?', 18);

let message = (age < 3) ? 'Здравствуй, малыш!' :
  (age < 18) ? 'Привет!' :
  (age < 100) ? 'Здравствуйте!' :
  'Какой необычный возраст!';

alert( message );
```

это обычная последовательная проверка:

1. Первый знак вопроса проверяет age < 3.
2. Если верно – возвращает 'Здравствуй, малыш!'. В противном случае, проверяет выражение после двоеточия „：“, вычисляет age < 18.
3. Если это верно – возвращает 'Привет!'. В противном случае, проверяет выражение после следующего двоеточия „：“, вычисляет age < 100.
4. Если это верно – возвращает 'Здравствуйте!'. В противном случае, возвращает выражение после последнего двоеточия – 'Какой необычный возраст!'.

То же самое при использовании if..else:

```
if (age < 3) {
  message = 'Здравствуй, малыш!';
} else if (age < 18) {
  message = 'Привет!';
} else if (age < 100) {
  message = 'Здравствуйте!';
} else {
```

```
message = 'Какой необычный возраст!';
}
```

switch

Конструкция switch заменяет сразу несколько if. Она имеет один или более блок case и необязательный default. Несколько значений case можно группировать.

Переменная «x» поочерёдно проверяется на строгое равенство значению в первом case, затем значению во втором case и так далее.

Если равенство установлено, то выполнится и текущая директива, и все остальные директивы, расположенные ниже (без проверок). Чтобы выполнилась только текущая директива, после её инструкций надо установить оператор **break**.

Если ни один case не совпал, то выполнится вариант default, если его указали.

```
const letter = 'a'

switch(letter) {
  case 'a': // if (letter === a)
    console.log('Alpha');
    break;

  case 'b':
  case 'c':
  case 'd':
    console.log('Other');
    break;

  default:
    console.log("Nothing");
    break;
}

// Alpha
```

Если упустить break, то будет так:

```
const letter = 'a'

switch(letter) {
  case 'a':
    console.log('Alpha');

  case 'b':
  case 'c':
  case 'd':
    console.log('Other');

  default:
    console.log("Nothing");
}

// Alpha
// Other
// Nothing
```

Циклы

while

Всё как обычно: предикат, условие, выполнение. Инструкции break и continue присутствуют: Continue – это пропуск текущей итерации и переход на новую. Break – остановка и выход из цикла.

```
const number = 10;
let i = 1;

while (i <= number) {
  console.log(i);
  i += 1;
}
```

Если тело цикла состоит из одной инструкции, то {фигурные} скобки не требуются:

```
let i = 3;
while (i) alert(i -= 1);
```

do...while

Выражение **do...while** создает цикл, который выполняет указанное выражение до тех пор, пока условие не станет ложным. Условие проверяется после выполнения выражения, то есть выражение выполнится как минимум один раз. Такая форма синтаксиса оправдана, если вы хотите, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике используется редко.

Документация [здесь](#).

Цикл сначала выполнит тело, а затем проверит условие, и пока его значение равно true, он будет выполняться снова и снова:

```
do {
  тело цикла
} while (условие)

let i = 0;
do {
  alert( i );
  i += 1;
} while (i < 3);
```

for

Цикл for состоит из трех блоков:

```
for(начальное действие; условие; шаг) {
  тело цикла;
}
```

Блоки:

Начальное действие – любое действие, которое выполняется перед запуском цикла. Например, можно создать переменную-счётчик.

Условие – собственно, условие, которое проверяется на каждой итерации. Если условие истинно, то выполняется шаг цикла.

Шаг – это изменение счётчика. Кроме шага, сюда можно записать любые другие операции.

Любой блок цикла for может быть оставлен пустым, т.е. пропущен:

```
let i = 0; // мы уже имеем объявленную i с присвоенным значением
for (; i < 3; i++) { // нет необходимости в начальном действии
    alert( i ); // 0, 1, 2
}
```

При этом сами точки с запятой ; обязательно должны присутствовать, иначе будет ошибка синтаксиса.

В процессе работы цикла выражение, которое содержится в блоке условий, каждый раз вычисляется заново.

Например, если это массив, то может каждый раз вычисляться его длина, хотя она остаётся неизменной.

Повторное вычисление оказывается избыточным. В таком случае, можно определить константу за пределами цикла и присвоить ей необходимое значение:

```
lastIndex = array.length - 1;
```

Область видимости цикла

Любые переменные (объявленные с помощью const или let), созданные при инициализации (в круглых скобках) цикла имеют локальную область видимости: они видны только в пределах цикла и существуют только в период выполнения цикла. После завершения цикла эти локальные переменные удаляются из памяти. Попытка обратиться за пределами цикла к локальным переменным цикла вызовет ошибку ReferenceError.

Управляющие инструкции

В циклах JavaScript доступны для использования две инструкции, влияющие на их поведение: break и continue. Инструкция **break** производит *выход из цикла*. Не из функции, а из цикла. Встретив её, интерпретатор выходит из всего цикла и переходит к инструкциям, идущими сразу за ним. Вместо инструкции break рекомендуется использовать цикл while.

Инструкция **continue** позволяет пропустить текущую итерацию цикла.

Ниже пример с функцией myCompact, которая удаляет null элементы из массива:

```
const myCompact = (coll) => {
    const result = [];

    for (const item of coll) {
        if (item === null) {
            continue;
        }

        result.push(item);
    }

    return result;
};
```

То же самое можно сделать без continue:

```
const myCompact = (coll) => {
    const result = [];

    for (const item of coll) {
        if (item !== null) {
            result.push(item);
        }
    }

    return result;
};
```

Метки для break/continue

Бывает, нужно выйти одновременно из нескольких уровней цикла сразу. Обычный `break` во внутреннем цикле прервёт только внутренний цикл, а не внешний и одновременно. Одновременно пропускать итерации или выключать несколько циклов можно с помощью меток.

Break с меткой выключит несколько циклов.

Директива `continue` также может быть использована с меткой. В этом случае управление перейдёт на следующую итерацию цикла с меткой.

Вызов `break/continue` возможен только внутри цикла, и метка должна находиться где-то выше этой директивы.

Метка имеет вид идентификатора с двоеточием перед циклом:

```
labelName: for (...) {  
  ...  
}
```

Можно размещать метку на отдельной строке:

```
labelName:  
for (...) {  
  ...  
}
```

Вызов `break` в цикле ниже ищет ближайший внешний цикл с такой меткой и прерывает его.

```
outer: for (let i = 1; i < 5; i += 1) {  
  console.log('цикл 1, итерация', i);  
  
  for(let j = 1; j < 2; j += 1) {  
    console.log('цикл 2, итерация', j);  
    if (j === 1) {  
      break outer;  
    }  
  }  
}  
  
// цикл 1, итерация 1  
// цикл 2, итерация 1
```

for...of

Оператор `for...of` выполняет цикл обхода [итерируемых объектов](#) (`Array`, `Map`, `Set`, объект [аргументов](#) и подобных), вызывая на каждом шаге итерации операторы для каждого значения из различных свойств объекта.

Документация [здесь](#).

Я один раз поставил «`in`» вместо «`of`». Это [отдельная](#) инструкция.

Также ниже смотри метод `.forEach()` для массивов и массивоподобных объектов.

```
const myArray = [1, 2, 3];  
  
for(let i of myArray) {  
  console.log(i);  
}
```

for...in

Для перебора всех свойств (ключей) из объекта используется цикл по свойствам for..in. Это цикл по ключам. Порядок перебора соответствует порядку объявления ключей, если они нечисловые. Если ключи числовые – они сортируются автоматом (в современных браузерах) и обход производится соответственно по возрастанию.

Важно:

Цикл for...in также будет обходить и ключи прототипа объекта, если они перечислимые.

Пример:

при каждой итерации i будет принимать значение ключа в объекте:

```
const object = {
  key1: 'value1',
  key2: 'value2',
  key3: {
    subKey1: 'subValue1',
    subKey2: 'subValue2',
  }
}

for (let i in object) {
  console.log(i);
}

// key1
// key2
// key3
```

Если подставить массив, то i будет принимать значения индексов (потому что массив – это тоже объект):

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

for (let i in arr) {
  console.log(i);
}

// 0
// 1
// 2
```

Если в качестве свойств используются числа, то они будут отсортированы при вызове.

```
let codes = {
  "49": "Германия",
  "41": "Швейцария",
  "44": "Великобритания",
  "1": "США"
};

for (let code in codes) {
  console.log(code);
}

// 1, 41, 44, 49
```

Телефонные коды идут в порядке возрастания, потому что они являются целыми числами: 1, 41, 44, 49.

Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений.

А вот свойства "+49" или "1.2" таковыми не являются.

Если ключи не целочисленные, то они перебираются в порядке создания.

Функции (кратко)

Обычные значения, такие как строки или числа представляют собой *данные*.

Функции, с другой стороны, можно воспринимать как «действия».

Записано в файл «**Функции, полный конспект**».

</>

JS - Типы данных

Как устроены примитивы

Примитив – значение «примитивного» типа.

Есть 7 примитивных типов: string, number, boolean, symbol, null, undefined и bigint.

Объект

- Может хранить множество значений как свойства.

- Объявляется при помощи фигурных скобок {}, например: {name: "Рома", age: 30}. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что мы можем хранить функцию как одно из свойств объекта.

Каждый примитив имеет свой собственный «объект-обёртку», которые называются: String, Number, Boolean и Symbol. Таким образом, они имеют разный набор методов, и это позволяет с ними работать: применять методы и всё такое. Обёртка предоставляет нужную функциональность, а после удаляется.

Конструкторы обёрток

Каждый примитив имеет свой собственный «объект-обёртку»: String, Number, Boolean и Symbol, потому что им нужен разный набор методов.

Конструкторы new String/ new Number/ new Boolean предназначены только для автоматического использования движком, специально их использовать не рекомендуется, потому что возвращаемое значение будет именно объектом, а не примитивом. Конструкторы всегда возвращают объект. Например, можно создать число 0 через конструктор, и оно будет не примитивом, а объектом. Логическое значение числа 0 – false, а любого объекта – true, что доказывается в примере ниже.

```
const a = new Number(0);

console.log( typeof a ); // object
console.log( typeof 0 ); // number

console.log( a === 0 ); // false
console.log( a == 0 ); // true
```

Оператор нестрогого равенства преобразует значение объекта в примитив, поэтому проверка на нестрогое равенство с нулём проходит. Строгое равенство показывает, что типы данных у переменных разные.

Но эти функции String/Number/Boolean можно использовать **без оператора new**. Без «new» они будут превращать значение в примитивный тип: в строку, в число, в булевый тип.

К примеру, следующее вполне допустимо:

```
let num = Number("123"); // превращает строку в число
```

null/undefined не имеют методов.

У них нет соответствующих «объектов-обёрток» и они не имеют никаких методов. В некотором смысле, они самые примитивные.

Попытка доступа к свойствам такого значения возвратит ошибку.

Задачи

Можно ли добавить свойство строке?

```
let str = "Привет";
str.test = 5;
console.log(str.test);

// TypeError: Cannot create property 'test' on string 'Привет'
```

В зависимости от того, используете ли вы строгий режим (use strict):

- undefined (без strict)
- ошибка (strict mode)

Давайте посмотрим что происходит:

1. В момент обращения к свойству str создаётся «объект-обёртка».
2. В строгом режиме, попытка изменения этого объекта выдаёт ошибку.
3. Без строгого режима, операция продолжается, объект получает свойство test, но после этого он удаляется, так что на последней линии str больше не имеет свойства test.

Данный пример наглядно показывает, что примитивы не являются объектами. Они не могут хранить дополнительные данные.

Числа

В JavaScript существует два типа чисел: обычные и BigInt.

Обычные числа типа **number**, которые хранятся в 64-битном формате [IEEE-754](#), который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers).

BigInt числа дают возможность работать с целыми числами, когда необходимо работать со значениями более чем 2^{53} или менее чем -2^{53} . Так как BigInt числа нужны достаточно редко, мы рассмотрим их в отдельной главе [BigInt](#).

См. также раздел «численное преобразование».

e, e-

В жизни чтобы не ошибиться, вместо записи 1 000 000 можно написать 1 млн. В JavaScript можно использовать букву «е» и «е-», чтобы укоротить запись числа.

«е» производит операцию умножения числа на 1 с указанным количеством нулей:

«е-» производит деление на 1 с указанным количеством нулей:

```
1e3;
// 1 * 1000 = 1000

1e-3;
// 1 / 1000 = 0.001

1.7e1;
// 1.7 * 10 = 17
```

Шестнадцатеричные, двоичные и восьмеричные числа

0x – шестнадцатеричные числа.

0b – двоичные

0o – восьмеричные

```
16
console.log( 0xff );
// 255
console.log( 0xFF );
// 255 (регистр не имеет значения)

2
console.log(0b11111111);
// 255

8
console.log(0o377);
// 255
```

Шестнадцатеричные числа широко используются в JavaScript для представления цветов, кодировки символов и многоного другого.

Системы счисления `toString(base)`

Метод `num.toString(base)` возвращает строковое представление числа `num` в системе счисления `base`.

```
let num = 255;
num.toString(16);
// ff

(12).toString(2);
(12..toString(2);

// 1100
```

Если надо вызвать метод непосредственно на числе, то надо поставить две точки .. после числа или взять число в скобки.

Если мы поставим одну точку: `123456.toString(36)`, тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки или скобки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод.

`base` может варьироваться от 2 до 36 (по умолчанию 10).

Часто используемые:

base=16 — для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть 0..9 или A..F.

base=2 — обычно используется для отладки побитовых операций, цифры 0 или 1.

base=36 — максимальное основание, цифры могут быть 0..9 или A..Z. То есть, используется весь латинский алфавит для представления числа.

Округление

`Math.floor`

Округление в меньшую сторону: 3.1 становится 3, а -1.1 — -2.

`Math.ceil`

Округление в большую сторону: 3.1 становится 4, а -1.1 — -1.

Math.round

Округление до ближайшего целого: 3.1 становится 3, 3.6 — 4, а -1.1 — -1.

А 0.5 куда округляется? От 0.5 — в большую сторону.

Math.trunc (не поддерживается в Internet Explorer)

Производит удаление дробной части без округления: 3.1 становится 3, а -1.1 — -1.

num.toFixed(n)

Метод `toFixed(n)` округляет число до n знаков после запятой аналогично `round` и возвращает строковое представление результата.

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

Эти функции охватывают все возможные способы обработки десятичной части.

Например, если 1.2345 надо округлить до 2-х знаков после запятой, есть два пути решения:

1. Умножить и разделить.

Например, чтобы округлить число до второго знака после запятой, мы можем умножить число на 100, вызвать функцию округления и разделить обратно:

```
let num = 1.23456;  
Math.floor(num * 100) / 100;  
// 1.23
```

2. Использовать `toFixed(n)`.

Метод `toFixed(n)` округляет число до n знаков после запятой аналогично `round` и возвращает строковое представление результата.

Результатом `toFixed` является строка, поэтому её надо преобразовать в число через унарный оператор + или `Number()`.

Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки.

```
(1.23456).toFixed(2);  
// 1.23  
  
13..toFixed(5);  
// 13.00000
```

Неточные вычисления

Внутри JavaScript число представлено в виде 64-битного формата [IEEE-754](#). Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранение знака.

Если число слишком большое и переполнит 64-битное хранилище, JavaScript вернёт бесконечность.

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript — это потеря точности.

Посмотрите на это (неверное!) сравнение:

```
0.1 + 0.2 === 0.3;
```

```
// false  
  
console.log(0.1 + 0.2)  
// 0.3000000000000004
```

Почему так происходит.

Число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как 0.1, 0.2, на самом деле являются бесконечной дробью в двоичной форме.

Другими словами, 0.1 – это единица делённая на десять — 1/10, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с одной третьей: 1/3, которая становится бесконечной дробью 0.33333(3).

Деление на 10 гарантированно хорошо работает в десятичной системе, но деление на 3 – нет. По той же причине и в двоичной системе счисления, деление на 2 обязательно сработает, а 1/10 становится бесконечной дробью.

Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют нам увидеть эту «крошечную потерю точности», но она существует.

И когда мы суммируем 2 числа, их «неточности» тоже суммируются.

Наиболее надёжный способ — это округлить результат используя метод [toFixed\(n\)](#).

Помните, что метод toFixed всегда возвращает строку. Это удобно для форматирования цен в интернет-магазине \$0.30.

Забавный пример:

```
// Привет! Я – число, растущее само по себе!  
alert( 999999999999999 ); // покажет 1000000000000000
```

Причина та же – потеря точности. Из 64 бит, отведённых на число, сами цифры числа занимают до 52 бит, остальные 11 бит хранят позицию десятичной точки и один бит – знак. Так что если 52 бит не хватает на цифры, то при записи пропадут младшие разряды.

Интерпретатор не выдаст ошибку, но в результате получится «не совсем то число», что мы и видим в примере выше. Как говорится: «как смог, так записал».

Два нуля

Другим забавным следствием внутреннего представления чисел является наличие двух нулей: 0 и -0.

Все потому, что знак «минус» представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая ноль.

В большинстве случаев это поведение незаметно, операторы в JavaScript воспринимают 0 и -0 одинаковыми.

isFinite и isNaN и Object.is

Существуют специальные числовые значения:

[Infinity](#), [-Infinity](#) – математическая бесконечность ∞

[NaN](#) – ошибка вычисления

Эти значения принадлежат типу `number`, но они не являются «обычными» числами, поэтому нужны специальные функции для их проверки.

isNaN (v)

преобразует значение в число и проверяет является ли оно `NaN`.

Важно: `NaN` уникально тем, что оно не является равным ни чему другому, даже самому себе. Поэтому этот метод – единственный способ проверить `NaN` на `NaN`.

```
NaN === NaN;          // false  
isNaN(NaN);         // true  
  
isNaN('');           // false  
isNaN("str");        // true
```

```
isNaN(null);      // false  
isNaN(undefined); // true
```

isFinite (v)

Преобразует аргумент в число и возвращает true, если оно является обычным числом.

Возвращает false, если оно NaN, Infinity, undefined или строка - не число.

```
isFinite(NaN);      // false  
isFinite(Infinity); // false  
isFinite(undefined); // false  
  
isFinite('');       // true  
isFinite(null);    // true  
isFinite('19');    // true  
isFinite('str');   // false
```

Иногда `isFinite` используется для проверки, содержится ли в строке число.

Помните, что пустая строка интерпретируется как 0 во всех числовых функциях, включая `isFinite`.

См. задачу «[Ввод числового значения](#)».

Object.is()

определяет, являются ли два значения [одинаковыми значениями](#). [MDN](#).

```
let isSame = Object.is(value1, value2);
```

работает с NaN
Object.is(NaN, NaN);
// true;

Отличает нули
Object.is(0, -0);
// false

Метод сравнивает значения как равенство, но, в отличие от равенства, также работает в двух ситуациях:

1. Работает с NaN: `Object.is(NaN, NaN) === true`

2. Показывает, что значения 0 и -0 разные: `Object.is(0, -0) === false`, это редко используется, но технически эти значения разные.

Отличие от оператора ==

Оператор == использует приведение типов обоих operandов перед проверкой на равенство (в результате получается, что проверка "" == false даёт true).

Метод Object.is приведение типов не выполняет.

Отличие от оператора ===

Оператор === считает числовые значения -0 и +0 равными, а значение Number.NaN не равным самому себе.

Во всех других случаях Object.is(a, b) идентичен a === b.

Этот способ сравнения часто используется в спецификации JavaScript. Когда внутреннему алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is (Определение [SameValue](#)).

Преобразование к числу, `parseInt` и `parseFloat`

Для явного преобразования к числу можно использовать + или `Number()`.

Если строка не является в точности числом, то результат будет NaN. Единственное исключение — пробелы в начале строки и в конце не приводят к NaN, они игнорируются.

В жизни часто значения записаны с единицами измерения: 100px, 12pt в CSS, 19€. Получить значения без единицы измерения можно с помощью `parseInt` и `parseFloat`.

Функция `parseInt` парсит строку и возвращает целое число, а `parseFloat` – число с плавающей точкой.

Как только они сталкиваются с первым символом – не буквой, они возвращают прочитанные числа. Если первое значение – не цифра, то они возвращают ошибку.

```
console.log( parseInt('100px') );
// 100
console.log( parseFloat('12.5em') );
// 12.5

console.log( parseInt('12.3') );
// 12, вернётся только целая часть
console.log( parseFloat('12.3.4') );
// 12.3, остановка чтения на второй точке
```

Функция `parseInt()` имеет необязательный второй параметр. Он определяет систему счисления, таким образом `parseInt` может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.:

```
console.log( parseInt('0xff', 16) );
// 255
```

Math

В JavaScript встроен объект [Math](#), который содержит различные математические функции и константы.

Их множество, подробнее – в документации.

См. соответствующий раздел с объектами и методами снизу.

Случайное число

Функция генерирует случайное число с плавающей точкой от `min` до `max` (но не включая `max`).

Для этого нужно преобразовать каждое значение из интервала 0-1 в значения от `min` до `max`. Это можно сделать в 2 шага:

1. Если умножить число от 0-1 на (`max - min`), тогда интервал возможных значений от 0-1 увеличивается до от 0 до (`max-min`).
2. Соответственно, надо прибавить `min` к нулю, чтобы интервал стал от `min` до `max`.

```
Min = 2, Max = 5
0,5 * (5 - 2) = 1,5
2 + 1,5 = 3,5
```

```
function random(min, max) {
  return min + Math.random() * (max - min);
}
```

Функция генерирует случайное целое число от `min` до `max` (включительно). Любое число из интервала `min..max` должно появляться с одинаковой вероятностью.

Правильное решение задачи

Есть много правильных решений этой задачи.

Одно из них – правильно указать границы интервала. Чтобы выровнять интервалы, мы можем генерировать числа от 0.5 до 3.5, это позволит добавить необходимые вероятности к `min` и `max`. Это костыль для предыдущего метода:

```
function randomInteger(min, max) {
  // получить случайное число от (min-0.5) до (max+0.5)
```

```
let rand = min - 0.5 + Math.random() * (max - min + 1);
return Math.round(rand);
}
```

Другое правильное решение – это использовать Math.floor для получения случайного числа от min до max+1:

```
function randomInteger(min, max) {
  // случайное число от min до (max+1)
  let rand = min + Math.random() * (max + 1 - min);
  return Math.floor(rand);
}
```

Все интервалы имеют одинаковую длину, что выравнивает вероятность получения случайных чисел. Теперь все интервалы отображаются следующим образом:

```
число от 1 ... до 1.9999999999 округлится до 1
число от 2 ... до 2.9999999999 округлится до 2
число от 3 ... до 3.9999999999 округлится до 3
```

Самое очевидное, но **неправильное** решение – генерировать случайное число от min до max и округлять его. Функция будет работать, но неправильно. Вероятность получить min и max значения в 2 раза меньше, чем любое другое число.

```
function randomInteger(min, max) {
  let rand = min + Math.random() * (max - min);
  return Math.round(rand);
}
```

Это происходит потому, что метод Math.round() получает случайные числа из интервала 1..3 и округляет их следующим образом:

```
число от 1 ... до 1.4999999999 округлится до 1
число от 1.5 ... до 2.4999999999 округлится до 2
число от 2.5 ... до 2.9999999999 округлится до 3
```

Строки

Внутренний формат для строк — всегда [UTF-16](#), вне зависимости от кодировки страницы.

Кавычки

Строка можно создать с помощью одинарных, двойных либо обратных кавычек:

```
let single = 'single-quoted';
let double = "double-quoted";

let backticks = `backticks`;
```

Обратные кавычки имеют больше возможностей:

Возможность вставлять **произвольные выражения**, обернув их в `${ }` , т.е. делать интерполяцию:

```
const a = 123
```

```
console.log(`Значение = ${a}`);
// Значение = 123
```

Возможность делать **предзданное форматирование** текста.

Если попытаться использовать точно так же одинарные или двойные кавычки, то будет ошибка.

```
const str = `этот текст размещён
на нескольких строчках`;

console.log(str);
// этот текст размещён
// на нескольких строчках
```

Обратные кавычки также позволяют задавать «**шаблонную функцию**» перед первой обратной кавычкой.

Используемый синтаксис: `func`string``. Автоматически вызываемая функция `func` получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в [документации](#). На практике используется редко.

Ничего не понял.

Спецсимволы

\	Экранирование
\n	Перенос строки
\t	Табуляция
\xXX	Символ с шестнадцатеричным юникодным кодом XX например, \x7A - то же самое, что z.
\uXXXX	Символ в кодировке UTF-16 с шестнадцатеричным кодом XXXX например, \u00A9 — юникодное представление знака копирайта, ©. Код должен состоять ровно из 4 шестнадцатеричных цифр.
\u{X...XXXXXX}	Символ в кодировке UTF-32 с шестнадцатеричным кодом от U+0000 до U+10FFFF. Некоторые редкие символы кодируются двумя 16-битными словами и занимают 4 байта. Так можно вставлять символы с длинным кодом. (от 1 до 6 шестнадцатеричных цифр)

```
console.log( "\u00A9" );      // ©
console.log( "\u{20331}" );  // 倍, китайский иероглиф
console.log( "\u{1F60D}" );  // 😊, лицо с улыбкой и глазами в форме сердец
```

Доступ к символам

Получить символ в строке можно:

1. с помощью скобок с номером позиции [n]
2. методом [str.charAt\(n\)](#)

`charAt` существует по историческим причинам. Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт `undefined`, а `charAt` — пустую строку.

Можно перебрать строку посимвольно в цикле `for..of`:

```
for (let i of "Hello") {
  console.log(i);
}
```

Трюк с побитовым HE

Вряд ли пригодится. По ссылке можно прочитать главу в учебнике.

Сравнение строк

Строки сравниваются посимвольно в алфавитном порядке. Тем не менее, есть некоторые нюансы.

Строчные буквы больше заглавных:

```
alert( 'a' > 'Z' ); // true
```

Буквы, имеющие диакритические знаки, идут «не по порядку»:

```
alert( 'Österreich' > 'Zealand' ); // true
```

Строки кодируются в [UTF-16](#). Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот.

str.codePointAt(pos)

Возвращает код для символа, находящегося на позиции pos:

```
alert( "z".codePointAt(0) ); // 122
alert( "Z".codePointAt(0) ); // 90
```

Str.fromCodePoint(code)

Создаёт символ по его коду code:

```
alert( String.fromCodePoint(90) ); // Z
```

Через цикл «с... по» можно перебрать всю таблицу юникод.

Например, символы с кодами от 65 до 127 — это латиница и ещё некоторые распространённые символы.

```
let str = '';
for (let i = 65; i <= 127; i++) {
  str += String.fromCodePoint(i);
}
console.log( str );
// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Теперь очевидно, почему a > Z.

Символы сравниваются по их кодам. Больший код — больший символ. Код a (97) больше кода Z (90).

можно добавлять юникодные символы по их кодам, используя \u с шестнадцатеричным кодом символа. 90 — это «5а» в шестнадцатеричной системе счисления:

```
alert( '\u005a' ); // Z
```

Правильное сравнение

«Правильный» алгоритм сравнения строк сложнее, так как разные языки используют разные алфавиты.

Поэтому браузеру нужно знать, какой язык использовать для сравнения.

Все современные браузеры (IE10 дополнительная библиотека [Intl.js](#)) поддерживают стандарт [ECMA 402](#), обеспечивающий правильное сравнение строк на разных языках с учётом их правил. Для этого есть соответствующий метод.

[str.localeCompare\(str2\)](#)

возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

- Отрицательное число, если str меньше str2.
- Положительное число, если str больше str2.
- 0, если строки равны.

У этого метода есть два дополнительных аргумента.

Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй — определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между «а» и «á».

```
alert( 'Österreich'.localeCompare('Zealand') ); // -1
```

Суррогатные пары

Многие символы возможно записать одним 16-битным словом. Но 16 битов — это 65536 комбинаций, так что на все символы этого не хватит. Поэтому редкие символы записываются двумя 16-битными словами — это также называется «**суррогатная пара**».

Суррогатные пары не существовали, когда был создан JavaScript, поэтому язык не обрабатывает их адекватно

Длина таких строк = 2:

```
console.log( '𠮷'.length ); // 2
console.log( '߱'.length ); // 2
console.log( '߳'.length ); // 2
```

`String.fromCodePoint` и `str.codePointAt` — два редких метода, правильно работающие с суррогатными парами, но они и появились в языке недавно.

Получить символ суррогатной пары просто так не получится:

```
console.log( '𠮷'[0] ); // ?
console.log( '𠮷'[1] ); // ?
```

в главе [Перебираемые объекты](#) будут ещё способы работы с суррогатными парами. Для этого есть и специальные библиотеки, но нет достаточно широко известной, чтобы предложить её здесь.

Диакритические знаки и нормализация

См. по ссылке.

Методы (удалить?)

Изменение регистра

`toLowerCase()` и `toUpperCase()` меняют регистр символов всей строки:

```
'Interface'.toUpperCase();
// INTERFACE
```

```
'Interface'.toLowerCase();
// interface
```

Чтобы изменить регистр только первой буквы, можно делать так:

```
const a = 'str';
const b = a[0].toUpperCase() + a.slice(1);
console.log(b)
```

Поиск подстроки

.indexOf()

Метод возвращает индекс первого вхождения указанного значения в строку, на которой он был вызван, начиная с индекса fromIndex. Возвращает -1, если значение не найдено.

Является регистрозависимым.

```
str.indexOf(searchValue, [fromIndex])
```

fromIndex

Необязательный. Индекс, откуда начинать поиск. Значение по умолчанию = 0.

Такой же метод есть у массивов.

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей.

Мой пример:

```
let str = 'Ослик Иа-Иа посмотрел на виадук';

function foo(str, char, position = 0) {
  let newPosition = str.indexOf(char, position);
  if (newPosition === -1) {
    return;
  }
  console.log(newPosition);
  return foo(str, char, newPosition + 1);
};

foo(str, 'Иа');

// Найдено тут: 6
// Найдено тут: 9
```

Аналогичный оригинальный через циклы:

```
let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа';
let pos = 0;

while (true) {
  let foundPos = str.indexOf(target, pos);

  if (foundPos == -1) {
    break;
  }

  console.log(`Найдено тут: ${foundPos}`);
  pos = foundPos + 1; // продолжаем со следующей позиции
}

// Найдено тут: 6
// Найдено тут: 9
```

.lastIndexOf()

Похожий метод, который ищет с конца строки к её началу.

Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

```
str.lastIndexOf(searchValue[, fromIndex])
```

fromIndex

Необязательный параметр. Местоположение внутри строки, откуда начинать поиск, нумерация индексов идёт слева направо. Значение по умолчанию = str.length. Если оно отрицательно, трактуется как 0. Если fromIndex > str.length, параметр fromIndex будет трактоваться как str.length.

```
console.log('dddd'.lastIndexOf('d', 3))  
// 3
```

.includes()

Метод проверяет, содержит ли строка заданную подстроку, и возвращает true или false.

Метод является регистрозависимым.

Правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна.

```
str.includes(searchString[, position])
```

position

Позиция в строке, с которой начинать поиск строки searchString, по умолчанию 0.

.startsWith()

Определяет, начинается ли строка символами другой строки. Возвращает true или false.

Так, например, при наборе номера +7 можно определить, что номер российский.

```
str.startsWith(searchString[, position])
```

searchString

Символы, искомые в начале данной строки

position

Необязательный. Позиция, с которой начинать поиск строки searchString

.endsWith()

Определяет, заканчивается ли строка символами другой строки, возвращая, соответственно, true или false.

```
str.endsWith(searchString[, position])
```

searchString

Символы, искомые в конце данной строки.

position

Необязательный. Искать от 0 до этого символа; по умолчанию установлен в реальную длину строки.

Получение подстрок (резз)

метод	выбирает	отрицательные значения
substr	подстроку от позиции start до end	если start < 0, то берёт от конца строки
slice	подстроку от позиции start до end	если start < 0, то берёт от конца строки
substring	подстроку от позиции start до end	если start < 0, то берёт от конца строки

<code>slice(start, end)</code>	от start до end (не включая end)	можно передавать, отсчитывает влево от последнего индекса
<code>substring(start, end)</code>	от start до end (не включая end)	равнозначны 0
<code>substr(start, length)</code>	length символов, начиная от start	start может быть отрицательным

.substring ()

Документация [здесь](#).

Метод возвращает подстроку между двумя индексами, или от одного индекса и до конца строки.

```
str.substring(indexA[, indexB])
```

`indexA`

от 0 до длины строки. Индекс будет включён в результирующую подстроку.

`indexB`

Необязательный. От 0 до длины строки. Индекс не будет включён в результирующую подстроку.

Если аргумент `indexA` будет больше аргумента `indexB`, то метод поменяет их местами.

.substr (a, b)

Рекомендуется не использовать. Позволяет указать длину вместо конечной позиции.

.slice ()

Метод `slice()` извлекает часть строки и возвращает новую строку.

```
str.slice(beginSlice[, endSlice])
```

`beginSlice`

Индекс начала извлечения Если отрицателен, то отсчитывается от конца строки.

`endSlice`

Необязательный. Индекс окончания извлечения, в подстроку не входит. Если отрицателен, то тоже отсчитывается от конца строки.

Задачи

Сделать первый символ заглавным

Напишите функцию `ucFirst(str)`, возвращающую строку `str` с заглавным первым символом.

Можно пересоздать строку на основе существующей с первым заглавным первым:

```
const str2 = '';
const newStr2 = str2[0].toUpperCase() + str2.slice(1);
// TypeError: Cannot read property 'toUpperCase' of undefined
```

Это не работает с пустой строкой. Выхода два:

- Использовать `str.charAt(0)`, поскольку этот метод всегда возвращает строку (для пустой строки — пустую).
- Добавить проверку на пустую строку.

```
function ucFirst(str) {
  if (!str) return str;

  return str[0].toUpperCase() + str.slice(1);
}
```

Проверка на спам

Напишите функцию checkSpam(str), возвращающую true, если str содержит 'viagra' или 'XXX', а иначе false. Функция должна быть нечувствительна к регистру:

```
checkSpam('buy ViAgRA now') == true
checkSpam('free xxxxx') == true
checkSpam("innocent rabbit") == false
```

```
function checkSpam(str) {
  const stopList = ['viagra', 'xxx'];

  for(let word of stopList) {
    if (str.toLowerCase().includes(word)) {
      return true;
    }
  }

  return false;
}
```

Усечение строки

Создайте функцию truncate(str, maxlength), которая проверяет длину строки str и, если она превосходит maxlength, заменяет конец str на "...", так, чтобы её длина стала равна maxlength.

Результатом функции должна быть та же строка, если усечение не требуется, либо, если необходимо, усечённая строка.

В качестве многоточия здесь используется ... — ровно один специальный юникодный символ. Это не то же самое, что ... — три точки.

```
truncate("Вот, что мне хотелось бы сказать на эту тему:", 20)
// "Вот, что мне хотел..."
```

```
function truncate(str, maxlength) {
  if (str.length <= maxlength) {
    return str;
  }

  const newString = str.slice(0, maxlength - 1) + '...';
  return newString;
}

console.log(truncate("Вот, что мне хотелось бы сказать на эту тему:", 20));
// Вот, что мне хотел...

console.log(truncate("Всем привет!", 20));
// Всем привет!
```

Выделить число

Есть стоимость в виде строки "\$120". То есть сначала идёт знак валюты, а затем – число.

Создайте функцию extractCurrencyValue(str), которая будет из такой строки выделять числовое значение и возвращать его.

```
function extractCurrencyValue(str) {
  return Number(str.slice(1));
```

```
// return +str.slice(1);
}
```

Массивы

Руководство по массивам [тут](#).

Массив – это структура данных, содержащая упорядоченный набор элементов и предоставляющая возможность произвольного доступа к своим элементам.

Для упорядоченных данных использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных `Array`.

Массив следует считать особой структурой, позволяющей работать с *упорядоченными данными*. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому используйте их именно в таких случаях. Если вам нужны произвольные ключи, лучше подойдёт обычный объект `{ }`.

Задача массива представить списки (коллекции элементов) в виде единой структуры, которая позволяет работать с ними как с единым целым.

JavaScript позволяет создавать массив из разнотипных данных. На практике такой подход лучше не использовать. Для разнотипных данных, обычно, хорошо подходит объект.

Виды массива?

Массив в JavaScript — динамическая структура. Её можно расширять прямо в процессе работы программы. В языках, близких к железу, таких как Си, размер массива — постоянная величина. При необходимости расширения в подобных языках создают новый массив).

Несмотря на то, что в определении говорится об упорядоченности, элементы массива не всегда идут один за другим:

элементы	1	2	3	...	n
индексы	[0]	[1]	[2]	...	[n - 1]

Последний индекс в массиве всегда меньше размера массива на единицу.

Для справки:

Массив, как и всё в js – это объект и у него есть свойства. Например, свойство `length`, к которому можно обращаться через квадратные скобки.

Когда мы создаём массив, то в реальности создаётся объект, и фактически квадратные скобки являются синтаксическим сахаром, т.е. заменой на уровне синтаксиса конструкции через `new`. Фактически мы создаём новый инстанс элемента типа `Array`, в котором через запятую перечисляем элементы, которые будут внутри него. Массив вообще можно создать как объект через ключ-значение:

```
const obj = { 0: 1, 1: 'string', 3: [3, NaN] }
console.log(obj[0]); // 1
```

Объекты, имеющие индексированные свойства и `length`, называются **псевдомассивами**. Они также могут иметь другие свойства и методы, но у них нет встроенных методов массивов.

Если мы заглянем в спецификацию, мы увидим, что большинство встроенных методов рассчитывают на то, что они будут работать с итерируемыми объектами или псевдомассивами вместо «настоящих» массивов, потому что эти объекты более абстрактны.

`Array.from(obj[, mapFn, thisArg])` создаёт настоящий Array из итерируемого объекта или псевдомассива obj, и затем мы можем применять к нему методы массивов. Необязательные аргументы mapFn и thisArg позволяют применять функцию с задаваемым контекстом к каждому элементу.

Создание и обращение к элементам

Существует два варианта синтаксиса для создания пустого массива:

```
let arr = new Array();
let arr = new Array(item1, item2...);

let arr = [];

Array.from(obj[, mapFn, thisArg])
```

Если конструктор «`new Array`» вызывается с одним аргументом, который представляет собой число, он создаёт массив *без элементов, но с заданной длиной*.

Висячая запятая

Список элементов массива, как и список свойств объекта, может оканчиваться запятой

```
// обращение к элементам
array[0]    // 1
array[3]    // [3, NaN]
array[3][0] // 3

// если обратиться к элементу, которого не существует, ошибка не произойдёт
array[4];   // undefined

// длина массива (также можно обращаться через квадратные скобки)
array.length;
array['length']; // 4
```

Изменение и дополнение элементов

Значение существующего элемента в массиве можно изменить, если обратиться к индексу и прописать изменённое значение

```
const myArray = [1, 2, 3];

// изменение значения по индексу
myArray[0] = 'change value';

myArray
// [ 'change value', 2, 3 ]
```

Выход за границу массива

Это когда извлекается элемент по несуществующему индексу. Обращение по несуществующему индексу возвращает значение `undefined`. При этом никаких ошибок не возникает, это рассматривается как нормальная ситуация.

```
const myArray = [1, 2, 3];
myArray[5]; // undefined
```

При добавлении в массив нового элемента по индексу, превышающему максимальный более чем на единицу, вставляются фиктивные «пустые» элементы, содержащие значение `undefined`. По-моему это называется произвольным дополнением. Если в массиве 3 элемента, можно сразу же добавить шестой. Если распечатать этот массив, то будет видно недостающие пустые элементы. Длина массива будет равна 6.

Важно обратить внимание, что и если обратиться к несуществующему элементу (за пределами массива) и если обратиться к существующему индексу с пустым элементом, у них будет одинаковое значение: `undefined`. Таким образом, `undefined` не всегда означает, что этого индекса не существует.

```
const myArray = [1, 2, 3];

// добавление нового значения в массив
myArray[5] = 'new value'

myArray
// [ 1, 2, 3, <2 empty items>, 'new value' ]
```

Удаление элементов

Самого удаления не происходит, индекс просто очищается от значения и остаётся существовать пустым, как `undefined`. Длина массива после этого не изменяется.

```
const array = [1, 2, 3];

delete array[1]
array[1];      // undefined
array.length;   // 3

console.log(array);
// [ 1, , 3]
```

Удалять что-то из массива – плохая практика, поэтому нужно создать другой массив без этих элементов. Например, применить фильтрацию.

Метод, которым можно «нормально» удалять элементы – **`splice`**.

Сортировка элементов `.sort()`

Метод `sort()` на месте сортирует элементы массива и возвращает отсортированный массив. Сортировка не обязательно устойчива. Порядок сортировки по умолчанию соответствует порядку кодовых точек Unicode.

```
arr.sort([compareFunction])
```

`compareFunction(a, b)`

Необязательный параметр. Указывает функцию, определяющую порядок сортировки. Если её не указывать, массив сортируется в соответствии со значениями кодовых точек каждого символа Unicode, полученных путём преобразования каждого элемента в строку.

При числовой сортировке, 9 идёт перед 80, но поскольку числа преобразуются в строки, то «80» идёт перед «9» в соответствии с порядком в Unicode.

Числа и слова сортируются по Unicode

```
let fruit = ['арбуз', 'вишня', 'банан'];
fruit.sort();
// [ 'арбуз', 'банан', 'вишня' ]

let numbers = [10, 2, 9, 70, 71];
numbers.sort();
// [ 10, 2, 70, 71, 9 ]
```

Если compareFunction указана, то она принимает на вход 2 элемента: a и b. Если она в результате какой-то операции возвращает:

- 1, то «a» считается меньше, чем «b»
- 0, то «a» считается равным «b»
- 1, то «a» считается больше, чем «b»

Этот пример хорош для понимания:

```
let numbers = [10, 2, 9, 70, 71];

const compare = (a, b) => {
  if (a < b) {
    return -1;
  }
  if (a > b) {
    return 1;
  }
  return 0
};

numbers.sort(compare);
// [ 2, 9, 10, 70, 71 ]
```

Для числового сравнения, функция может просто вычесть b из a:

```
let numbers = [10, 2, 9, 70, 71];

const compare = (a, b) => a - b;

numbers.sort(compare);
// [ 2, 9, 10, 70, 71 ]

numbers.sort((a, b) => -(a - b));
// [ 71, 70, 10, 9, 2 ]
```

Массив объектов может быть отсортирован по любому из своих свойств:

```
let items = [
  { name: 'Edward', value: 21 },
  { name: 'Sharpe', value: 37 },
  { name: 'And', value: 45 }
];

// сортировка по именам
items.sort(foo = (a, b) => {
  if (a.name > b.name) {
    return 1;
  }
  if (a.name < b.name) {
    return -1;
  }
  // a должно быть равным b
  return 0;
});

// сортировка по значениям
items.sort((a, b) => {
  if (a.value > b.value) {
    return 1;
```

```
    }
    if (a.value < b.value) {
        return -1
    }
    return 0;
});
```

Перебор элементов

Тут говорится про цикл `for of`, я это пропущу.

Технически, так как массив является объектом, можно использовать и вариант `for..in`. Но на самом деле это – плохая идея. Существуют скрытые недостатки этого способа:

1. Цикл `for..in` выполняет перебор *всех свойств* объекта, а не только цифровых.

В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые *выглядят, как массив*. То есть, у них есть свойство `length` и индексы, но они также могут иметь дополнительные нечисловые свойства и методы, которые нам обычно не нужны. Тем не менее, цикл `for..in` выведет и их. Поэтому, если нам приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

2. Цикл `for..in` оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее.

Увеличение скорости выполнения может иметь значение только при возникновении узких мест. Но мы всё же должны представлять разницу.

В общем, не следует использовать цикл `for..in` для массивов.

Преобразование к массиву

`Array.from`

создаёт новый экземпляр `Array` из массивоподобного или итерируемого объекта.

К некоторым структурам данных нельзя применять функции высшего порядка типа `filter`, `reduce`, `map`. Поэтому придётся производить их преобразование к массиву.

Документация [здесь](#).

Смотри также `Array.from` в «Перебираемых объектах» ниже. Там будут примеры, которые сложно понять без повторения темы про перебираемые объекты.

```
Array.from(arrayLike[, mapFn[, thisArg]])
```

`arrayLike`

массивоподобный или итерируемый объект, преобразуемый в массив.

`mapFn`

(необязательный) отображающая функция, вызываемая для каждого элемента массива.

`thisArg`

(необязательный) значение, используемое в качестве `this` при выполнении функции `mapFn`.

Пример использования:

```
const map = new Map([[1, 'a'], [2, 'b'], [3, 'c']]);
const array = Array.from(map);
// [ [ 1, 'a' ], [ 2, 'b' ], [ 3, 'c' ] ]
```

```
const mySet = new Set ([1, 2, 3]); // Set { 1, 2, 3 }
const array = Array.from(mySet);
// [ 1, 2, 3 ]
```

`.join`

Соединение элементов массивоподобного объекта.

В следующем примере соединяется массивоподобный объект (список аргументов функции) с использованием вызова Function.prototype.call для Array.prototype.join.

```
function f(a, b, c) {  
    var s = Array.prototype.join.call(arguments);  
    //var s = [].join.call(arguments);  
    console.log(s);  
}  
  
f(1, 'a', true);  
// '1,a,true'
```

...rest

Для преобразования можно использовать ...rest оператор

#? (пример дописать)

Очередь, стек

Очередь – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

- push добавляет элемент в конец.
- shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

Существует и другой вариант применения для массивов – структура данных, называемая стек.

Она поддерживает два вида операций:

- push добавляет элемент в конец.
- pop удаляет последний элемент.

Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху:

Массивы в JavaScript могут работать и как очередь, и как стек. В компьютерных науках структура данных, делающая это возможным, называется двусторонняя очередь.

.push()

Добавляет один или более элементов в конец массива и возвращает новую длину массива.

.pop()

Удаляет последний элемент из массива и возвращает его.

.shift()

Удаляет первый элемент из массива и возвращает его.

.unshift()

Добавляет один или более элементов в начало массива и возвращает новую длину массива.

Эффективность

Методы push/pop выполняются быстро, а методы shift/unshift – медленно.

Просто взять и удалить элемент с номером 0 недостаточно. Нужно также заново пронумеровать остальные элементы.

Операция shift должна выполнить 3 действия:

1. Удалить элемент с индексом 0.
2. Сдвинуть все элементы влево, заново пронумеровать их, заменив 1 на 0, 2 на 1 и т.д.
3. Обновить свойство length .

Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.

То же самое происходит с `unshift`: чтобы добавить элемент в начало массива, нам нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

А что же с `push/pop`? Им не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод `pop` очищает индекс и уменьшает значение `length`.

Метод `pop` не требует перемещения, потому что остальные элементы остаются с теми же индексами. Именно поэтому он выполняется очень быстро. Аналогично работает метод `push`.

Немного о «`length`»

Свойство `length` автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а наибольший цифровой индекс плюс один.

Например, если в массиве один единственный элемент, имеющий индекс 9, он даст длину `.length = 10`.

Ещё один интересный факт о свойстве `length` – его можно перезаписать.

Если мы вручную увеличим его, ничего интересного не произойдёт. Зато, если мы уменьшим его, массив станет короче. Этот процесс необратим, как мы можем понять из примера:

```
let arr = [1, 2, 3, 4, 5];

arr.length = 2; // укорачиваем до двух элементов
console.log( arr ); // [1, 2]

arr.length = 5; // возвращаем length как было
console.log( arr[3] ); // undefined: значения не восстановились
```

Таким образом, самый простой способ очистить массив – это `arr.length = 0;`

Многомерные массивы

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

```
let matrix = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

console.log( matrix[1][1] ); // 5, центральный элемент
```

`toString`

Массивы по-своему реализуют метод `toString`, который возвращает список элементов, разделённых запятыми.

```
let arr = [1, 2, 3];

console.log( arr ); // 1,2,3
console.log( String(arr) === '1,2,3' ); // true
```

Массивы реализуют только преобразование `toString`.

Они не имеют ни `Symbol.toPrimitive`, ни функционирующего `valueOf`.

Таким образом, `[]` становится пустой строкой, `[1]` становится "1", а `[1, 2]` становится "1,2".

```
console.log( [] + 1 );    // "1"
console.log( [1] + 1 );   // "11"
console.log( [1,2] + 1 ); // "1,21"
```

Мутация массивов и функции

Проектируя функции работающие с массивами, есть два пути: менять исходный массив или формировать внутри новый и возвращать его наружу. В подавляющем большинстве стоит предпочитать второй.

В каких же случаях стоит менять сам массив? Есть ровно одна причина по которой так делают – производительность. Именно поэтому некоторые встроенные методы массивов меняют их, например `reverse()` или `sort()`

Обычно в документации каждой функции отдельно подчёркивают, изменяет ли она исходный массив или возвращает результатом новый массив, не модифицируя исходный. Например, метод `concat()`, в отличие от `sort()`, возвращает новый массив.

Копия массива

Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению». В результате мы имеем две независимые переменные.

Объекты ведут себя иначе.

Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него.

Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется. Теперь у нас есть две переменные, каждая из которых содержит ссылку на один и тот же объект.

```
// в примитивах копируются сами объекты
let a = 1;
let b = a;

b += 1;
console.log(a)
// 1

// объекты не копируются, а дублируются ссылки на них
let obj = { key1: 'a', key2: 'b' }
let obj2 = obj;
console.log(obj === obj2);
// true, это один и тот же объект

// массивы: тоже дублируются ссылки
let arr = [1, 2, 3];
let arr2 = arr;
console.log(arr === arr2);
// true, это один и тот же массив
```

Независимая копия массива делается несколькими способами.

- Создать новый массив и запушить в него элементы из старого массива.

Использовать:

- `slice()`
- `...rest`
- `Array.from()`
- `Array.of()`

```
let arr = [1, 2, 3];

let arr2 = arr.slice();
console.log(arr === arr2);
// false
```

```

let arr3 = [...arr];
console.log(arr === arr3);
// false

let arr4 = Array.from(arr);
console.log(arr === arr4);
// false

let arr5 = Array.of(arr);
console.log(arr === arr5);
// false

const makeArrayCopy = (original) => {
  let copy = [];
  original.forEach((element) => copy.push(element));
  return copy;
}

```

Перебираемые (итерируемые) объекты

Перебираемые (или итерируемые) объекты – это концепция, которая позволяет использовать такой объект в цикле `for..of`.

Это не только массивы, но и, например, строки.

[MDN](#). Всё, что записано ниже – это из учебника. Прочитать руководство.

Symbol.iterator

Чтобы создать свой итерируемый объект, нужно встроить в него специальный метод: `Symbol.iterator`.

1. Когда цикл `for..of` запускается, он один раз вызывает метод `Symbol.iterator`.

Если метода нет, то цикл выдаёт ошибку.

2. `Symbol.iterator` возвращает специальный объект с методом `.next: итератор`.

Дальше цикл `for..of` работает только с этим возвращённым объектом.

3. Когда `for..of` хочет получить следующее значение, он вызывает метод `.next()` этого объекта-итератора.

4. Результат вызова `next()` должен иметь вид `{done: Boolean, value: any}`, где `done=true` означает, что итерация закончена, в противном случае `value` содержит очередное значение.

Вот реализация итерируемого объекта `range`:

```

let range = {
  from: 1,
  to: 5
};

// 1
range[Symbol.iterator] = function() {
  return { // 2
    current: this.from,
    last: this.to, // 3
    next() { // 4
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      }
    }
  }
}

```

```
    } else {
      return { done: true };
    }
  };
};

// теперь работает
for (let num of range) {
  console.log(num);
}

// 1, 2, 3, 4, 5
```

Обратите внимание на ключевую особенность итераторов: разделение ответственности. У самого range нет метода next(). Вместо этого другой объект, так называемый «итератор», создаётся вызовом range[Symbol.iterator](), и именно его next() генерирует значения.

Таким образом, итератор отделён от самого итерируемого объекта.

Технически мы можем объединить их и использовать сам range как итератор, чтобы упростить код.

Например, вот так:

```
let range = {
  from: 1,
  to: 5,

  [Symbol.iterator]() {
    this.current = this.from;
    return this;
  },

  next() {
    if (this.current <= this.to) {
      return { done: false, value: this.current++ };
    } else {
      return { done: true };
    }
  }
};
```

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next(), и он запоминает текущее состояние итерации в this.current.

Недостаток такого подхода в том, что теперь мы не можем использовать этот объект в двух параллельных циклах for..of: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах for..of, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

Можно сделать бесконечный итератор. Например, range будет бесконечным при range.to = Infinity. Или мы можем создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

Метод next не имеет ограничений, он может возвращать всё новые и новые значения, это нормально.

Явный вызов итератора

Чтобы понять устройство итераторов чуть глубже, давайте посмотрим, как их использовать явно.

Мы будем перебирать строку точно так же, как цикл for..of, но вручную, прямыми вызовами. Нижеприведённый код получает строковый итератор и берёт из него значения:

```
let str = "Hello";
```

```
// делает то же самое, что и
// for (let char of str) alert(char);

let iterator = str[Symbol.iterator]();

while (true) {
  let result = iterator.next();
  if (result.done) break;
  console.log(result.value); // выводит символы один за другим
}
```

Такое редко бывает необходимо, но это даёт нам больше контроля над процессом, чем `for..of`. Например, мы можем разбить процесс итерации на части: перебрать немного элементов, затем остановиться, сделать что-то ещё и потом продолжить ???

Итерируемые объекты и псевдомассивы

Есть два официальных термина, которые очень похожи, но в то же время сильно различаются. Поэтому убедитесь, что вы как следует поняли их, чтобы избежать путаницы.

Итерируемые объекты – это объекты, которые реализуют метод `Symbol.iterator`, как было описано выше.

Псевдомассивы – это объекты, у которых есть индексы и свойство `length`, то есть, они выглядят как массивы.

При использовании JavaScript можно встретить объекты, которые являются итерируемыми и/или псевдомассивами.

Например, строки итерируемые (для них работает `for..of`) являются псевдомассивами (они индексированы и есть `length`). Но итерируемый объект может не быть псевдомассивом. И наоборот: псевдомассив может не быть итерируемым.

Объект `range` из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и `length`.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

```
let arrayLike = { // есть индексы и свойство length => псевдомассив
  0: "Hello",
  1: "World",
  length: 2
};

// Ошибка (отсутствует Symbol.iterator)
for (let item of arrayLike) {}
```

И итерируемые объекты, и псевдомассивы – это обычно *не массивы*, у них нет методов `push`, `pop` и т.д. Довольно неудобно, если у нас есть такой объект и мы хотим работать с ним как с массивом. Есть универсальный метод [Array.from](#), который принимает итерируемый объект или псевдомассив и делает из него «настоящий» `Array`.

`Array.from`, фишки

`Array.from` принимает объект, проверяет, является ли он итерируемым или псевдомассивом, затем создаёт новый массив и копирует туда все элементы.

`Array.from` с псевдомассивом:

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};
```

```
let arr = Array.from(arrayLike); // (*)
console.log(arr.pop());
// World (метод работает)
```

Array.from с итерируемым объектом:

```
let range = {
  from: 1,
  to: 5
};

range[Symbol.iterator] = function() {
  return {
    current: this.from,
    last: this.to,

    next() {
      if (this.current <= this.last) {
        return { done: false, value: this.current++ };
      } else {
        return { done: true };
      }
    }
  };
};

const arr = Array.from(range);
console.log(arr)
// [ 1, 2, 3, 4, 5 ]
```

Кста, Array.from можно применить к строке и тогда она будет разбита на массив символов.

Это не то же самое, что и str.split, потому что Array.from опирается на итерируемость, поэтому умеет работать с суррогатными парами, а .split – нет:

```
let str = '𠮷𠮷';

// разбивает строку на массив её элементов
let chars = Array.from(str);

console.log(chars[0]); // 𠮷
console.log(chars[1]); // 𠮷
console.log(chars.length); // 2
```

Технически это то же самое, что и for..of, он тоже работает с парами корректно.

Мы можем даже создать slice, который поддерживает суррогатные пары:

```
function mySlice(str, start, end) {
  return Array.from(str).slice(start, end).join(' ');
}

let str = '𠮷𠮷𩿵𩿵';

console.log( mySlice(str, 1, 3) );
// 𩿵𩿵

console.log( str.slice(1, 3) );
// встроенный метод: ❓❓
```

Итого

Объекты, которые можно использовать в цикле for..of, называются *итерируемыми*.

Технически итерируемые объекты должны иметь метод Symbol.iterator.

- Результат вызова obj[Symbol.iterator] называется *итератором*. Он управляет процессом итерации.

- Итератор должен иметь метод next(), который возвращает объект {done: Boolean, value: any}, где done:true сигнализирует об окончании процесса итерации, в противном случае value – следующее значение.

- Метод Symbol.iterator автоматически вызывается циклом for..of, но можно вызвать его и напрямую.

- Встроенные итерируемые объекты, такие как строки или массивы, также реализуют метод Symbol.iterator.

- Строковой итератор знает про суррогатные пары.

Map

Map – это коллекция пар ключ/значение, как и Object, но Map позволяет использовать ключи любого типа, включая функции и объекты.

В отличие от объектов, ключи не приводятся к строкам. [MDN](#).

Все отличия от Объекта (с MDN):

Ключи объекта: строки и символы

Ключи map: любое значение, включая функции, объекты и примитивы.

Ключи в Map упорядочены.

Во время итерации, ключи возвращаются в порядке вставки.

Узнать количество элементов в map можно через свойство size.

Количество элементов Объекта может быть определено только вручную.

Map - итерируемый объект и может быть итерирован

Объект требует ручного получения списка ключей и их итерации.

Объект имеет прототип и поэтому имеет стандартный набор ключей, который, при неосторожности, может пересекаться с вашими ключами. С момента выхода ES5 это может быть изменено с помощью map = Object.create(null).

Map может иметь более высокую производительность в случаях частого добавления или удаления ключей.

```
new Map([iterable])
new Map([[key, value]])
const myMap = new Map([ ["1", "str1"], [1, "num1"] ]);

iterable
Массив или любой другой итерируемый объект, чьими элементами являются пары ключ-значение
(массивы из двух элементов [ [ 1, 'one' ] ]). Все пары ключ-значение будут добавлены в
новый экземпляр Map
```

Map итерируется в порядке вставки его элементов.

Цикл for...of будет возвращать массив [key, value] на каждой итерации.

Чтобы сравнивать ключи, объект Map использует алгоритм [SameValueZero](#). Это почти такое же сравнение, что и ===, при этом NaN считается равным NaN. Так что и NaN также может использоваться в качестве ключа.

Методы и свойства map

map.set(key, value)

записывает по ключу key значение value и возвращает обновлённый объект map, так что мы можем объединить вызовы в цепочку:

```
map.set("1", "str1").set(1, "num1").set(true, "bool1");
```

map.get(key)

возвращает значение по ключу или undefined, если ключ key отсутствует.

map.has(key)

возвращает true, если ключ key присутствует в коллекции, иначе false.

map.delete(key)

удаляет элемент по ключу key.

map.clear()

очищает коллекцию от всех элементов.

map.size

возвращает текущее количество элементов.

map.keys()

map.values()

возвращает итерируемый объект [Map Iterator] по ключам

возвращает итерируемый объект [Map Iterator] по значениям

```
const myMap = new Map([["1", "str1"], [1, "num1"], [true, "bool1"]]);

myMap.keys()
// [Map Iterator] { '1', 1, true }

console.log(myMap.values())
// [Map Iterator] { 'str1', 'num1', 'bool1' }
```

map.entries()

возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

```
const myMap = new Map([["1", "str1"], [1, "num1"], [true, "bool1"]]);

console.log(myMap.entries())
// [Map Entries] { [ '1', 'str1' ], [ 1, 'num1' ], [ true, 'bool1' ] }
```

.forEach

имеется.

Перебор Map

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов.

Для перебора коллекции Map есть 3 метода:

map.keys() – возвращает итерируемый объект по ключам,

map.values() – возвращает итерируемый объект по значениям,

`map.entries()` – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в `for..of`.
`.forEach` – имеется.

```
const myMap = new Map();
myMap.set("1", "str1").set(1, "num1").set(true, "bool1");

for(let i of myMap) {
  console.log(i)
}

// [ '1', 'str1' ]
// [ 1, 'num1' ]
// [ true, 'bool1' ]
```

Мар, конвертация

При создании Мар мы можем указать массив (или другой итерируемый объект) с парами ключ-значение:

```
const arr = [ [ '1', 'str1' ], [ 1, 'num1' ], [ true, 'bool1' ] ];

let map = new Map(arr);

console.log( map );
// Map { '1' => 'str1', 1 => 'num1', true => 'bool1' }
```

Мар из объекта.

Для этого надо сначала перегнать объект в массив массивов ключ-значение с помощью метода `Object.entries(obj)`, а потом сделать из этого массива мар:

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));
```

Объект из мар

Можно сделать с помощью метода объекта `Object.fromEntries` и метода `map.entries`:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);

let obj = Object.fromEntries(map.entries());

// можно вообще просто так сделать
let obj = Object.fromEntries(map);
```

Это то же самое, так как `Object.fromEntries` ожидает перебираемый объект в качестве аргумента, не обязательно массив. А перебор мар как раз возвращает пары ключ/значение, так же, как и `map.entries()`. Так что в итоге у нас будет обычный объект с теми же ключами/значениями, что и в мар.

WeakMap

Документация [здесь](#).

WeakMap – это Мар-подобная коллекция пар ключ-значение, позволяющая использовать в качестве ключей только объекты, а значения могут быть произвольных типов. Пары автоматически удаляются, как только ключи (объекты) становятся недоступны иными путями.

Если мы используем объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удален из памяти (и из объекта WeakMap) автоматически.

WeakMap не поддерживает перебор. Также, нет способа взять все ключи или значения из него, потому что соответствующих методов тоже нет.

Методы WeakMap

В WeakMap присутствуют только следующие методы:

```
weakMap.get(key)  
weakMap.set(key, value)  
weakMap.delete(key)  
weakMap.has(key)
```

отсутствуют:

```
weakMap.keys()  
weakMap.values()  
weakMap.entries()
```

Где использовать?

WeakMap и WeakSet используются как вспомогательные структуры данных в дополнение к «основному» месту хранения объекта. Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в WeakMap или в WeakSet, то он будет удален автоматически.

Если мы работаем с объектом, который «принадлежит» другому коду, может быть даже сторонней библиотеке, и хотим сохранить у себя какие-то данные для него, которые должны существовать лишь пока существует этот объект, то WeakMap – то, что нужно.

Мы кладём эти данные в WeakMap, используя объект как ключ, и когда сборщик мусора удалит объекты из памяти, ассоциированные с ними данные тоже автоматически исчезнут.

Давайте рассмотрим один пример.

Предположим, у нас есть код, который ведёт учёт посещений для пользователей. Информация хранится в коллекции Map: объект, представляющий пользователя, является ключом, а количество визитов – значением. Когда пользователь нас покидает (его объект удаляется сборщиком мусора), то больше нет смысла хранить соответствующий счётчик посещений.

Вот пример реализации счётчика посещений с использованием WeakMap:

```
// visitsCount.js  
let visitsCountMap = new WeakMap(); // map: пользователь => число визитов  
  
// увеличиваем счётчик  
function countUser(user) {  
    let count = visitsCountMap.get(user) || 0;  
    visitsCountMap.set(user, count + 1);  
}  
  
// main.js  
let john = { name: "John" };  
countUser(john); //ведём подсчёт посещений  
  
// пользователь покинул нас
```

```
john = null;
```

Применение для кеширования

Другая частая сфера применения – это кеширование, когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его.

Если мы будем использовать WeakMap вместо Map, то закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

```
// cache.js
let cache = new WeakMap();

// вычисляем и запоминаем результат
function process(obj) {
  if (!cache.has(obj)) {
    let result = /* вычисляем результат для объекта */ obj;

    cache.set(obj, result);
  }

  return cache.get(obj);
}

// main.js
let obj = {/* какой-то объект */};

let result1 = process(obj);
let result2 = process(obj);

// ...позже, когда объект больше не нужен:
obj = null;

// Нет возможности получить cache.size, так как это WeakMap,
// но он равен 0 или скоро будет равен 0
// Когда сборщик мусора удаляет obj, связанные с ним данные из кеша тоже удаляются
```

Set

Это множество: массив уникальных элементов.

Объекты Set позволяют сохранять уникальные значения любого типа: как [примитивы](#), так и другие типы объектов.

[MDN](#)

```
new Set([iterable]);
```

При передаче итерируемого объекта, все его элементы будут добавлены в новый Set. Иначе (или при null) новый Set будет пуст.

Обход выполняется в порядке вставки элементов.

Сравнение значений – на алгоритме строгое равенство «==», NaN равно NaN.

Можно перебрать с помощью цикла for...of или метода .forEach.

Альтернативой множеству Set может выступать массив и дополнительный код для проверки уже имеющегося элемента с помощью [arr.find](#). Но в этом случае будет хуже производительность. Множество Set лучше оптимизировано для добавлений, оно автоматически проверяет элемент на уникальность.

Методы

Все методы set можно посмотреть в [Set.prototype](#).

`set.add(value)`

добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.

`set.delete(value)`

удаляет значение, возвращает true если value было в множестве на момент вызова, иначе false.

`set.has(value)`

возвращает true, если значение присутствует в множестве, иначе false.

`set.clear()`

удаляет все имеющиеся значения.

`set.size`

возвращает количество элементов в множестве.

[.entries\(\)](#)

Метод возвращает итератор [Set Entries], который содержит массивы [значение, значение] для каждого элемента в объекте Set в порядке их добавления.

Для объекта Set не существует ключа key. Тем не менее, чтобы API было схож с объектом Map, каждая запись содержит значение как в ключе, так и в значении.

```
const mySet = new Set([1, 2, 3, 4]);
console.log(mySet.entries());
// [Set Entries] { [ 1, 1 ], [ 2, 2 ], [ 3, 3 ], [ 4, 4 ] }
```

[.values\(\)](#)

`.keys`

Метод values() возвращает новый Итератор, который содержит значения для каждого элемента в объекте Set в порядке их добавления.

Метод keys() является синонимом этого метода (для схожести с объектами Map). Он ведёт себя точно так же и возвращает значения элементов Set.

```
const mySet = new Set([1, 2, 3, 4]);
mySet.keys();
// [Set Iterator] { 1, 2, 3, 4 }
```

[.forEach\(\)](#)

Метод выполняет функцию по одному разу для каждого элемента из Set в порядке их расположения. callback вызывается с тремя аргументами:

- значение элемента
- ключ элемента
- Set объект обхода

В объектах типа Set нет ключей, поэтому оба первых аргумента принимают значение содержащееся в Set. Это делает метод forEach() для объекта Set совместимым с методами forEach() других объектов, таких как Map и Array.

Это действительно так, значение появляется в списке аргументов дважды.

Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит немного странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот.

WeakSet

Аналогичен Set, но добавлять в WeakSet можно только объекты (не примитивные значения).

Объект присутствует в множестве только до тех пор, пока доступен где-то ещё, или сборщик мусора удалит его.

Объект WeakSet - коллекция, элементами которой могут быть только объекты. Ссылки на эти объекты в WeakSet являются слабыми. Каждый объект может быть добавлен в WeakSet только один раз.

Документация [здесь](#).

Не перебираемый.

```
new WeakSet([iterable]);
```

iterable

При передаче итерируемого объекта, все его элементы будут добавлены в новый WeakSet. Null обрабатывается как undefined.

Главные отличия от объекта [Set](#):

- WeakSet **содержит только объекты**, тогда как Set - значения любого типа.

- Ссылки на объекты в WeakSet являются слабыми: если на объект, хранимый в WeakSet нет ни одной внешней ссылки, то сборщик мусора удалит этот объект. Также это означает, что WeakSet **не итерируем**, так как нет возможности получить список текущих хранимых в WeakSet объектов.

Методы WeakSet

WeakSet.add(value)

добавляет новый объект в конец объекта WeakSet. Добавлять можно только объекты.

WeakSet.has()

Определяет, содержит WeakSet объект value или нет, возвращая, соответственно, true или false.

WeakSet.delete()

Удаляет из WeakSet элемент value.

.size - отсутствует

.keys - отсутствует

Weak объекты, назначение

WeakMap и WeakSet используются как вспомогательные структуры данных в дополнение к «основному» месту хранения объекта. Если объект удаляется из основного хранилища и нигде не используется, кроме как в качестве ключа в WeakMap или в WeakSet, то он будет удалён автоматически.

Служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве WeakSet может что-то сказать нам об объекте.

Например, мы можем добавлять пользователей в WeakSet для учёта уникальных посещений сайта:

```
let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
```

```
let mary = { name: "Mary" };

visitedSet.add(john); // John заходил к нам
visitedSet.add(pete); // потом Pete
visitedSet.add(john); // John снова

// visitedSet сейчас содержит двух пользователей
// проверим, заходил ли John
alert(visitedSet.has(john)); // true

// проверим, заходила ли Mary
alert(visitedSet.has(mary)); // false

john = null;
// структура данных visitedSet будет очищена автоматически
```

Object.keys, values, entries

map.keys(), map.values(), map.entries() – это универсальные методы, и существует общее соглашение использовать их для структур данных. Если бы мы делали собственную структуру данных, нам также следовало бы их реализовать.

Методы поддерживаются для структур:

Object
Map
Set
Array

Date

Объект [Date](#) содержит дату и время, а также предоставляет методы управления ими.

`new Date()`

Для создания нового объекта Date нужно вызвать конструктор `new Date()`.

Если конструктор вызван без аргументов, то будет создан объект Date для текущих даты и времени, согласно системным настройкам.

В конструктор можно передавать аргументы, из которых будет сделан объект.

Важно!

При создании объект Date будет считать указанное время как UTC.

При использовании геттеров автоматически возвращать поправку на местное время.

Это значит, что если создаёшь дату «2020-01-01 00:00», то геттер вернёт «2019-12-31 21:00»

```
let now = new Date();
// 2020-05-23T17:13:14.255Z

new Date(timestamp)
// milliseconds

new Date("YYYY-MM-DDTHH:mm:ss.sss-Z")
// datestring ('2012-01-26T13:51:50.417-07:00');

new Date(year, month, date, hours, minutes, seconds, ms)
// отдельные значения

Date.now()
// вернёт timestamp
```

```
Date.parse('2012-01-26T13:51:50.417-07:00')
// вернёт timestamp
```

Требования к аргументам:

year

должен состоять из четырёх цифр

month

начинается с 0 (январь) по 11 (декабрь).

date

день месяца, начинается с 1.

Если указать 0, то месяц отматывается на 1 назад, а день становится последним днём предыдущего месяца.

hours/minutes/seconds/ms

Если отсутствуют, их значением становится 0.

- если нужно сделать независимую копию объекта, то в конструктор можно передать уже существующий объект даты: new Date(date);

- timestamp – количество миллисекунд, прошедших с 1 января 1970 года. Это легковесное численное представление даты. Миллисекунда – это 1/1000 секунды.

- Прочитать дату из строки. Алгоритм разбора такой же, как в Date.parse ([MDN](#)).

Если время не указано, то оно ставится в полночь по Гринвичу и меняется в соответствии с часовым поясом места выполнения кода.

- Если передано больше 1 числа, то объект создаётся с заданными компонентами в местном часовом поясе. Обязательны только первые два аргумента.

Date.now()

Возвращает текущий timestamp времени. Не путать с конструктором «new Date()»!

Семантически эквивалентен new Date().getTime(), но работает быстрее, т.к. не создаёт промежуточный объект. Его можно использовать в бенчмарках, где замеряется время работы кода.

```
Date.now(); // 1598739411224
```

Геттеры

.getFullYear()

Получить год (4 цифры)

.getMonth()

Получить месяц, от 0 до 11.

Важно! Счёт месяцев начинается с нуля. Январь – это месяц номер 0.

.getDate()

Получить день месяца, от 1 до 31, что несколько противоречит названию метода.

Важно! Если в качестве дня прописать 0, то месяц минусуется назад и получится последний день предыдущего месяца.

.getDay()

Вернуть день недели.

Важно! Дни недели считаются с нуля. Первый день недели – воскресенье, а не понедельник. Таким образом:

- 0 – воскресенье
- 1 – понедельник
- 2 – вторник
- 3 – среда
- 4 – четверг
- 5 – пятница
- 6 – суббота

.getHours(), .getMinutes(), .getSeconds(), .getMilliseconds()

Получить, соответственно, часы, минуты, секунды или миллисекунды. Возвращается тип Number.

.getTime()

Для заданной даты возвращает таймстамп – количество миллисекунд, прошедших с 1 января 1970 года UTC+0.

.getTimezoneOffset()

Возвращает разницу в минутах между местным часовым поясом и UTC.

```
// если вы в часовом поясе UTC-1, то выводится 60  
// если вы в часовом поясе UTC+3, выводится -180  
alert( new Date().getTimezoneOffset() );
```

UTC

Все вышеперечисленные методы возвращают значения в соответствии с местным часовым поясом.

Однако существуют и их UTC-варианты, возвращающие день, месяц, год для временной зоны UTC+0:

[getUTCFullYear\(\)](#), [getUTCMonth\(\)](#), [getUTCDay\(\)](#). Для их использования требуется после "get" подставить "UTC".

Два особых метода без UTC-варианта:

`getTime()`

`getTimezoneOffset()`.

```
// текущая дата  
let date = new Date();  
  
// час в вашем текущем часовом поясе  
alert( date.getHours() );  
  
// час в часовом поясе UTC+0 (лондонское время без перехода на летнее время)  
alert( date.getUTCHours() );
```

Сеттеры

Следующие методы позволяют установить компоненты даты и времени:

`.setFullYear(year, [month], [date])`

год

`.setMonth(month, [date])`

месяц

Важно! Счёт месяцев начинается с нуля. Январь – это месяц номер 0.

```
. setDate(date)
```

День

Важно! Если в качестве дня прописать 0, то месяц минусанётся назад и получится последний день предыдущего месяца.

```
. setHours(hour, [min], [sec], [ms])  
. setMinutes(min, [sec], [ms])  
. setSeconds(sec, [ms])  
. setMilliseconds(ms)
```

```
. setTime(milliseconds)
```

Устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC.

Работает это так:

```
let a = new Date(2020, 4, 1, 10);  
console.log(a)  
// 2020-05-01T07:00:00  
  
// накинуть 2 часа к дате  
a.setHours( a.getHours() + 2 )  
console.log(a)  
// 2020-05-01T09:00:00
```

У всех этих методов, кроме setTime(), есть **UTC**-вариант, например: setUTCHours().

Некоторые методы могут устанавливать сразу несколько компонентов даты, например: setHours.
Если какая-то компонента не указана, она не меняется.

Автоисправление даты

Объект Date самостоятельно корректируется при введении значений, выходящих за рамки допустимых. Это полезно для сложения/вычитания дней/месяцев/недель. Например, если записать 61 минуту, то лишнее посчитается в час. Високосный год тоже считается сам.

Если в качестве дня прописать 0, то месяц минусанётся назад и получится последний день предыдущего месяца.

Эту возможность часто используют, чтобы получить дату по прошествии заданного отрезка времени. Например, получим дату спустя 70 секунд с текущего момента:

```
let date = new Date();  
date.setSeconds(date.getSeconds() + 70);  
  
console.log( date );  
// выводит правильную дату, которая будет через 70 секунд
```

Преобразование к числу, вычитание дат

Если объект Date преобразовать в число (+Number), то получим таймстамп по аналогии с date.getTime().

Важный побочный эффект:

Если один объект Date вычесть из другого, то получится разность в миллисекундах.

Этот приём можно использовать для измерения времени между событиями:

```
let start = new Date();  
  
// выполняем некоторые действия  
for (let i = 0; i < 100000; i++) {  
  let doSomething = i * i * i;  
}
```

```
let end = new Date();

console.log(`Цикл отработал за ${end - start} миллисекунд`);
// Цикл отработал за 3 миллисекунд
```

Бенчмаркинг

Интересный раздел, как замерять производительность работы функций. Переписывать сюда не буду, лучше перейти и почитать.

Если кратко, то Date2.getTime() – Date1.getTime() быстрее, чем Date2() – Date1()

Порой нам нужно измерить время с большей точностью. Собственными средствами JavaScript измерять время в микросекундах (одна миллионная секунды) нельзя, но в большинстве сред такая возможность есть.

К примеру, в браузерах есть метод [performance.now\(\)](#), возвращающий количество миллисекунд с начала загрузки страницы с точностью до микросекунд (3 цифры после точки):

```
alert(`Загрузка началась ${performance.now()}мс назад`);
```

Получаем что-то вроде: "Загрузка началась 34731.26000000001мс назад"

.26 – это микросекунды (260 микросекунд)

корректными являются только первые три цифры после точки, а остальные -- это ошибка точности

В Node.js для этого предусмотрен модуль `microtime` и ряд других способов. Технически почти любое устройство или среда позволяет добиться большей точности, просто её нет в объекте `Date`.

Date.parse(str), парсинг строк

Считывает дату из строки.

Возвращает таймстамп или NaN.

[MDN](#).

Формат строки должен быть таким:

```
YYYY-MM-DDTHH:mm:ss.sss-Z
```

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');
console.log(ms);
// 132761110417 (таймстамп)
```

Обязательный параметр – только год YYYY.

YYYY-MM-DD – это год-месяц-день.

Символ "T" используется в качестве разделителя.

HH:mm:ss.sss – часы, минуты, секунды и миллисекунды.

'Z' обозначает часовой пояс в формате +-hh:mm. Если указать просто букву Z, то получим UTC+0.

Можно тут же создать объект `new Date` из таймстампа:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );
alert(date);
```

Формат JSON

Часто бывает нужно преобразовать объект в строку, чтобы отправить или передать его куда-то.

[JSON](#) (JavaScript Object Notation) – это общий формат для представления значений и объектов.

Документация [здесь](#).

Кроме своих двух методов он не содержит никакой интересной функциональности.

Ограничения

JSON сериализует **следующие типы данных:**

- Объекты { ... }
- Массивы [...]

Примитивы:

- строки,
- числа,
- логические значения true/false,
- null.

Например:

```
// число в JSON остаётся числом
console.log( JSON.stringify(1) ) // 1

// строка в JSON по-прежнему остаётся строкой, но в двойных кавычках
console.log( JSON.stringify('test') ) // "test"

console.log( JSON.stringify(true) ); // true

console.log( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

Пропуск значений

JSON не сериализует некоторые специфические свойства объектов:

- Методы объекта (функции)
- Символьные свойства
- Свойства, содержащие значение undefined

```
let user = {
  // метод
  sayHi() {
    console.log("Hello");
  },
  // символ
  [Symbol("id")]: 123,
  // undefined
  something: undefined
};

console.log( JSON.stringify(user) );
// {} (пустой объект)
```

Циклические ссылки приводят к ошибке.

Здесь преобразование завершается неудачно из-за циклической ссылки: room.occupiedBy ссылается на meetup, и meetup.place ссылается на room

```
let room = {
  number: 23
};
```

```
let meetup = {  
    title: "Conference",  
    participants: ["john", "ann"]  
};  
  
meetup.place = room; // meetup ссылается на room  
room.occupiedBy = meetup; // room ссылается на meetup  
  
JSON.stringify(meetup); // Ошибка: Преобразование циклической структуры в JSON
```

Устройство сериализованного объекта:

Объекты и массивы:

- имена свойств должны быть строками, заключёнными в "двойные кавычки";
- конечные (висячие) запятые запрещены.

Числа:

- ведущие нули запрещены;
- перед десятичной запятой обязательно должна быть хотя бы одна цифра.

Строки:

- только в "двойных кавычках";
- может быть заэкранирован только ограниченный набор символов;
- некоторые управляющие символы запрещены;
- разрешены юникодные символы разделительной линии ([U+2028](#)) и разделительного параграфа ([U+2029](#)).

JSON не поддерживает комментарии, добавление комментария в JSON делает его недействительным.

Существует ещё один формат [JSON5](#), который поддерживает ключи без кавычек, комментарии и т.д. Но это самостоятельная библиотека, а не спецификация языка.

Методы JSON

JSON.stringify()

Метод преобразует значение JavaScript в строку JSON и возвращает её.

Такая строка называется *JSON-форматированным* или *сериализованным* объектом.

```
JSON.stringify(value[, replacer[, space]])
```

```
JSON.stringify(myObject, null, 2)
```

value

Значение, преобразуемое в строку JSON.

replacer

Необязательный. Функция(key, value), которая преобразует значения и свойства по ходу их преобразования в строку. Или массив с перечнем свойств, которые будут сериализованы.

space

Необязательный. Делает результат нормально отформатированным (расставляя пробелы). Можно указать число (пробелов) или строку-символ, которая будет ставиться вместо пробелов.

Подробное описание функции replacer даётся в статье [Использование родного объекта JSON](#) руководства по JavaScript.

JSON.parse()

Декодирует строку JSON и возвращает «собранный» объект.

Выбрасывает исключение SyntaxError, если разбираемая строка является некорректным JSON.

```
JSON.parse(text[, foo(key, value) ])
```

text

Разбираемая строка JSON. Смотрите документацию по объекту JSON для описания синтаксиса JSON.

reviver

Необязательный. Функция, которая преобразует каждое разбираемое значения.

obj.toJSON

У объектов есть свойство `.valueOf` для преобразования к примитиву и `.toString` для преобразования в строку. Аналогично, у объектов есть свойство `.toJSON`, которое определяет, как объект сконвертируется в json, и его тоже можно переназначать.

Свойство `.toJSON` должно быть именно методом, который что-то возвратит. Если в качестве значения свойства написать так, то будет ошибка: `toJSON = 42`.

Вот пример с объектом Date. Когда перегоняешь его в json, он выглядит как читаемая строка с датой:

```
const date = new Date();
const a = JSON.stringify(date);

console.log(a);
// "2020-06-05T12:21:09.951Z"
```

Пример самостоятельной реализации `.toJson` в своих объектах:

```
const myObject = {
  key1: 'value 1',
  key2: 'value 2',

  toJSON: function() {
    return 42
  }
};

console.log( JSON.stringify(myObject, null, 1) );
// 42
```

Пример с Date

При попытке использовать свойство объекта Date получаем ошибку, потому что после конвертации date стал строкой, а не объектом:

```
const myObject = {
  date: new Date(0, 0)
};

const a = JSON.stringify(myObject);
// { "date": "1899-12-31T21:29:43.000Z" }

const b = JSON.parse(a);
// { date: '1899-12-31T21:29:43.000Z' }

console.log(b.date.getHours())
// TypeError: b.date.getHours is not a function
```

Reviver с соответствующей функцией поправит ситуацию. Функция примитивная: как только встречает ключ date, создаёт из её значения объект date.

Всё работает хорошо:

```
...
const foo = function(key, value) {
  if (key === 'date') {
    return new Date(value);
  }
  return value;
};

const b = JSON.parse(a, foo);

console.log(b.date.getHours())
// 0
```

Symbol

Документация [здесь](#).

Символ (Symbol) — это уникальный и неизменяемый тип данных, который может быть использован как идентификатор для свойств объектов.

Символьный объект (symbol object) — это объект-обёртка (англ. wrapper) для [примитивного](#) символьного типа.

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки или символы. «Символ» представляет собой уникальный идентификатор.

Создание

Создаются новые символы с помощью функции `Symbol()`

Для создания символов в глобальной области — `Symbol.for()`

```
let id = Symbol([описание])
let id = Symbol.for([описание])
```

Описание — необязательный параметр-строка. Это описание символа, которое может быть использовано во время отладки, но не для доступа к самому символу. Описание также называют именем символа.

Доступ к символу можно получить через переменную, куда он записан.

```
const sym = Symbol("тут описание символа");
console.log(sym)
// Symbol(это описание символа)
```

Особенности

Символы гарантированно уникальны.

Даже если мы создадим множество символов с одинаковым описанием, это всё равно будут разные символы.

Описание — это просто метка для удобства, которая ни на что не влияет.

Это как равенство объектов. Под каждый символ выделяется отдельное место в памяти. Чтобы они были равны, ссылки должны указывать на одно и то же место в памяти.

В примере создаются два символа с одинаковым описанием и они неравны друг другу:

```
const sym2 = Symbol("foo");
```

```
const sym3 = Symbol("foo");

sym2 === sym3
// false
```

Символы автоматически не преобразуются в строку.

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция alert любое значение автоматически преобразовывает в строку, а затем выводит. С символами будет ошибка.

Это – «защита» от путаницы. Строки и символы – принципиально разные типы данных и не должны неконтролируемо преобразовываться друг в друга.

Преобразовать символ в строку **можно только явно** с помощью метода .toString():

```
const sym = Symbol("описание");
const stringSymbol = sym.toString()

console.log( stringSymbol )
console.log( typeof stringSymbol)

// Symbol(описание)
// string
```

Чтобы получить только описание символа в виде строки, можно использовать встроенный метод .description:

```
const sym = Symbol("foo");

console.log( sym.description )
// foo
```

Использование

Символы имеют два основных варианта использования:

1. «скрытые» свойства объектов.

Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в for...in, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования. Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, что другие видеть не должны.

2. Системные символы

Существует множество системных символов, используемых внутри JavaScript, доступных как Symbol.*. Мы можем использовать их, чтобы изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать Symbol.iterator для [итераторов](#), Symbol.toPrimitive для настройки [преобразования объектов в примитивы](#) и так далее.

Символы как свойства объектов

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки или символы. «Символ» представляет собой уникальный идентификатор.

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы.

С символами не работает следующее:

- не перечисляются при итерации for...in
- не выводятся при Object.keys.
- Object.getOwnPropertyNames() не вернет символьные свойства объекта.

Это – часть общего принципа «сокрытия символьных свойств». Если другая библиотека или скрипт будут работать с нашим объектом, то при переборе они не получат ненароком наше символьное свойство.

С символами работают:

`Object.getOwnPropertySymbols()`.

Метод позволяет получить символы объекта.

Object.assign

в отличие от цикла `for..in`, скопирует и строковые, и символьные свойства.

Идея заключается в том, что, когда мы клонируем или объединяем объекты, мы обычно хотим скопировать все свойства, в т.ч. символы.

Запись символов в объект

Записать свойство объекта через символ можно через квадратные скобки (не через точку).

Это вызвано тем, что нам нужно использовать значение переменной в качестве ключа, а не строку «`sym`»:

```
const obj = {
  key1: 'value',
}

const sym = Symbol("foo");
obj[sym] = 'value2'

console.log(obj)
// { key1: 'value', [Symbol(foo)]: 'value2' }
```

То же самое при литеральном объявлении объекта `{ . . . }`: символ необходимо заключить в квадратные скобки:

```
const sym1 = Symbol('символ 1')

const obj = {
  key1: 'value',
  [sym1]: 'value2'
}

const sym2 = Symbol("символ 2");
obj[sym2] = 'value2'

console.log(obj)
// {
//   key1: 'value',
//   [Symbol(символ 1)]: 'value2',
//   [Symbol(символ 2)]: 'value2'
// }
```

При записи свойств с помощью символов, такие свойства невозможно перезаписать (можно). Могут быть 2 одинаковые строки, но нет одинаковых символов.

Глобальный реестр символов

Обычно все символы уникальны, даже если их имена совпадают. Но иногда мы наоборот хотим, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части нашего приложения хотят получить доступ к символу "id", подразумевая именно одно и то же свойство.

Когда нужно обратиться к одному и тому же символу, используется глобальный реестр символов.

Мы можем создавать в нём символы и обращаться к ним позже, и при каждом обращении нам гарантированно будет возвращаться один и тот же символ.

Глобальный реестр символов — это просто удобный глобальный репозиторий для экземпляров символов. Вы можете реализовать его самостоятельно, если хотите, но наличие такого встроенного хранилища означает, что среда выполнения может использовать его как место для публикации экземпляров символов, имеющих особое значение для данного контекста.

В вашем собственном приложении вы можете решить, что некоторые типы объектов будут иметь определенные свойства, доступные через некоторый символ. Весь ваш код может найти эти символы через `Symbol.for()`.

Сделав реестр частью среды выполнения, среда, такая как Node.js, может использовать механизм символов для расширения объектов, не опасаясь вызвать проблемы для устаревшего кода. Например, если Node хотел сделать так, чтобы вы могли узнать, сколько памяти использует объект, они могут придумать символ, поместить его в реестр и задокументировать раздел реестра. Тогда любой код мог бы использовать это

Символы, содержащиеся в реестре, называются **глобальными символами**.

Базовый синтаксис, использующий функцию `Symbol()`, создаст локальный, но не глобальный символ.

Для создания символов, доступных во всех файлах и в окружении (глобальной области), используйте методы `Symbol.for()` и `Symbol.keyFor()`, чтобы задать или получить символ из глобального символьного реестра.

См. описание и работу этих методов ниже.

`Symbol.for(key)`

Для чтения (или, при отсутствии, создания) символа из глобального реестра.

Ищет существующий символ по заданному ключу (имени, идентификатору, описанию) и возвращает его, если он найден.

Если символа с таким описанием в реестре нет, то создаст новый символ для данного ключа в глобальном реестре символов.

Как я понял, разница с обычными символами в том, что обычные сохраняются в переменные, а глобальные — записываются в реестр и для их «выгрузки» из общего реестра используется описание, которое именуется ключом: `key`.

Метод сначала проверяет, существует ли символ с заданным идентификатором в реестре — и возвращает его, если тот присутствует. Если символ с заданным ключом не найден, то создаст новый глобальный символ.

```
Symbol.for("foo"); // создаёт новый глобальный символ
Symbol.for("foo"); // возвращает символ, созданный прежде
```

```
// Однаковыми могут глобальные символы, но не локальные
Symbol.for("bar") === Symbol.for("bar"); // true
Symbol("bar") === Symbol("bar"); // false
```

```
// Идентификатор также используется в качестве описания
var sym = Symbol.for("mario");
sym.toString(); // "Symbol(mario)"
```

Глобальные символы могут быть одинаковыми, см. код выше. (я думаю, что они не одинаковые, это просто несколько переменных ссылаются на один и тот же символ).

Чтобы предотвратить конфликт имён ваших глобальных символов и глобальных символов из других библиотек, может оказаться неплохой идеей использование префиксов:

```
Symbol.for("mdn.foo");
Symbol.for("mdn.bar");
```

Symbol.keyFor(sym)

Только для глобальных символов. Если символ неглобальный, метод не сможет его найти в реестре и вернёт `undefined`.

Передаёшь сюда глобальный символ (переменную, в которой хранится), возвращается его описание (имя, ключ). Если в глобальном реестре символов такого нет, то вернёт [undefined](#).

```
const globalSym = Symbol.for('foo');
// сначала создаёшь

console.log( Symbol.keyFor(globalSym) );
// получаешь "foo"
```

Системные (известные) символы

Существует множество «системных» символов, использующихся внутри самого JavaScript, и мы можем использовать их, чтобы настраивать различные аспекты поведения объектов. Эти символы перечислены в спецификации в таблице [Well-known symbols](#).

Symbol.iterator

Метод, возвращающий итератор по умолчанию для объекта. Используется конструкцией [for...of](#).

С ним можно создавать собственные итерируемые объекты. Он есть, например, у массива и строки.

См. «[перебираемые \(итерируемые\) объекты](#)».

Symbol.hasInstance

Метод, определяющий, распознает ли конструктор некоторый объект как свой экземпляр. Используется оператором [instanceof](#).

Symbol.species

`Symbol.species` — известный символ, позволяющий определить конструктор, использующийся для создания порождённых объектов. Свойство `Symbol.species`, содержащее аксессор (геттер), позволяет подклассам переопределить конструктор, используемый по умолчанию для создания новых объектов.

Допустим, встроенный класс `Array` надо расширить и мы пишем дочерний класс `MyArray`. Если к экземплярам `MyArray` применять функции типа `.map` или `.filter`, которые возвращают отфильтрованный массив, то, используя их на экземплярах `MyArray` они будут возвращать тоже экземпляры `MyArray`. Это поведение задано по умолчанию.

Если понадобиться возвращать объекты типа [Array](#) в методах дочернего класса `MyArray`, то символ `Symbol.species` позволит это сделать.

Другие коллекции, такие как `Map`, `Set`, работают аналогично: они также используют `Symbol.species`.

Чтобы при использовании `map`, `filter` и т.д. на инстансах `MyArray` возвращался экземпляр обычного массива (а не `MyArray`), надо записать символ `Symbol.species` так:

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }

  // перегружаем species для использования родительского конструктора Array
  static get [Symbol.species]() {
    return Array;
  }
}
```

Symbol.toPrimitive

Метод, преобразующий объект в примитив (примитивное значение).

`Symbol.toPrimitive` описывает свойство объекта как функцию, которая вызывается при преобразовании объекта в соответствующее примитивное значение.

Эта тема тесно связана с хинтами, про которые написано в теме про объекты.

С помощью свойства `Symbol.toPrimitive` (которое описывается как функция), объект может быть приведен к примитивному типу. Функция вызывается со строковым аргументом `hint`, который передает желаемый тип примитива. Значением аргумента `hint` может быть одно из следующих значений: `"number"`, `"string"`, и `"default"`.

```
// Объект со свойством Symbol.toPrimitive
var obj2 = {
  [Symbol.toPrimitive](hint) {
    if (hint == 'number') {
      return 777;
    }
    if (hint == 'string') {
      return 'hello!';
    }
    return true;
  }
};

console.log(Number(obj2));      // 10          -- желаемый тип (hint) - "number"
console.log(String(obj2));     // "hello"     -- желаемый тип (hint) - "string"
console.log(obj2 + '');        // "true"      -- желаемый тип (hint) - "default"

// Объект без свойства Symbol.toPrimitive
console.log(Number({}));      // NaN
console.log(String({}));       // "[object Object]"
console.log({} + '');         // "[object Object]"
```

В этом примере указано, как конвертировать объект в число:

```
const object1 = {
  [Symbol.toPrimitive](hint) {
    if (hint === 'number') {
      return 42;
    }
    return null;
  }
};

console.log(+object1);
// 42
```

toString/valueOf

Методы `toString` и `valueOf` берут своё начало с древних времён. Они не символы, так как в то время символов ещё не существовало, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода `Symbol.toPrimitive`, движок JavaScript пытается найти эти методы и вызвать их следующим образом:

`toString` -> `valueOf` для хинта со значением «`string`».

`valueOf` -> `toString` – в ином случае.

Получится то же поведение, что и с `Symbol.toPrimitive`.

Довольно часто мы хотим описать одно «универсальное» преобразование объекта к примитиву для всех ситуаций. Для этого достаточно создать один `toString` отсутствие `Symbol.toPrimitive` и `valueOf`, `toString` обработает все случаи преобразований к примитивам.

Symbol.toStringTag

Строковое значение, используемое в качестве описания объекта по умолчанию. Используется функцией Object.prototype.toString().

При создании собственного класса JavaScript по умолчанию использует тег "Object". Т.е. ты создаёшь какой-то класс и при приведении его к строке будет тег [object Object].

С помощью toStringTag можно установить свой собственный тег для класса:

```
class MyClass {
  get [Symbol.toStringTag]() {
    return 'this is MyClass';
  }
}

const a = new MyClass();
console.log(a.toString());
// [object this is MyClass]
```

Аналогично – для отдельного объекта:

```
let user = {
  [Symbol.toStringTag]: "User"
};

console.log(user.toString()); // [object User]
```

Многие Javascript типы имеют теги по умолчанию:

```
Object.prototype.toString.call('str');      // "[object String]"
Object.prototype.toString.call([1, 2]);       // "[object Array]"
Object.prototype.toString.call(3);            // "[object Number]"
Object.prototype.toString.call(true);         // "[object Boolean]"
Object.prototype.toString.call(undefined);     // "[object Undefined]"
Object.prototype.toString.call(null);         // "[object Null]"
// ... and more
```

Другие имеют встроенный символ toStringTag:

```
Object.prototype.toString.call(new Map());      // "[object Map]"
Object.prototype.toString.call(function* () {}); // "[object GeneratorFunction]"
Object.prototype.toString.call(Promise.resolve()); // "[object Promise]"
// ... and more
```

Методы символов

Symbol()

Создать локальный символ.

```
let id = Symbol([описание])
```

Symbol.description

Документация [здесь](#).

The read-only `description` property is a string returning the optional description of `Symbol` objects.

Работает с локальными, глобальными и известными символами:

```
const a = Symbol('описание');
console.log( a.description );
// описание

console.log( Symbol.for('foo').description );
// "foo"

console.log(Symbol.iterator.description);
// "Symbol.iterator"
```

`Symbol.for(key)`

Принимает имя и создаёт (или возвращает) глобальный символ.

Для чтения (или, при отсутствии, создания) символа из глобального реестра.

Ищет глобальный символ по заданному ключу (имени, идентификатору, описанию) и возвращает его.

Если символа с таким описанием в реестре нет, то создаст новый символ для данного ключа в глобальном реестре.

Метод сначала проверяет, существует ли символ с заданным идентификатором в реестре — и возвращает его, если тот присутствует. Если символ с заданным ключом не найден, то создаст новый глобальный символ.

```
Symbol.for("foo");    // создаёт новый глобальный символ
Symbol.for("foo");    // возвращает глобальный символ, созданный ранее
```

`Symbol.keyFor(sym)`

Принимает глобальный символ и возвращает его имя.

Возвращает строку с ключом заданного символа, если он есть в глобальном реестре символов, либо `undefined`, если его там нет.

Этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт `undefined`.

```
const globalSym = Symbol.for('foo');
// сначала создаёшь

console.log( Symbol.keyFor(globalSym) );
// получаешь "foo"
```

`Symbol.prototype`

Содержит прототип конструктора `Symbol`.

`Symbol.length`

Содержит длину, всегда равную 0 (нулю).

Для объектов:

`Object.assign`

Это метод клонирования объекта. В отличие от цикла `for..in`, скопирует и строковые, и символьные свойства.

Object.getOwnPropertySymbols()

Возвращает массив всех символьных свойств, найденных непосредственно на переданном объекте.

Поскольку изначально никакой объект не содержит собственных символьных свойств, метод

Object.getOwnPropertySymbols() будет возвращать пустой массив, пока вы не установите символьные свойства на вашем объекте.

Дестракчеринг

Документация [здесь](#).

Деструктуризация (дестракчеринг) — специальный синтаксис, позволяющий извлекать части из составных данных. Можно использовать как синтаксический сахар для объявления переменных, обмена значениями и т.д.

Деструктуризация массива

Общий паттерн такой:

```
let [переменная, переменная] = массив[элемент1, элемент2]
```

```
let [a, b] = [1, 2, 3];
```

ненужные элементы могут быть пропущены

```
let[c, , d] = [1, 2, 3];
```

сразу в свойства объектов

```
[user.name, user.surname] = "Ilya Kantor".split(' ')
```

деструктурировать можно любой перебираемый объект

```
let [one, two, three] = new Set([1, 2, 3]);
```

```
let [a, b, c] = "abc";
```

Значения по умолчанию

```
let [name = "Guest", surname = "Anonymous"] = ["Julius"];
```

Обмен значениями

```
let a = 1;
let b = 2;
[a, b] = [b, a];
```

обойти объект по-взросому

```
for (const [key, value] of Object.entries(obj) ) {
  console.log(key);
  console.log(value);
}
```

Деструктуризация объекта

Ну вот:

```
const/let { <ключ объекта>: <переменная>, в которую надо сохранить значение ключа} = объект
```

ключ объекта

Наименование существующего ключа в объекте, из которого будет скопировано значение
переменная

Создаваемая переменная, в которую будет скопировано значение из объекта

Наименование ключа и переменной совпадают:

```
let { width, height } = options
```

Значения по умолчанию

```
let {width = 100, height = 200} = options
let {incomingProperty: varName = defaultValue} = options
```

...Rest и дестракчеринг

Можно взять из объекта несколько необходимых свойств, а все остальные присвоить куда-нибудь

```
let options = { title: "Menu", height: 200, width: 100 };
let {title, ...rest} = options;
```

Можно присваивать в уже существующие переменные.

```
let title, width, height;
({title, width, height} = {title: "Menu", width: 200, height: 100});
```

Внимание на скобки, без скобок работать не будет.

JS обрабатывает {...} как самостоятельный блок кода, но на самом-то деле у нас деструктуризация. Чтобы показать JavaScript, что это не блок кода, можно заключить выражение в скобки ({...} = {...})

Деструктуризация параметров функции

Когда мы передаём в функцию аргумент при вызове, его значение присваивается формальному параметру функции. Это неявное автоматическое присвоение:

```
const foo = (x) => {
  // let x = arg
};
```

Для аргументов-объектов

Типичная ситуация, когда на вход функции приходит объект с большим количеством свойств. В таких случаях надо забрать только нужные значения:

```
const func = ({ name, surname } = {}) => {
  console.log(name);
  console.log(surname);
};

// внутри происходит это:
// let { name, surname } = { name: 'John', surname: 'Doe' };
```

Для аргументов-массивов

```
const func = ([first, second]) => { ... };
const func = ([first = 1, second = 2]) => { ... };
```

Прочая хуйня по деструктуризации

Дестракчеринг массива можно использовать в циклах:

```
const points = [ [4, 3], [0, -3] ];
for (const [x, y] of points) {
  console.log([x, y]);
}
```

```
// => [ 4, 3 ]
// => [ 0, -3 ]
```

То же самое для структуры map:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

for (let [key, value] of user) {
  console.log(`#${key}: ${value}`); // name: John, затем age: 30
}
```

Деревья и дестракчеринг массива

Если при дестракчеринге указать переменную так, что ей не будет соответствовать ни один элемент массива, то в переменную запишется значение undefined. Это можно использовать при работе с деревьями:

```
const node = ['a', ['b']];

// присваивание в какой-то функции
const [name, children] = node;

// b эту проверку не пройдёт, потому что он лист и у него нет потомков
if (!children) {
  return [name];
}
```

Умные параметры функций

Если функция имеет много параметров, то вот так – плохой способ их объявлять:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) { ... }
```

- невозможно запомнить порядок, в котором эти параметры передаются в функцию;
- чтобы какие-то параметры в середине использовали значение по умолчанию, им надо будет вручную прописывать undefined:

```
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

Эти проблемы решаются деструктуризацией. Нужно сделать так, чтобы функция на вход принимала объект из параметров.

Обратите внимание, что такое деструктуризование подразумевает, что в функцию обязательно будет передан объект. Если нам нужны все значения по умолчанию, следует передать пустой объект, а лучше сделать его передачу по умолчанию. Если весь объект аргументов по умолчанию равен {}, то всегда есть что-то, что можно деструктурировать.

Самый простой вариант реализации:

```
// объект с параметрами
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// функция извлекает свойства в переменные
function showMenu({title = "Untitled", width = 200, height = 100, items = []} = {}) {
  console.log(`#${title} ${width} ${height}`);
  console.log(items);
}
```

```
showMenu(options);
// My Menu 200 100
// Item1, Item2
```

Более сложное деструктурирование со вложенными объектами и двоеточием:

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({
  title = "Untitled",
  width: w = 100,           // width присваиваем в w
  height: h = 200,          // height присваиваем в h
  items: [item1, item2]    // первый элемент items присваивается в item1, второй в item2
} = {}) {

  console.log(` ${title} ${w} ${h}`); // My Menu 100 200
  console.log(item1); // Item1
  console.log(item2); // Item2
}

showMenu(options);
```

Вложенная деструктуризация

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства.

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

let { size: {width, height}, items: [item1, item2], title = "Menu" } = options;

console.log(title); // Menu
console.log(width); // 100
console.log(height); // 200
console.log(item1); // Cake
console.log(item2); // Donut
```

Комбинирование деструктуризации

Деструктуризацию объекта можно комбинировать с деструктуризацией массива.

На практике эта возможность (именно глубокая деструктуризация сложных структур) используется очень редко, потому что такой код достаточно сложен для восприятия.

```
const x = { o: [1, 2, 3] };
const { o: [a, b, c] } = x;

console.log(a); // => 1
console.log(b); // => 2
```

```
console.log(c); // => 3

const y = { o: [[1, 2, 3], { what: 'WHAT' } ] };
const { o: [[one, two, three], { what } ] } = y;

console.log(one); // => 1
console.log(two); // => 2
console.log(three); // => 3
console.log(what); // => 'WHAT'
```

Rest, Spread

Оператор «...» выполняет различные действия в зависимости от того, где применяется.

По сути, spread осуществляет итерацию. Поэтому он работает со всеми с итерируемыми объектами:

```
let str = "Привет";
alert( [...str] ); // П,р,и,в,е,т
```

Rest у функций документация [здесь](#).

Spread документация [здесь](#).

Функции, Rest и Spread

При объявлении функции

...rest упакует лишние аргументы в массив.
Rest может быть только один и обязательно в конце.
`function foo(...rest) {...}`
`const func = (a, b, ...params) => {...}`

При вызове функции

...spread извлекает элементы.
Может быть любое количество spread-операторов, они могут располагаться в любом порядке.
`foo(...arr);`
`Math.max(...arr1, ...arr2);`

arguments

К аргументам функции можно обращаться и по-старому — через псевдомассив arguments.

Отличие ...rest от arguments (дубликат из раздела про функции):

Существует три основных отличия оставшихся параметров от объекта [arguments](#):

1. оставшиеся параметры включают только те, которым не задано отдельное имя, в то время как объект arguments содержит все аргументы, передаваемые в функцию;
2. объект arguments не является массивом, в то время как оставшиеся параметры являются экземпляром [Array](#);
3. объект arguments имеет дополнительную функциональность (например, свойство callee).
4. Стрелочные функции не имеют arguments.

См. также раздел по функциям с arguments.

Массивы, Rest и Spread

Rest

Разложить на первый элемент и все остальные

```
const [first, second, ...rest] = ['apple', 'orange', 'banana', 'pineapple'];
```

```
Если интересует только часть массива, то slice() лучше  
const rest = fruits.slice(1);
```

Spread

```
Слияние нескольких массивов  
let merged = [0, ...arr, 2, ...arr2];
```

```
Конвертация в массив  
[...str]  
[...args]
```

Array.from — более универсальный метод:

- Array.from работает как с псевдомассивами, так и с итерируемыми объектами
- Оператор расширения работает только с итерируемыми объектами

Объекты и Spread

```
Разложить объект на список пар «ключ:значение», можно делать копию объекта  
const a = { key1: 'value1' };  
const b = { ...a };
```

Запись новых и перезапись существующих свойств

```
const a = { key1: 'value1' };  
const b = { ...a, newValue: 'newValue' };
```

Слияние объектов

```
const a = { key1: 'value1' };  
const b = { key2: 'value2' };  
const c = { ...a, ...b }
```

Обновление свойства объекта без мутации

```
const object = { key1: value1, key2: value2 };  
const newObject = { ...object, key2: newValue };
```

</>

ФУНКЦИИ

Функция – это фрагмент кода для выполнения определённых задач, к которому можно обратиться из другого места программы.

Функция – способ группировки команд.

Обычные значения, такие как строки или числа представляют собой *данные*. Функции можно воспринимать как «действия».

См. также:

Деструктуризация параметров функции

Обычные функции

Есть 2 способа объявить обычную функцию: declaration и expression.

```
declaration
всплывает, работает super
function foo(arg) {
    return arg;
}
```

```
expression
const foo = function(arg) {
    return arg;
};
```

named Function Expression

это для того, чтобы FE стал локальным и мог вызывать сам себя

```
let sayHi = function func(who) {
    alert(`Hello, ${who}`);
};
```

Разница между ними:

1. Declaration всплывают
2. Declaration в классах имеет скрытое свойство [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.
3. У Expression ставится точка с запятой на конце, а у Declaration нет.

Пояснения по разнице (можно пропустить)

Когда движок JavaScript готовится выполнять код, он ищет в нём Function Declaration и создаёт их, поэтому они могут быть вызваны раньше своих определений. Expression создаются только в тот момент, когда выполнение доходит до них и только после этого могут использоваться.

Всплытие, поднятие, hoisting – это способность функций, объявленных через function declaration, быть вызываемыми ещё до их объявления.

Expression использует внутри себя инструкции присваивания let sayHi = ...; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой лишь завершает инструкцию.

Есть ещё Named Function Expression

Делается это для того, чтобы FE стал локальным и мог вызывать сам себя. См. примеры ниже в «Объект функции».

Стрелочные функции

Анонимны, если их не присвоить какой-то переменной.

Мануал [здесь](#).

однострочные:

```
() => {}
const foo = (argument) => argument;
```

многострочные

```
const foo = (argument) => {
    return argument;
};
```

Особенности стрелочных функций:

- не имеют super
- не имеют «arguments»
- привязаны к значению this (контекст) того места, где объявлены, а не вызваны.
- нельзя использовать как конструкторы с new (следствие отсутствия this).

Создание методов объектов

Методы объектов создаются такими бесстрелочными функциями:

```
const company = {
  name: 'Hexlet',

  // сокращённая запись метода
  getName() {
    return this.name;
  }

  getName: function getName() {
    return this.name;
  },
};
```

Доступ к переменным

Переменные, объявленные внутри функции, видны только внутри этой функции.

Функция обладает доступом к внешним переменным и может изменять их значение. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

Параметры функции

Мы можем передать внутрь функции любую информацию, используя параметры (аргументы) функции.

Передаваемые значения копируются в параметры функции и становятся локальными переменными. Функция получает именно копию.

```
const foo = (n) => {
  n = 'abc';      // перезаписывает параметр
  return n;
}

let a = 2;
foo(a);
console.log(a); // 2 - значение не изменилось
```

Параметры по умолчанию

Если параметр не указан, то его значением становится `undefined`.

Чтобы задать параметр по умолчанию:

```
const foo = (x = 'default') => x;
```

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы. Например, с помощью оператора «или»:

```
function showMessage(from, text) {
  text = text || 'текст не добавлен';
}
```

Return

Результат функции с пустым `return` или без него – `undefined`

Если функция не возвращает значения, это всё равно, как если бы она возвращала undefined:

```
const foo = () => {
  return
};

console.log(foo());
// undefined
```

return и перевод строки

Интерпретатор подставит точку с запятой после return.

Корректный перевод

```
return (
  some + long + expression
)
```

Не выполнится

```
return;
(some + long + expression)
```

Выбор имени функции

Используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение.

get...	возвращают значение
calc...	вычисляют
create...	создают
show	показывают
check...	проверяют и возвращают логическое значение, и т.д.

Сверхкороткие имена функций

Имена функций, которые используются очень часто, иногда делают сверхкороткими.

Во фреймворке [jQuery](#) есть функция с именем \$. В библиотеке [Lodash](#) основная функция представлена именем _. Это исключения.

Одна функция – одно действие

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием.

Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе (в этом случае мы можем создать третью функцию, которая будет их вызывать).

Функции-коллбеки

Можно передавать функции как параметры. Идея в том, что мы передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо. Например, задаётся вопрос, и в зависимости от варианта ответа вызывается та или иная функция.

Бросать коллбеки в функцию можно прям так, они всё равно будут вызываться:

```
const promise = ...
.catch(console.log);
```

Функции высшего порядка

Map, filter и reduce. Перенёс в методы массивов, потому что неудобно сюда мотать, когда они нужны.

Аргументы

Rest

```
function foo(...rest) { ... }
foo(...spread);
```

См. подробнее в разделе Rest и Spread

Отличие ...rest от arguments

1. оставшиеся параметры включают в себя только те, которым не задано отдельное имя, в то время как объект arguments содержит все аргументы, передаваемые в функцию;
2. объект arguments не является массивом, в то время как оставшиеся параметры являются массивом;
3. объект arguments имеет дополнительную функциональность, специфичную только для него (например, свойство callee).

Arguments

Документация [тут](#).

Объект arguments – это псевдомассив, который содержит аргументы, переданные в **нестрелочную** функцию. Псевдомассив означает, что arguments имеет свойство length, а элементы индексируются начиная с нуля, его можно обойти в цикле.

Лучше использовать rest параметры.

Стрелочные функции не имеют "arguments". Если разместить стрелочную функцию внутри обычной, она будет подтаскивать аргументы обычной функции из-за замыкания.

Преобразование arguments в массив:

```
var args = Array.prototype.slice.call(arguments);
var args = [].slice.call(arguments);

// ES2015
const args = Array.from(arguments);
const args = [...arguments];
```

Пример не самый удачный, но всё же: создание функции, соединяющей несколько строк.

В функцию первым параметром передаётся separator, а потом любое количество строк:

```
function myConcat(separator) {
  var args = Array.prototype.slice.call(arguments, 1);
  return args.join(separator);
}
```

Рекурсия и стек

Рекурсивный способ возведения числа в степень:

```
function pow(x, n) {
```

```

if (n == 1) {
    return x;
} else {
    return x * pow(x, n - 1);
}

alert( pow(2, 3) ); // 8

ещё короче:
function pow(x, n) {
    return (n == 1) ? x : (x * pow(x, n - 1));
}

```

Рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – к ещё более простому и так далее, пока значение не станет очевидно.

Глубина рекурсии – общее количество вложенных вызовов (включая первый)

Максимальная глубина рекурсии ограничена движком JavaScript.

В примере выше она будет равна степени n.

Рекурсивная (рекурсивно определяемая) **структура данных** – это структура, которая повторяет саму себя в своих частях. Речь идёт о дереве. Применительно к вебу, это html. HTML-теги могут содержать фрагменты текста/комментарии или другие теги и т.д. Это снова рекурсивное определение.

Контекст выполнения, стек

Работа рекурсивных вызовов:

Информация о процессе выполнения запущенной функции хранится в её контексте выполнения (execution context).

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this и прочую служебную информацию.

Стек вызовов (call stack) – LIFO-стек, хранящий информацию для возврата управления из функций в программу (или подпрограмму, при вложенных или рекурсивных вызовах). При вызове подпрограммы или возникновении прерывания, в стек заносится адрес возврата — адрес в памяти следующей инструкции приостановленной программы и управление передается подпрограмме или подпрограмме-обработчику.

Когда функция производит вложенный вызов, происходит следующее:

1. Выполнение текущей функции приостанавливается.
2. Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – *стеке контекстов выполнения*.
3. Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
4. После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

Все вызовы складываются в стек. Вызывается одна и та же функция, но контекст у неё всегда разный. Например, в одном вызове x = 2, в другом (рекурсивном) x = 1. Когда выполнение подвызова закончится, можно будет легко вернуться назад, потому что контекст сохраняет как переменные, так и точное место кода, в котором он остановился.

Отличие рекурсии от итеративного процесса в том, что в рекурсии подвызовы складываются в стек, а в итеративный вариант использует один контекст, в котором будут последовательно меняться значения переменных и результата.

Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее, потому что не надо создавать и запоминать множество контекстов.

Рекурсивные обходы

Для дерева тяжело писать императивную функцию и легко рекурсивную.

Эта функция считает зарплату работников из отделов, которые могут быть в другом подотделе.

Принцип: если встречен массив, то это лист и надо подсчитать результат; если объект – использовать рекурсивный вызов:

```
let company = {
  sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 600}],
  development: {
    sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800}],
    internals: [{name: 'Jack', salary: 1300}]
  }
};

// Функция для подсчёта суммы зарплат
function sumSalaries(department) {
  if (Array.isArray(department)) {
    return department.reduce((prev, current) => prev + current.salary, 0); // сумма элементов массива
  } else {
    let sum = 0;
    for (let subdep of Object.values(department)) {
      sum += sumSalaries(subdep); // рекурсивно вызывается для подотделов, суммируя результаты
    }
    return sum;
  }
}

alert(sumSalaries(company)); // 6700
```

Связанный список

Для лучшего понимания мы рассмотрим ещё одну рекурсивную структуру под названием «связанный список», которая в некоторых случаях может использоваться в качестве альтернативы массиву.

Методы массива shift и unshift – дорогостоящие, потому что придётся переиндексировать весь массив. Самые малозатратные операции – с концом массива, это push и pop.

Если нужны быстрые вставка/удаление, мы можем выбрать другую структуру данных, называемую [связанный список](#). Элемент связанного списка – это value (значение) в первой ячейке и next – свойство во второй ячейке, которое содержит указание (ссылается) на следующий элемент связанного списка или null.

В отличие от массивов, нет перенумерации, элементы легко переставляются.

```
let list = {
  value: 1,
  next: {
    value: 2,
    next: {
      value: 3,
      next: {
        value: 4,
        next: null
      }
    }
  }
};

// Альтернативный код для создания:
let list = { value: 1 };
list.next = { value: 2 };
```

```

list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// Список можно легко разделить на несколько частей и впоследствии объединить обратно:
let secondList = list.next.next;
list.next.next = null;

// Для объединения:
list.next.next = secondList;

// можем вставить или удалить элементы из любого места.
// Например, для добавления нового элемента нам нужно обновить первый элемент списка:
let list = { value: 1 };
list.next = { value: 2 };
list.next.next = { value: 3 };
list.next.next.next = { value: 4 };

// добавление нового элемента в список
list = { value: "new item", next: list };

// Чтобы удалить элемент из середины списка, нужно изменить значение next предыдущего элемента:
list.next = list.next.next;

```

Списки могут быть улучшены:

- Можно добавить свойство `prev` в дополнение к `next` для ссылки на предыдущий элемент, чтобы легко двигаться по списку назад.
- Можно также добавить переменную `tail`, которая будет ссылаться на последний элемент списка (и обновлять её при добавлении/удалении элементов с конца).
- ...Возможны другие изменения: главное, чтобы структура данных соответствовала нашим задачам с точки зрения производительности и удобства.

Замыкание

Замыкание – запоминаение функцией части окружения, где она была задана. Функция замыкает в себе идентификаторы (все, что мы определяем) из лексической области видимости.

Замыкание – это функция, которая запоминает свои внешние переменные и может получить к ним доступ.

В JavaScript, все функции изначально являются замыканиями (есть только одно исключение: синтаксис [new Function](#)). Они автоматически запоминают, где были созданы, с помощью скрытого свойства `[[Environment]]` и все они могут получить доступ к внешним переменным.

Когда на собеседовании фронтенд-разработчик получает вопрос: «что такое замыкание?», – правильным ответом будет определение замыкания и объяснения того факта, что все функции в JavaScript являются замыканиями, и, может быть, несколько слов о технических деталях: свойстве `[[Environment]]` и о том, как работает лексическое окружение.

Лексическое Окружение

Понятие «лексическое окружение» или «статическое окружение» в JavaScript относится к возможности доступа к переменным, функциям и объектам на основе их расположения в исходном коде.

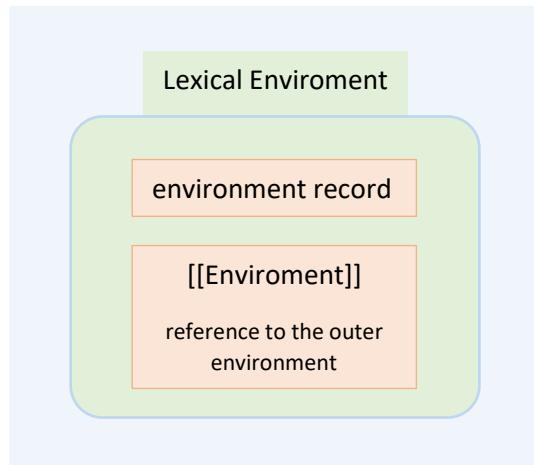
Лексическое окружение, `LexicalEnvironment` – это специальный скрытый объект, который хранит в себе сведения о идентификаторах и переменных. Такой объект есть у каждой выполняемой функции, блока кода и скрипта.

Здесь «идентификатор» – это имя переменной или функции, а «переменная» – это ссылка на объект (сюда входят и функции) или значение примитивного типа.

Объект лексического окружения состоит из двух частей:

1. Запись окружения (`environment record`) – объект, в котором хранятся все локальные переменные (а также другая информация, типа значение `this`).

2. Ссылка на внешнее окружение (reference to the outer environment) — ссылка, позволяющая обращаться к внешнему (родительскому) лексическому окружению. То есть к тому, что соответствует коду снаружи (снаружи от текущих фигурных скобок).



Получается, что скрытый объект `LexicalEnvironment` состоит из двух частей-окружений:

Первое – локальное в **Environment Record**.

Второе – глобальное в **[["Enviroment"]]**, ссылается на глобальный объект.

Мы не можем получить его в нашем коде и изменять напрямую. Сам движок JavaScript может оптимизировать его, уничтожать неиспользуемые переменные для освобождения памяти и выполнять другие внутренние уловки.

«Переменная» – это просто свойство специального внутреннего объекта: `Environment Record`.

Работа с переменными – это на самом деле работа со свойствами этого объекта. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».

Концептуально лексическое окружение выглядит так:

```
lexicalEnvironment = {  
  environmentRecord: {  
    <identifier> : <value>,  
    <identifier> : <value>  
  }  
  outer: <Reference to the parent lexical environment>  
}
```

Функции, в отличие от переменных, полностью инициализируются когда создаётся лексическое окружение, а не тогда, когда интерпретатор доходит до них. Вот почему мы можем вызвать функцию, объявленную через `Function Declaration`, до того, как она определена.

Функции и лексическое окружение

При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова. Внутри функции – своё лексическое окружение.

Функции имеют доступ ко всем переменным во внешнем (глобальном) окружении.

Если внутри функции создаётся переменная с таким же именем, какое у переменной в глобальном окружении, переменная внутри функции перекрывает внешнюю.

Когда код хочет получить доступ к переменной, он сначала ищёт её во внутреннем лексическом окружении, затем во внешнем, затем в следующем внешнем и так далее, до глобального.

Каждый новый вызов функции создаёт одно новое лексическое окружение

Новое лексическое окружение функции создаётся каждый раз, когда функция выполняется.

Для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

Вложенные функции

Функция называется «вложенной», когда она создаётся внутри другой функции.

Вложенная функция может быть возвращена и использована, она всё так же **будет иметь доступ к тем же переменным, которые были для неё внешними в момент её создания.**

Дочерняя функция всегда имеет доступ к внешним переменным того места, где она была создана.

```
function makeCounter() {  
  let count = 0;  
  
  return function() {  
    return count += 1; // есть доступ к внешней переменной "count"  
  };  
}  
  
let counter = makeCounter();  
  
alert( counter() ); // 0  
alert( counter() ); // 1  
alert( counter() ); // 2
```

Сбросить счётчик `count` из кода, который не принадлежит `makeCounter`, нельзя: `count` – локальная переменная родительской функции, мы не можем получить к ней доступ извне.

Новый вызов функции – новое лексическое окружение. Для каждого вызова `makeCounter()` создаётся новое лексическое окружение функции, со своим собственным `count`. Так что если вернуть и сохранить в переменные несколько дочерних функций, то функции будут независимы друг от друга и их счётчики в родительской функции – тоже независимы:

```
function makeCounter() {  
  let count = 0;  
  return function() {  
    return count++;  
  };  
}  
  
let counter1 = makeCounter();  
let counter2 = makeCounter();  
  
alert( counter1() ); // 0  
alert( counter1() ); // 1  
  
alert( counter2() ); // 0 (независимо)
```

Блоки кода {}

Лексическое окружение существует для любых блоков кода в фигурных скобках: {...}

Для цикла у каждой итерации своё отдельное лексическое окружение. Если переменная объявлена в `for(let ...)`, то она также в отдельном лексическом окружении.

Можно самостоятельно использовать блоки кода {...}, чтобы изолировать переменные в «локальной области видимости».

Иключение – `var`. Для `var` не существует блочной области видимости, она всегда в глобальной.

Например, в браузере все скрипты (кроме `type="module"`) разделяют одну общую глобальную область. Так что, если мы создадим глобальную переменную в одном скрипте, она станет доступна и в других. Но это становится источником конфликтов, если два скрипта используют одно и то же имя переменной и перезаписывают друг друга. Это может произойти, если название переменной – широко распространённое слово.

Если мы хотим этого избежать, мы можем использовать блок кода для изоляции всего скрипта или какой-то его части. Из-за того, что у блока есть собственное лексическое окружение, код снаружи него (или в другом скрипте) не видит переменные этого блока:

```
{  
  let message = "Hello";  
  alert(message); // Hello  
}  
  
alert(message); // Ошибка: переменная message не определена
```

IIFE

IIFE (Immediately Invoked Function Expression) это JavaScript [функция](#), которая выполняется сразу же после того, как она была определена. Функция становится мгновенно выполняющимся функциональным выражением. Переменная, которой присвоено IIFE, хранит в себе результат выполнения функции, но не саму функцию.

[MDN](#)

```
(function () {  
  statements  
})();
```

Это тип выражений, также известный как [Self-Executing Anonymous Function](#), который состоит из двух основных частей.

- Первая - это сама анонимная функция с лексическим скопом, заключеннымнутри [Оператора группировки](#) (). Благодаря этому переменные IIFE замыкаются в его пределах, у него есть свои локальные переменные и глобальная область видимости ими не засоряется.
- Вторая часть – вызов функции () .

В прошлом в JavaScript не было такого лексического окружения на уровне блоков кода.

Так что программистам пришлось что-то придумать.

Пример:

```
(function() {  
  let message = "Hello";  
  alert(message); // Hello  
})();
```

Со скобками получается так: (создать функцию)(вызывать).

Это трюк, который позволяет показать JavaScript, что функция была создана в контексте другого выражения.

Кроме скобок, существуют и другие пути. В настоящий момент нет необходимости писать подобный код (а как тогда? мб анонимные функции?)

```
// Пути создания IIFE  
  
(function() {  
  alert("Скобки вокруг функции");  
})();  
  
(function() {  
  alert("Скобки вокруг всего");  
})();  
  
!function() {  
  alert("Выражение начинается с логического оператора NOT");  
}();  
  
+function() {
```

```
    alert("Выражение начинается с унарного плюса");
}();
```

Сборка мусора (функции)

Лексическое окружение очищается и удаляется после того, как функция выполнилась.

Если есть вложенная функция, которая всё ещё доступна после выполнения `f`, то у неё есть свойство `[[Environment]]`, которое ссылается на внешнее лексическое окружение, тем самым оставляя его «живым».

```
function f() {
  let value = 123;
  function g() { alert(value); }
  return g;
}

let g = f();
// g доступно и продолжает держать внешнее лексическое окружение в памяти
```

если `f()` вызывается несколько раз и возвращаемые функции где-то сохраняются, тогда все соответствующие каждой сохранённой функции объекты лексического окружения продолжат держаться в памяти. Вот три такие функции в коде ниже:

```
function f() {
  let value = Math.random();
  return function() { alert(value); };
}

// три функции в массиве, каждая из них ссылается на лексическое окружение
// из своего вызова f()
let arr = [f(), f(), f()];
```

Объект лексического окружения существует только до того момента, пока есть хотя бы одна вложенная функция, которая ссылается на него.

Оптимизация на практике

В теории, пока функция жива, все внешние переменные тоже сохраняются.

Но на практике движки JavaScript пытаются это оптимизировать. Они анализируют использование переменных и, если легко по коду понять, что внешняя переменная не используется – она удаляется.

Одним из важных побочных эффектов в V8 (Chrome, Opera) является то, что такая переменная становится недоступной при отладке.

Попробуйте запустить следующий пример в Chrome с открытой Developer Tools.

Когда код будет поставлен на паузу, напишите в консоли `alert(value)`.

Такой переменной не существует! В теории, она должна быть доступна, но попала под оптимизацию движка.

```
function f() {
  let value = Math.random();

  function g() {
    debugger; // в консоли: напишите alert(value); Такой переменной нет!
  }
  return g;
}

let g = f();
g();
```

Глобальный объект

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.

Глобальный объект имеет универсальное стандартное имя для всех сред исполнения – `globalThis`.

В браузере глобальный объект – это `window`, в Node.js – `global`.

Ко всем свойствам глобального объекта можно обращаться напрямую, без указания `window`.

```
alert("Привет");
window.alert("Привет");
```

Есть ещё термин **глобальные объекты**, или стандартные встроенные объекты.

Это объекты в глобальном пространстве имён (но только, если не используется строгий режим ECMAScript 5! В противном случае они возвращают `undefined`).

Их не следует путать с самим глобальным объектом. Сам же глобальный объект в глобальном пространстве имён доступен через оператор `this`.

Запись свойств в глобальный объект

В глобальный объект можно дозаписывать свойства, но делать это не рекомендуется.

Во-первых, `var` становятся свойствами глобального объекта, если только не используются [модули](#).

Во-вторых, можно что-то дозаписать в ГО вручную, оно станет доступным отовсюду:

```
window.currentUser = {
  name: "John"
};

// где угодно в коде
alert(currentUser.name);
alert(window.currentUser.name);
```

Глобальный объект можно использовать для записи полифилов: проверить наличие встроенного объекта `Promise` и если его нет, то его можно дозаписать:

```
if (!window.Promise) { alert("Ваш браузер очень старый!"); }

if (!window.Promise) { window.Promise = ... // реализация }
```

Объект функции, NFE

в JavaScript функция – это значение. Каждое значение в JavaScript имеет свой тип. Тип функции – это объект.

Функции можно не только вызывать, но и использовать как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д. Объект функции содержит несколько полезных свойств.

Функции могут содержать дополнительные свойства. Многие известные JavaScript-библиотеки используют эту возможность.

Они создают «основную» функцию и добавляют множество «вспомогательных» функций внутрь первой.

Например, библиотека [lodash](#) создаёт функцию `_`, а потом добавляет в неё `_.clone`, `_.keyBy` и другие свойства.

Они делают это, чтобы уменьшить засорение глобального пространства имён посредством того, что одна библиотека предоставляет только одну глобальную переменную, уменьшая вероятность конфликта имён.

Таким образом, функция может не только делать что-то сама по себе, но также и предоставлять полезную функциональность через свои свойства.

.name

Имя функции доступно как свойство «.name».

Если функция не имеет .name, то JavaScript пытается определить его из контекста. Если корректное имя определить невозможно (анонимная функция в массиве), то свойство name имеет пустое значение.

```
function sayHi() {
  alert("Hi");
}
alert(sayHi.name); // sayHi

let sayHi = function() {
  alert("Hi");
};
alert(sayHi.name); // sayHi
```

.length

Свойство «length» содержит количество параметров функции в её объявлении. Остаточные параметры не считаются.

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```

Свойство length иногда используется для [интроспекций](#) в функциях, которые работают с другими функциями.

Интроспекция и её пример.

Интроспекция (англ. type introspection) в программировании — возможность запросить тип и структуру объекта во время выполнения программы. Интроспекция может использоваться для реализации ad-hoc-полиморфизма. Например, функция ask в примере ниже принимает следующие параметры: вопрос и какие-то функции-обработчики. Когда пользователь отвечает на вопрос, функция вызывает обработчики. Мы можем передать два типа обработчиков:

- Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
- Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

```
function ask(question, ...handlers) {
  let isYes = confirm(question);

  for(let handler of handlers) {
    if (handler.length == 0) {
      if (isYes) handler();
    } else {
      handler(isYes);
    }
  }

  // для положительных ответов вызываются оба типа обработчиков
  // для отрицательных - только второго типа
  ask("Вопрос?", () => alert('Вы ответили да'), result => alert(result));
```

Это частный случай так называемого [Ad-hoc-полиморфизма](#) – обработка аргументов в зависимости от их типа или, как в нашем случае – от значения `length`.

!!!! Дозапись свойств в функции

В функции, как в любые объекты, можно дозаписывать свойства.

Рассматривать это предлагается на примере изготовления свойства-функции-счётчика, который считает количество своих запусков.

Давайте добавим свойство `counter` для отслеживания общего количества вызовов:

```
function sayHi() {
  console.log("Hi");
  sayHi.counter += 1;
}

// начальное значение установлено вне функции
// если записать в функцию, то значение всегда будет 1
sayHi.counter = 0;

sayHi(); // Hi
sayHi(); // Hi
console.log(`Вызвана ${sayHi.counter} раза`); // Вызвана 2 раза
```

Свойство функции, назначенное как `sayHi.counter = 0`, не объявляет локальную переменную `counter` внутри неё. Пример выше работает, потому что свойство функции находится вне этой функции. Когда функция отрабатывает, её удаляет сборщик мусора, а внешнее значение остаётся в глобальной области видимости (хз, почему так).

Если поместить свойство `counter` внутри функции, то ничего работать не будет. Функция вызывается, увеличивает счётчик на 1, на этом заканчивается и сборщик мусора удалит её с внутренними свойствами. Повторные вызовы начнут всё сначала:

```
function sayHi() {
  console.log("Hi");
  sayHi.counter = 0;
  sayHi.counter += 1;
}

sayHi();
sayHi();
console.log(sayHi.counter); // 1
```

Иногда свойства функции могут использоваться вместо замыканий. Ещё один способ сохранить переменную, в которой хранится количество вызовов – сделать функцию внутри другой функции и присвоить все это какой-то внешней переменной. Например, можно переписать функцию-счётчик из главы [Замыкание](#), используя её свойство:

```
function makeCounter() {

  function foo() {
    return foo.count += 1;
  }

  // вместо let count = 0
  foo.count = 0;
  return foo;
}
```

```
let a = makeCounter();
console.log( a() ); // 1
console.log( a() ); // 2

// если сделать так, то ничего не получится
console.log( makeCounter()() ) // 1
console.log( makeCounter()() ) // 1
```

Свойство count хранится в лексическом окружении вышестоящей функции. Но при этом надо, чтобы сама функция не удалялась, т.е. была присвоена внешней переменной.

Это хуже или лучше, чем использовать замыкание?

Основное отличие в том, что если значение count живёт во внешней переменной, то оно не доступно для внешнего кода. Изменить его могут только вложенные функции. А если оно присвоено как свойство функции, то мы можем его получить:

```
function makeCounter() {

    function counter() {
        return counter.count+=1;
    };

    counter.count = 0;
    return counter;
}

let a = makeCounter();
// обращение напрямую
console.log( a.count ); // 0
// возможность изменения свойства
a.count = 9;

console.log( a() );      // 10
```

Вот ещё пример работы:

```
Объявление свойства снаружи работает
function f() {
}

f.key = 123;
console.log(f);    // [Function: f] { key: 123 }

объявление внутри работает только после вызова...
function f() {
    f.key = 'value';
}

f();
console.log(f);  // [Function: f] { key: 'value' }

... и не работает до вызова
function f() {
    f.key = 'value';
}

console.log(f);  // [Function: f]
```

```
не работает
function f() {
  key: 123;
  return;
}
console.log(f.key);
// undefined
```

Named Function Expression

Named Function Expression или NFE – это термин для Function Expression, у которого есть имя:

```
let sayHi = function func(who) {
  alert(`Hello, ${who}`);
};
```

Делается это для того, чтобы FE стал локальным и мог вызывать сам себя.
функция sayHi вызывает себя с "Guest", если не передан параметр who:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // использует func, чтобы снова вызвать себя же
  }
};

sayHi(); // Hello, Guest

// А вот так - не сработает:
func(); // Ошибка, func не определена (недоступна вне функции)
```

Если так не сделать, то код будет ломаться в случае изменения переменной, в которую записана функция:

```
let sayHi = function(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    sayHi("Guest"); // Ошибка: sayHi не является функцией
  }
};

let welcome = sayHi;
sayHi = null;

welcome(); // Ошибка, вложенный вызов sayHi больше не работает!
```

Так происходит, потому что функция берёт sayHi из внешнего лексического окружения. Так как локальная переменная sayHi отсутствует, используется внешняя. И на момент вызова эта внешняя sayHi равна null. Необходимо имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы.

Починим:

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
```

```
    func("Guest"); // Теперь всё в порядке
}
};

let welcome = sayHi;
sayHi = null;

welcome(); // Hello, Guest (вложенный вызов работает)
```

Теперь всё работает, потому что имя "func" локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию. Внешний код все ещё содержит переменные sayHi и welcome, но теперь func – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Это не работает с Function Declaration. Для Function Declaration синтаксис не предусматривает возможность объявить дополнительное «внутреннее» имя. Зачастую, когда нам нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

Создание функций через new Function

Конструктор функции на [MDN](#).

Главная фишка – создать функцию из строки.

new Function позволяет превратить любую строку в функцию. Например, получить функцию с сервера и затем выполнить её. Такой метод создает функцию динамически, но страдает от проблем безопасности и производительности. Такие функции выполняются только в глобальной области.

Функция создаётся с заданными аргументами arg1...argN и телом functionBody (это определение, которое обычно пишется в фигурных скобках).

```
let func = new Function(arg1, ...argN, functionBody);
```

Аргументы могут быть объявлены несколькими способами. Эти 3 объявления ниже эквивалентны:

```
new Function('a', 'b', 'return a + b'); // стандартный синтаксис
new Function('a,b', 'return a + b'); // через запятую в одной строке
new Function('a , b', 'return a + b'); // через запятую с пробелами в одной строке
```

Функция создаётся полностью «на лету» из строки, переданной во время выполнения.

```
// с аргументами
let sum = new Function('a', 'b', 'return a + b');
console.log( sum(1, 2) ); // 3

// функция без аргументов, достаточно указать только тело
let sayHi = new Function('console.log("Hello")');
sayHi(); // Hello
```

Что с замыканием?

Обычные функции запоминают лексическое окружение, где они объявлены.

Когда функция создаётся через new Function, в её [[Environment]] записывается ссылка не на внешнее лексическое окружение, в котором она была создана, а на глобальное. Поэтому такая функция имеет **доступ только к глобальным переменным**.

В примере функция создаётся внутри другой и не видит её лексическое окружение:

```
function getFunc() {  
    let value = "test";  
  
    let func = new Function('console.log(value)');  
  
    return func;  
}  
  
getFunc();  
// ReferenceError: value is not defined
```

Поэтому лучшее архитектурное решение – это передавать в такие функции параметры явно. Кроме того, это не вызывает проблем у минификаторов. Если бы new Function имела доступ к внешним переменным, при этом были бы проблемы с минификаторами (о них написано ниже).

Про минификаторы:

перед отправкой JavaScript-кода на реальные работающие проекты код сжимается с помощью *минификатора* – специальной программы, которая уменьшает размер кода, удаляя комментарии, лишние пробелы, и, что самое главное, локальным переменным даются укороченные имена.

Например, если в функции объявляется переменная let userName, то минификатор изменяет её на let a (или другую букву, если она не занята) и изменяет её везде. Обычно так делать безопасно, потому что переменная является локальной, и никто снаружи не имеет к ней доступ. И внутри функции минификатор заменяет каждое её упоминание. Минификаторы достаточно умные. Они не просто осуществляют «тупой» поиск-замену, они анализируют структуру кода, и поэтому ничего не ломается.

Так что если бы даже new Function и имела доступ к внешним переменным, она не смогла бы найти переименованную userName.

Таймеры setTimeout и setInterval

Для планирования вызова существуют два метода:

setTimeout	вызвать функцию один раз через определённый интервал времени.
clearTimeout	отменить вызов

setInterval	вызывать функцию регулярно , повторяя вызов через определённый интервал времени.
clearInterval	отменить вызовы

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам.

Функция, переданная в таймер, выполняется не в текущем стеке вызовов, поэтому к таймерам применимы все особенности АФ. try/catch не ловит ошибки. Ловить ошибки надо с помощью колбеков.

При этом таймер не делает операцию, которая в него передана, асинхронной. Таймер только откладывает время её выполнения. Если сама операция синхронная, то после запуска она заблокирует основной поток выполнения программы, и все остальные будут ждать её завершения.

Таймеры срабатывают либо по заданному времени, либо с задержкой. Рантайм проверяет таймеры, когда в текущем стеке вызовов всё выполнено. Если происходит тяжёлое вычисление, то колбеки и таймеры будут ждать, пока вычисление закончится. Время срабатывания сдвигается в большую сторону, поэтому справедливо говорить, что в таймерах задается минимальное время, после которого будет запущен код.

setTimeout в браузере устанавливает this=window для вызова функции. Для методов в отрыве от объекта он пытается получить window.метод, которого не существует.

В Node.js this становится объектом таймера.

setTimeout()

Вызов функции или выполнение фрагмента один раз через определённый интервал времени.

[MDN](#)

```
let timerId = setTimeout(func|code, [delay], [arg1], [argN])
```

func | code

Функция или строка кода для выполнения.

delay

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

arg1, argN...

Аргументы, передаваемые в функцию (не поддерживается в IE9-)

Частая ошибка – передать в таймер не саму функцию, а её вызов.

Это не работает, потому что setTimeout ожидает ссылку на функцию.

Передать данные внутрь функции можно тремя способами:

```
const f = (a, b) => console.log(a + b);
```

1. дописать параметры в setTimeout

```
setTimeout(f, 1000, 5, 8);
```

2. функция-обёртка

```
setTimeout(() => foo(5, 8), 1000);
```

3. bind, частичное применение, первый параметр null потому что контекст не меняется

```
setTimeout(f.bind(null, 5, 8), 1000);
```

clearTimeout (id)

Вызов **setTimeout** возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения. В браузере идентификатор таймера это числовое значение, в node.js это объект. Синтаксис для отмены таймера:

```
let timerId = setTimeout(f, 1000); // таймер запущен
clearTimeout(timerId);           // отменён
```

setInterval()

setInterval автоматически запускает функцию через указанный промежуток времени до тех пор, пока её явно не остановят через **clearInterval**. Время между запусками равно переданному второму параметру.

Имеет такую же сигнатуру, как и setTimeout:

```
let timerId = setInterval(func|code, [delay], [arg1], [argN])
```

clearInterval (id)

Остановить setInterval можно через **clearInterval**, передав ему ID таймера.

Во время показа alert (в смысле, во время всплытия окошка) время тоже идёт и не ставится на паузу.

В большинстве браузеров, включая Chrome и Firefox, внутренний счётчик продолжает тикать во время показа alert/confirm/prompt.

Если подождать с закрытием alert несколько секунд, то следующий alert будет показан сразу, как только вы закроете предыдущий.

Использование двух таймеров вместе:

- setInterval каждые 5 секунд выполняет печать
- setTimeout через 16 секунд выключает интервальный таймер через clearInterval

```
const id = setInterval(() => console.log(new Date()), 1000);
setTimeout(() => clearInterval(id), 4000); // автоматом отключить через 5 сек

// 2020-05-27T15:01:46.418Z
// 2020-05-27T15:01:47.424Z
// 2020-05-27T15:01:48.424Z
```

Таймер можно остановить изнутри, передав в колбек его *id*.

```
let counter = 0;

const foo = () => {
  counter += 1;

  if (counter === 4) {
    clearInterval(id);
    return;
  }

  console.log(new Date());
};

const id = setInterval(foo, 2000);
```

Рекурсивный setTimeout

Запускать что-то регулярно можно через интервальный setInterval или рекурсивный setTimeout.

Рекурсия проявляется не через return, а в том, что таймер через какое-то время вызывает такой же таймер.

```
// вместо setInterval

let timerId = setTimeout(function tick() {
  alert('tick');
  timerId = setTimeout(tick, 2000);
}, 2000);

// То же самое, только функция объявлена снаружи

function foo() {
  console.log('tick');
  id = setTimeout(foo, 1000);
}

let id = setTimeout(foo, 1000);
```

Рекурсивный setTimeout – более гибкий метод, чем setInterval. С его помощью последующий вызов может быть задан по-разному в зависимости от результатов предыдущего.

Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд... Вот псевдокод:

```
let delay = 5000;
```

```
let timerId = setTimeout(function request() {
  // ...отправить запрос...

  if (ошибка запроса из-за перегрузки сервера) {
    // увеличить интервал для следующего запроса
    delay *= 2;
  }

  timerId = setTimeout(request, delay);
}, delay);
```

Можно запускать функции через 2 секунды строго после их выполнения. SetInterval запустит функцию и сразу же начнёт отсчёт для запуска новой, не дожидаясь окончания старой. SetTimeout можно установить так, что отсчитывать время он будет только после окончания операции:

```
let i = 1;

setTimeout(function run() {
  func(i);
  setTimeout(run, 100);
}, 100);
```

Рекурсивный setTimeout гарантирует фиксированную задержку не между запусками, а между исполнением и новым запуском функции (здесь 100 мс). Это потому, что новый вызов планируется в конце предыдущего.

```
const foo = () => {
  let i = 0;
  while (i < 10000000000) {
    i += 1;
  }
  console.log('!')
  let id = setTimeout(foo, 100);
  return
};

foo()
```

setTimeout с нулевой задержкой

Особый вариант использования: setTimeout(func, 0) или просто setTimeout(func).

Это планирует вызов func настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода (стека).

```
setTimeout(() => console.log('привет'))

const foo = () => {
  let i = 0;
  while (i < 10000000000) {
    i += 1;
  }
  console.log('!')
  return
};

foo()

// !
// привет
```

Есть и более продвинутые случаи использования нулевой задержки в браузерах, которые мы рассмотрим в главе [Событийный цикл: микрозадачи и макрозадачи](#).

Минимальная задержка вложенных таймеров в браузере

В браузере есть ограничение на то, как часто внутренние счётчики могут выполняться. В [стандарте HTML5](#) говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд».

Проверяем. Код ниже моментально запускает одну функцию за другой, замеряя задержку между запусками. Разница между запусками (между первым стартом и последующими) сохраняется в массив:

```
let start = Date.now();
let times = [];

setTimeout(function run() {
  times.push(Date.now() - start); // запоминаем задержку от предыдущего вызова

  // измеряем задержку на протяжении 100 мс
  if (start + 100 < Date.now()) {
    console.log(times);
    return
  }
  return setTimeout(run); // если нужно ещё запланировать
});

// 1,1,1,1,9,15,20,24,30,35,40,45,50,55,59,64,70,75,80,85,90,95,100
```

то же самое, только функция объявлена снаружи

```
let start = Date.now();
let times = [];

function run() {
  times.push(Date.now() - start);

  if (Date.now() > start + 100) {
    console.log(times)
    return;
  }

  return setTimeout(run);
}

setTimeout(run);
```

Первый таймер запускается сразу, а затем задержка вступает в игру, и мы видим 9, 15, 20, 24...

Аналогичное происходит при использовании `setInterval`

Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач.

Например, [`setImmediate`](#) для Node.js. Так что это ограничение относится только к браузерам.

</>

Контекст, `call`, `apply`, `bind`

Это общая тема для разделов из учебника «переадресация вызова» и «привязка контекста».

При передаче методов объекта в качестве колбэков, например для `setTimeout`, возникает проблема – потеря `this`. Это может быть в следующих случаях:

1. передача метода отдельно от объекта;

2. использование таймера. Таймер в браузере устанавливает this=window для вызова функции. В Node.js this становится объектом таймера. setTimeout получает функцию отдельно от объекта и теряет контекст.

Примеры таких потерь:

```
let user = {
  firstName: "Вася",
  sayHi() {
    console.log(`Привет, ${this.firstName}!`);
  }
};

const a = user.sayHi;
a(); // Привет, undefined!

setTimeout(user.sayHi, 100); // Привет, undefined!
```

ФУНКЦИИ ДЛЯ ПЕРЕАДРЕСАЦИИ ВЫЗОВА

Передать контекст явно и переадресовать вызов могут методы .call, .apply и .bind.

```
const res = foo.call(this, x, y);
аргументы через запятую и сразу же выполняется

const res = foo.apply(this, [x, y]);
аргументы в массиве и сразу же выполняется

const res = (foo.bind(this, x, y))();
аргументы через запятую, но не выполняется
```

Пример:

```
let myObject = {
  name: 'myObjectect',
  method() {
    console.log(this.name);
  }
};

const a = myObject.method;
a(); // undefined
a.call(myObject); // correct
```

Стрелочная функция-обёртка

Стрелочная функция запоминает окружение, в котором была создана. Таким образом создаётся замыкание:

```
const obj = {
  foo() {
    console.log(this);
  }
}

setTimeout( () => obj.foo(), 1 ); // правильный объект
setTimeout( obj.foo, 1 ); // Timeout { ... }
```

В данном стрелочная функция замыкается на глобальном окружении и объект «obj» достаётся из него. В коде появилась уязвимость: до момента срабатывания setTimeout в переменную может быть записано другое значение. .bind гарантирует, что такого не случится.

.bind

Метод создаёт новую функцию и устанавливает ей контекст выполнения `this`. Кроме того, к функции можно применить все или часть аргументов (частичное применение).

В отличие от `call` и `apply`, `bind` не вызывает функцию сразу же, а возвращает "обёртку", которую можно вызвать позже.

Документация [здесь](#).

Привязка выполняется только один раз и затем её нельзя отменить. Это связано с тем, что `.bind` возвращает экзотический объект [bound function](#), возвращаемый при первом вызове. Следующий вызов `bind` будет устанавливать контекст уже для этого объекта. Это ни на что не повлияет.

```
let boundFoo = foo.bind(this);
let boundFoo = foo.bind(this, arg1, arg2...);
```

`this`

Значение, передаваемое в качестве `this` в целевую функцию при вызове привязанной функции

`arg1, arg2...`

Аргументы целевой функции («предустановленные»), передаваемые перед аргументами привязанной функции

Методы объектов фиксируются так:

```
const obj = {
  name: 'correct',
  method() {
    console.log(this.name)
  }
};

const bindMethod = obj.method.bind(obj)
bindMethod(); // correct
```

Массовая привязка всех методов к объекту

Если у объекта много методов и их надо активно передавать куда-то ещё, то можно привязать их контекст в цикле. И они всё ещё остаются методами исходного объекта и исправно работают с ним.

```
for (let key in user) {
  if (typeof user[key] == 'function') {
    user[key] = user[key].bind(user);
  }
}
```

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например [_.bindAll\(obj\)](#) в lodash.

Частичное применение

Когда мы привязываем аргументы, такая функция называется «частично применённой» или «частичной». Частичное применение удобно, когда мы не хотим повторять один и тот же аргумент много раз.

Контекст – это обязательный параметр для `bind`, так что нужно передать туда что-нибудь вроде `null`, это такая функция. В примере ниже, функции `multi` не нужен контекст `this`, она работает сама по себе, поэтому нет смысла что-то указывать первым аргументом в качестве контекста.

```
const multi = (a, b) => a * b;
const double = multi.bind(null, 2);
```

```
console.log( double(3) ) // 6
```

Частичное применение для методов объекта

Иногда надо зафиксировать только аргументы без контекста. Null в данном случае не подойдёт, потому что методы могут использовать this. Можно создать такую вспомогательную функцию partial, которая привязывает только аргументы:

```
function partial(foo, ...argsBound) {
  return function(...args) {
    return foo.call(this, ...argsBound, ...args);
  }
}

let object = {
  name: 'Ivan',
  say(time, phrase) {
    console.log(`[${time}] ${this.name}: ${phrase}!`);
  }
};

const date = new Date();
// добавить новый метод в объект
// часы и минуты из даты - это 1-й аргумент, который $time
object.sayNow = partial(object.say, date.getHours() + ':' + date.getMinutes());

// 2-й аргумент - это phrase. Его надо добавить
object.sayNow('Hello')
// [15:24] name: Hello!
```

В этом примере используется call, а не bind, потому что call вызывается сразу же.

Также есть готовый вариант [_.partial](#) из библиотеки lodash.

Декораторы и кеш

Рассмотрим, как перенаправлять вызовы между функциями и как их декорировать.

Декораторы — это, по сути, «обёртки», которые дают нам возможность изменить поведение функции, не изменяя её код. Это приём программирования, который позволяет взять существующую функцию и изменить/расширить её поведение.

Заимствование метода — использование метода одной структуры данных в контексте другой структуры.

Допустим, есть детерминированная функция «slow(x)», выполняющая ресурсоёмкие вычисления, которая для одного и того же «x» всегда возвращает один и тот же результат. Имеет смысл кэшировать (запоминать) возвращаемые ею результаты, чтобы повторно их не вычислять. Вместо того, чтобы усложнять slow(x) дополнительной функциональностью, мы заключим её в функцию-обёртку — «wrapper», которая добавит кэширование.

Логика кэширования является отдельной, она не увеличивает сложность оригинальной функции.

В качестве хранилища будет использоваться Map, потому что в Map в качестве ключей можно использовать любые типы данных.

Кеш: с функцией

Самый простой вариант — с обычной внешней функцией:

```
function slow(x) {
  console.log(`Вычисляется ${x}`);
  // ресурсоёмкие вычисления
  return x;
}
```

```

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) { // если кеш содержит такой аргумент...
      return cache.get(x); // читаем результат из кеша
    }

    let result = func(x); // иначе, вызываем функцию
    cache.set(x, result); // и запоминаем результат
    return result;
  };
}

// запись декоратора в переменную
slow = cachingDecorator(slow);

console.log( slow(1) ); // первый запуск вычисляется
console.log( slow(1) ); // повторный читается из кеша

```

Кеш: с методами объектов

Вариант выше не работает с методами объектов. Проблема в том, что при передаче в декоратор метода произойдёт потеря this и это будет вызывать ошибку: не удаётся прочитать свойство «someMethod» из «undefined». Такую неправильную реализацию я закину в конец, чтобы глаза не мозолила.

Правильная реализация декоратора, чтобы он мог работать с методами объектов, предполагает использование .call, .bind или .apply. В одной из строчек можно использовать любой из 3-х методов:

```

const res = foo.call(this, x);
const res = foo.apply(this, [x]);
const res = (foo.bind(this, x))();

```

```

let myObject = {
  name: 'myObjectect',
  method(arg) {
    console.log(this.name, arg);
  }
};

function decorator(foo) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {
      console.log('cache');
      return cache.get(x);
    }

    const res = foo.call(this, x);
    // const res = foo.apply(this, [x]);
    // const res = (foo.bind(this, x))();
    cache.set(x, res);
    return res;
  };
}

myObject.method = decorator(myObject.method);
myObject.method(123);

```

```
// myObjectect 123
```

Кеш: несколько аргументов

У функций выше есть проблема: они не могут кешировать больше одного аргумента. Это можно исправить несколькими способами:

1. Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна, чем встроенный Map, и поддерживает множественные ключи.
2. Использовать вложенные коллекции: cache.set(min) будет Map, которая хранит пару (max, result). Тогда получить result мы сможем, вызвав cache.get(min).get(max).
3. Соединить два значения в одно: использовать строку из аргументов как ключ к Map. Можно передать **хеширующую** функцию в декоратор, которая перегоняет аргументы в строку.

Реализуем третий вариант: в обёртку должна быть встроена ещё одна функция – **hash()**, чтобы принимать аргументы и склеивать их между собой в строку. Ещё один технический вопрос связан с Arguments – это псевдомассив, поэтому к нему нельзя применить метод массивов .join() напрямую. Чтобы перегнать Arguments в массив, можно:

- использовать распаковку [...arguments];
- снова использовать заимствование метода, на этот раз у new Array.

```
function hash(args) {  
  return [].join.call(args);  
  return [...args].join;  
}
```

Внутри обёртки-декоратора

```
let key = hash(arguments);  
if (cache.has(key)) {  
  return cache.get(key);  
}
```

Этот трюк называется **заимствование метода**.

Метод join заимствуется из обычного массива: [].join.call в контексте arguments. Он намеренно написан так, что допускает любой псевдомассив this (не случайно, многие методы следуют этой практике). Вот почему он также работает с this=arguments.

Реализация:

```
let myObject = {  
  name: 'correct',  
  method(a, b) {  
    return [this.name, a, b];  
  }  
};
```

```
function hash(args) {  
  return [].join.call(args);  
}
```

```
function decorator(foo, hash) {  
  let cache = new Map();  
  
  return function() {  
    let key = hash(arguments);  
    if (cache.has(key)) {  
      return cache.get(key);  
    }  
  };  
}
```

```

    }
    let result = foo.apply(this, arguments);
    cache.set(key, result);
    return result;
};

}

myObject.method = decorator(myObject.method, hash);

console.log( myObject.method(3, 5) ); // первый вызов вычисляется
console.log( myObject.method(3, 5) ); // второй берёт значение из кеша
// [ 'correct', 3, 5 ]

```

#? Не понимаю, почему хеш в этом примере возвращает не строку, а [object Arguments]?

Для использования метода в отрыве от объекта (хотя я так и не понял, почему в отрыве, обёртка же в объект записывается), чтобы привязать this, можно использовать .call (с распаковкой аргументов) или .apply (умеет работать с псевдомассивами).

Кеш: неправильная реализация

Методы объектов требуют привязки к нему. В случае, если в первый вариант реализации кеша обернуть метод объекта, то объект отвяжется и в итоге получится ошибка: не удается прочитать свойство «someMethod» из «undefined». Декоратор передаёт вызов оригинальному методу, но без контекста.

Это похоже на ситуацию, когда метод объекта записывается в переменную:

```
let func = worker.slow;
func(2); // undefined
```

Неправильная реализация (из учебника, там немного другой пример):

```

let worker = {
  someMethod() {
    return 1;
  },
  slow(x) {
    console.log("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // здесь будет ошибка
    cache.set(x, result);
    return result;
  };
}

worker.slow = cachingDecorator(worker.slow);
console.log( worker.slow(2) );

// TypeError: this.someMethod is not a function

```

Итого

Декоратор – это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией.

Обычно безопасно заменить функцию или метод декорированным, за исключением одной мелочи. Если исходная функция предоставляет свойства, такие как `func.calledCount` или типа того (???), то декорированная функция их не предоставит. Потому что это обёртка. Так что нужно быть осторожным в их использовании. Некоторые декораторы предоставляют свои собственные свойства.

Декораторы можно рассматривать как «дополнительные возможности» или «аспекты», которые можно добавить в функцию. Мы можем добавить один или несколько декораторов. И всё это без изменения кода оригинальной функции!

Мы также рассмотрели пример **заимствования метода**, когда мы вызываем метод у объекта в контексте другого объекта. Весьма распространено заимствовать методы массива и применять их к `arguments`. В качестве альтернативы можно использовать объект с остаточными параметрами `...args`, который является реальным массивом.

Теория по функциям

Определение функции

Это её код, в котором она создаётся. Функция создаётся как константа. Тело функции – это всё, что записано в определении.

Встречая `return`, интерпретатор останавливает дальнейшее выполнение функции и возвращает указанное значение в то место, где была вызвана функция. Если в функции нет инструкции `return`, то интерпретатор автоматически возвращает `undefined`

Передача аргументов

Аргументы передаются в функцию в круглых скобках. Если их несколько, то они указываются через запятую. Аргументы можно не передавать и их значением станет `undefined`. В большинстве языков так делать нельзя. Имена аргументов имеют смысл исключительно в теле функции (см. область определения). Аргумент должен быть переменной, иначе он не сможет быть аргументом.

Параметры функции

Если передать в функцию больше аргументов, чем она рассчитывает получить, то лишние параметры будут просто проигнорированы.

О параметрах (аргументах) по умолчанию смотри ниже.

Формальные параметры – имена переменных в определении функции. Например в функции `const foo = (a, b) => a - b;` формальные параметры – это `a` и `b`. С ними пишутся формулы и т.д.

Фактические параметры – это то, что по факту было передано в функцию в момент вызова. Например, `foo(5, 8)` фактическими параметрами являются `5` и `8`.

Параметры по умолчанию

Мануал [тут](#).

Если не передать в функцию какой-то параметр, то его значение будет `undefined`. Параметры, которым «не досталось» аргумента, автоматически инициализируются значением `undefined`.

```
const greeting = name => console.log(`Hi, ${name}!`);
```

Параметры функции по умолчанию задаются через знак равенства. Это делает аргумент необязательным для передачи и указания. Все необязательные аргументы должны располагаться только в конце списка аргументов. Можно сделать так, чтобы при вызове функции без параметров она присваивала им какое-то значение, заданное по умолчанию:

```
const greeting = (name = 'anonymous') => console.log(`Hi, ${name}!`);
```

Интересная штука. Если какой-то язык не поддерживает значения по умолчанию, то можно просто добавить условную конструкцию:

```
const greeting = name => console.log(`Hi, ${name ? name : 'anonymous'}!`);
```

Детерминированность

Функция называется детерминированной, когда для одних и тех же входных аргументов она возвращает один и тот же результат. Например, функция, переворачивающая строку, детерминированная. Функция, возвращающая случайное число, не является детерминированной.

Предикат

Предикат отвечает на утвердительный вопрос «да» или «нет», возвращая только значение типа `bool`.

Поэтому предикаты именуются особым образом: с префикса «`is`» или «`has`».

Функция-предикат, проверяющая на совершеннолетие:

```
const isAdult = (age) => age >= 18;
```

Цикломатическая сложность

Цикломатическая сложность - это структурная или топологическая мера сложности компьютерной программы.

Это количество линейно независимых маршрутов через программный код. Если исходный код не содержит никаких точек ветвления или циклов, то сложность равна единице, поскольку есть только один маршрут через код.

Если код имеет единственный оператор `if`, содержащий простое условие, то существует два пути через код: один, если условие оператора `if` имеет значение `true`, и один — если `false`.

У функции `sum` цикломатическая сложность равна единице, а у функции `abs` — двойке, так как она содержит ветвление, а значит два независимых пути выполнения.

Линтеры многих языков измеряют показатель сложности и сигнализируют, если он больше 5 для одной функции.

guard expression

Это некое условие или выражение, которое уменьшает сложность функции. Её идея в том, что в начале функции проверяются самые простые условия, для которых нужно минимум вычислений. В зависимости от того, `true` оно или `false`, решается, будет ли дальше выполняться кусок кода или нет.

Позволяет сделать код более читаемым и понятным, избежать большого количества вложений и лишних проверок, если эти проверки нецелесообразны уже только потому, что условие `guard expression = false`.

Иными словами, это такое условие, которое определяет, будет ли целесообразным дальнейшее выполнение связанного участка кода.

Замыкание

Замыкание – это когда функция запоминает часть окружения, где она была задана. Функция замыкает в себе идентификаторы (все, что мы определяем) из лексической области видимости.

```
const savePassword = password => passwordForCheck => password === passwordForCheck;
```

Каррирование — это процесс превращения функции от n аргументов в цепочку вложенных n -функций от одного аргумента. Соответственно, каррированная функция — это множество функций от одного аргумента.

Предположим, что у нас есть функция `const sum = (a, b, c) => a + b + c`, которая складывает три числа. Тогда ее каррированная версия будет выглядеть так: `const sum2 = a => b => c => a + b + c`, а использование таким:

```
const sum2 = (a) => {
  return (b) => {
    return (c) => {
      return a + b + c;
    };
  };
};
```

Объекты первого класса

Объекты первого класса (также «first-class citizen») – это элементы, которые могут быть переданы как параметр в функцию, возвращены из функции или присвоены переменной. Это всё, что может быть данными: числа, строки и остальные типы данных, в т.ч. функции.

Функции обладают всеми перечисленными свойствами:

могут быть переданы в другую функцию как аргумент (см. функции высшего порядка);

```
arr.sort([compareFunction])  
  
say(() => 'hi!');  
// => hi!
```

могут быть возвращены из другой функции как результат;

```
const foo = () => ((a, b) => a + b);  
foo()(2, 3);  
// вызов
```

могут быть сохранены в переменной или структурах данных;

```
const x = () => console.log('hey')
```

Из вики:

Объект внутренне самоопознаем (независим от именования) (см. анонимные функции).

```
(v => v)('run');  
// run
```

Объект может быть создан во время выполнения программы;

Анонимные (лямбда) функции

Лямбда-функции не присвоены константе, у них нет имени (поэтому и анонимные).

Названием обязаны лямбда-исчислению в математике.

Анонимную функцию можно использовать напрямую, без сохранения в константе. Определил и сразу вызвал с помощью оператора вызова функции (`()`). При этом определение функции следует обернуть в круглые скобки (не путать с оператором вызова функции!), чтобы обозначить границы определения для интерпретатора.

```
((() => console.log('hey'))());  
// => hey
```

```
(v => v)('run');  
// run  
  
((a, b) => a + b)(3, 2);  
// 5  
// const sum = (a, b) => a + b;  
// sum(3, 2);
```

Минимальное определение функции, которое только возможно, выглядит так:

```
() => {}
```

Нужно понимать, что определение функции и её присваивание константе — две разных операции. Чистое определение, без присваивания переменной, выглядит так:

```
() => console.log('hey');
```

А здесь происходят следующие две операции:

- создание константы со значением в виде функции: `const x =`
- создание функции: `() => console.log('hey')`

```
const x = () => console.log('hey')
```

Справочно:

Функции НЕ являются примитивными значениями, они имеют сложную структуру и не могут быть сохранены в переменной как, к примеру, числа. Поэтому они хранятся по ссылке, так же как и [массивы](#). Поэтому, условимся, что, говоря "сохранили функцию в переменной" подразумеваем "сохранили ссылку на функцию в переменной". Это никак не меняет сути излагаемого, но знать об этом стоит.

Функции высшего порядка

Функции высшего порядка – это функции, которые принимают на вход и/или возвращают другие функции. Они реализуют обобщенный алгоритм задачи, делегируя обработку вызывающему коду через callback-функцию, что позволяет не реализовывать алгоритм каждый раз. Главный плюс от применения таких функций — серьёзное повышение коэффициента повторного использования кода.

Слово '**callback**' означает, что наша задача - передать функцию (но не вызывать!), а вызывать её будет основная функция.

Filter, map и reduce – три главные функции высшего порядка.

Передачу функции в качестве параметра в другую функцию хорошо демонстрирует метод [sort](#) для массивов. В качестве необязательного параметра в метод можно передать функцию для сортировки:

```
arr.sort([compareFunction])
```

Не обязательно создавать переменную для callback-функции. Можно использовать анонимную функцию:

```
users.sort((a, b) => {
  if (a.age === b.age) {
    return 0;
  }
  return a.age > b.age ? 1 : -1;
});
```

Различие между функциями высшего порядка и просто константами:

```
const sum = (a, b, c) => a + b + c;
const sumResult = sum(1, 2, 3);
```

`sumResult` – это не функция, а константа хранящая в себе вызов функции `sum`. Она не принимает функции в качестве аргумента и не возвращает их.

Чистые функции

Чистая функция – это:

1. детерминированная функция
2. без побочных эффектов.

Детерминированность

Детерминированная функция – функция, которая для одних и тех же входных данных всегда возвращает один и тот же результат.

Такими являются функции в математике. Для одного и того же x результат работы функции $y = f(x)$ будет один и тот же.

Функция `console.log` – детерминированная, она всегда возвращает одно и тоже значение для любых входных данных – `undefined`, а не то, что печатается на экран. Печать на экран – побочный эффект

Недетерминированная функция – функция, которая для одних и тех же входных аргументов (в т.ч. при отсутствии аргументов) может возвращать разные значения.

Например, генератор случайных чисел [`Math.random\(\)`](#) или возвращающая системное время [`Date.now\(\)`](#).

Функция становится недетерминированной и в том случае, если она обращается не только к своим аргументам, но и некоторым внешним данным, например глобальным переменным, переменным окружения и так далее. Внешние данные могут изменяться, и функция начнёт выдавать другой результат, даже если в ней передаются одни и те же аргументы.

Например, функция `getCurrentShell` обращается к переменной окружения `SHELL`. Но в разные моменты времени и в разных окружениях значение этой переменной может быть различным.

Побочные эффекты (side effects)

Побочный эффект – любые действия, изменяющие среду выполнения.

Это любые файловые операции, такие как запись в файл, отправка или приём данных по сети, даже вывод в консоль или чтение файла. Кроме того, побочными эффектами считаются обращения к глобальным переменным (как на чтение, так и запись) и изменение входных аргументов в случае, когда они передаются по ссылке.

Наличие побочных эффектов – вторая ключевая характеристика функций.

Вызов функции с побочными эффектами также считается побочным эффектом.

```
const sayHiTo = (name) => {
  const greeting = `Hi, ${name}!`;
  console.log(greeting);
};
```

Любые вычислительные операции не являются побочными эффектами. Например, функция, суммирующая два переданных аргументами числа.

```
const sum = (num1, num2) => num1 + num2;
```

Инкремент и декремент – единственные базовые арифметические операции в JS, которые обладают побочными эффектами (изменяют само значение в переменной). Именно поэтому с ними сложно работать в составных выражениях. Они могут приводить к таким сложноотлавливаемым ошибкам, что во многих языках вообще отказались от их введения (в Ruby и Python их нет). В JS [стандарты кодирования](#) предписывают их не использовать.

Чистые функции

Чистая функция – это детерминированная функция, которая не производит побочных эффектов. Такая функция зависит только от своих входных аргументов и всегда ведёт себя предсказуемо. Такие функции на 100% соответствуют своим математическим аналогам и могут рассматриваться как математические функции.

Чистые функции обладают рядом ключевых достоинств:

1. Их просто тестировать. Достаточно передать на вход функции нужные параметры и посмотреть ожидаемый выход.
2. Их безопасно запускать повторно, что особенно актуально в асинхронном коде или в случае многопоточного кода.
3. Их легко комбинировать, получая новое поведение без необходимости переписывать программу (подробнее далее по курсу).

[Дополнительные материалы](#)

[Побочные эффекты](#)

[Детерминированная функция](#)

Замыкание и самовызывающиеся функции

Паттерн DRY – don't repeat yourself, не нужно писать один и тот же код в разных местах.

Замыкание – это функция вместе со всеми внешними переменными, которые ей доступны.

В JS для новых переменных выделяется память. Переменные, на которые больше нет ссылок, удаляются из памяти. Область видимости создаётся функциями (кроме глобальной). Если одна функция выполнена и вызывает за собой другую, то вторая функция может ссылаться на переменные из первой, не создавая свои новые. Вторая функция ищет переменные у себя, не находит и идёт в родительскую область видимости.

Обратный эффект называется затенение. Это когда когда внутри дочерней функции объявляется переменная с таким же именем, как и переменная в материнской.

Для того, чтобы защитить функции от их переназначения (когда одна функция создана, а потом кто-то создаёт другую функцию с таким же именем и всё ломается), можно сделать из них модуль.

IIFE, immediately-invoked function expression – принцип, по которому функция создаётся и сразу же возвращает другую функцию (самовызывающаяся функция).

Самовызывающиеся функции пишутся через function expression.

Функции anon можно не подписывать, а оставить анонимными:

```
var getDateString = (function anon1() {
    // анонимная функция создаёт format
    function format(date) {
        return date.toGMTString()
    }

    // потом вызывает новую функцию, куда ты будешь скидывать дату,
    // которая потом использует format
    return function anon2(date) {
        date = date || new Date();
        return format(date);
    }
}()); // последний вызов - это вызов первой анонимной функции, которая вернёт
      // вторую анонимную функцию, в которую надо скинуть дату.

console.log(getDateString())
```

Чтобы функция получилась самовызывающаяся, надо поставить скобки вызова после неё одним из следующих способов:

```
(function () {
}());

(function () {
})();
```

Особенности работы this со стрелочными функциями

Главное отличие стрелочных функций от обычных – как они работают с контекстом.

Обычная ищет контекст в том месте, где она вызвана. Стрелочная – где она определена.

Если создать и обычную, и стрелочную функции, которые печатают this, то контекстом их вызова будет сам модуль, а в es6 this у модулей не определен. Поэтому вести они себя будут одинаково:

```
// стрелочная
const arrowFoo = () => {
  console.log(this);
};

// обычная
function commonFoo() {
  console.log(this);
}

arrowFoo(); // undefined
commonFoo(); // undefined
```

Если дозаписать эти функции в объект, то обычная связывается с контекстом объекта, а стрелочная – нет. Стрелочная не имеет своего контекста и связывается с лексическим окружением, то есть функцией, внутри которой она определена. Это нельзя изменить с помощью call или bind.

```
const obj = { arrowFoo, commonFoo };

obj.arrowFoo(); // undefined
arrowFoo.call({ name: 'this is name' }); // undefined
arrowFoo.bind({ name: 'this is name' })(); // undefined

obj.commonFoo(); // { arrowFoo: [Function: arrowFoo], commonFoo: [Function: commonFoo] }
```

Если сразу же определить эти функции в объекте, то результат будет таким же. Стрелочная функция описывается внутри объекта и вызывается из этого же объекта, но контекст все равно связан с местом определения функции (лексическим окружением) – а это сам модуль.

```
const obj = {
  arrowFoo: () => {
    console.log(this);
  },
  commonFoo() {
    console.log(this);
  },
};

obj.arrowFoo(); // undefined
obj.commonFoo(); // { arrowFoo: [Function: arrowFoo], commonFoo: [Function: commonFoo] }
```

Можно определить стрелочную функцию внутри обычной функции. Тогда стрелочная зацепится за контекст обычной. У стрелочной функции НЕ появился this, это не её контекст. Контекст заимствован у внешней функции commonFoo.

Стрелочная вызывается напрямую, а не из объекта. Обычные функции в такой ситуации теряют контекст, а стрелочная сохранила, потому что это контекст места её определения:

```
const obj = {
  name: 'this is name',
  commonFoo() {
    const arrow = () => console.log(this);
    arrow();
  },
};

obj.commonFoo(); // { name: 'this is name', commonFoo: [Function: commonFoo] }
```

Точно такой же код с обычной функцией не заработает, потому, что вызывается как обычная функция, а не метод. В таком случае ее контекст ничему не равен:

```
const obj = {
  name: 'this is name',
  commonFoo() {
    const f = function() {console.log(this.name)};
    f();
  },
};

obj.commonFoo(); // undefined
```

Отсутствие своего контекста делает невозможным использование оператора new вместе со стрелочными функциями:

```
const f = () => {};
const obj = new f();
// TypeError: function is not a constructor
```

</>

JS - Объекты

Документация – [здесь](#) (объект как конструктор).

Статья в руководстве о работе с объектами [здесь](#).

Объект (словарь, ассоциативный массив) – коллекция пар «ключ-значение», где каждый ключ уникален. Это набор свойств, и каждое свойство состоит из имени и ассоциированного с этим именем значения. Свойство объекта можно понимать как переменную, закрепленную за объектом.

Если значением свойства является функция, то её называют методом объекта. Различие свойства/метода это не более чем условность.

В качестве ключей – только строки и символы.

Другие типы данных будут автоматически преобразованы к строке. Зарезервированные слова разрешено использовать как имена свойств, такого ограничения нет. Но есть специальное свойство `__proto__`, которое по историческим причинам имеет особое поведение.

В качестве значений – любые типы данных.

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке». Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. См. раздел «Копирование объекта».

В этой связи, объект, объявленный через `const`, может быть изменён.

Объявление `const` защищает от изменений только само значение переменной. В случае с объектом, значение константы – это ссылка на объект, и это значение мы не меняем. Изменяя объект, мы действуем внутри объекта, а не переназначаем переменную.

Где использовать?

Объект подходит для хранения и обработки разнотипных данных, которые, как правило, описывают какую-то сущность. (Массив предназначен для хранения и обработки коллекций однотипных элементов.) Кроме того, объекты используются как хранилища для конфигурационных параметров или как способ передать в функцию множество разнородных данных в виде одного параметра.

Синтаксис

Создание объекта

Документация [здесь](#).

Объекты могут быть созданы с помощью:

```
{ key: value }    литеральной (инициирующей) нотации  
  
new Object()  
new Object({ key: value })  
  
Object.create( Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj) );  
  
Object.assign(target, source)  
Object.fromEntries([ key, value ])
```

Литеральная:

Внутри скобок через запятую перечисляются пары «ключ-значение» в формате key: value.

Последнее свойство объекта может заканчиваться запятой. Это называется «висячая запятая». Но в json придётся запятую убрать, он с ней не работает.

Создание объекта через конструктор new Object()

```
let myCar = new Object();  
let myCar = new Object({make: Ford, year: 1998,});
```

Создание объекта с помощью свойства Object.create()

Создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.

```
Object.create(proto[, propertiesObject])  
  
proto  
Объект, который станет прототипом вновь созданного объекта  
propertiesObject  
Необязательный. дескрипторы свойств.
```

Создание методов

Функцию, которая является свойством объекта, называют методом этого объекта.

Эти две записи не эквивалентны (см. функции):

```
const company = {  
  name: 'Hexlet',  
  
  // сокращённая запись метода  
  getName() {  
    return this.name;  
  }  
  
  getName: function getName() {  
    return this.name;
```

```
 },  
};
```

Сравнение объектов

Два объекта равны только в том случае, если это один и тот же объект.

Операторы равенства == и строгого равенства === для объектов работают одинаково.

```
let a = {};  
let b = a; // копирование по ссылке  
  
alert( a == b ); // true  
alert( a === b ); // true
```

Два независимых объекта с одинаковым содержимым не равны:

```
let a = {};  
let b = {};  
  
alert( a == b ); // false
```

Object.is()

определяет, являются ли два значения одинаковыми значениями. Но объект всё равно должен быть одним и тем же по ссылке.

```
var isSame = Object.is(value1, value2);
```

Копирование объектов

Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению». В результате мы имеем две независимые переменные.

Переменные с объектами хранят не сам объект, а его «адрес в памяти», ссылку.

Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется. В результате две переменные ссылаются на один и тот же объект.

Копировать объект можно следующими способами:

```
{ ...spread }  
Object.assign(target, source)  
Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));  
Object.fromEntries([ key, value ])
```

...spread

```
let copy = { ...original};
```

Object.assign

Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.

```
Object.assign(target, ...sources)  
  
var object = { key1: 'value1', key2: 'value2' };
```

```
var copy = Object.assign({}, object);

let user = { name: "Иван" };
let permissions1 = { canView: true };
let permissions2 = { canEdit: true };

Object.assign(user, permissions1, permissions2);
копираем все свойства в user
{ name: "Иван", canView: true, canEdit: true }
```

Object.create

Вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством [[Prototype]].

Тут комбинация сразу трёх методов: Object.create, Object.getPrototypeOf, Object.getOwnPropertyDescriptors

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Собрать новый объект из ключей и значений:

```
function getObjCopy(originalObject) {

  let copy = {};
  for (let key of Object.keys(originalObject)) {
    copy[key] = originalObject[key];
  }

  return copy;
};
```

Через метод `_.cloneDeep` из библиотеки lodash:

```
// import _ from 'lodash';
import { cloneDeep } from 'lodash';

let original = { key1: 'value1', key2: 'value2' };
let copy = cloneDeep(original);
```

Если объекты содержат в себе другие объекты, то простого копирования с ключами и свойствами не достаточно для создания независимой копии. Для этого надо использовать `_.cloneDeep` из библиотеки lodash или самому написать рекурсивную функцию. Такая функция должна проверять, является ли значение по ключу объектом, и если да – запускаться для него рекурсивно. Это называется «глубокое копирование».

Обращение к свойствам

Существует 2 способа доступа к элементам объекта:

точечная нотация
myObject.key1

скобочная нотация (см. [property accessors](#))
myObject['key1']

Обращение к несуществующему возвращает undefined.

Через точку можно обращаться только к таким свойствам, которые начинаются с буквы, \$, _.

Документация [здесь](#) и [здесь](#).

Квадратные скобки дают больше возможностей, чем точечная нотация.

Если свойство состоит из нескольких слов через пробел:

```
let user = { "likes birds": true };
user['likes birds']
```

Если свойство было записано как число, к нему можно обратиться только так:

```
const myObject = {1: 'value1' };
myObject['1']
```

В квадратных скобках можно указать переменную, в которой хранится имя свойства для обращения:

```
const a = 'key1';
const object = { key1: 'value 1' };

object.a    ничего не произойдёт
object[a]   value 1
```

Изменение свойств

Чтобы изменить существующее свойство, нужно обратиться по ключу и изменить значение.

Удаление свойств

Удалить элемент из объекта можно с помощью оператора delete.

```
delete myObject['key2'];
```

Добавление свойств

через нотации:

```
myObject['newKey'] = 'new value';
myObject.newKey = 'new value';
```

Для создания пары свойство-значение, достаточно просто передать в объект переменную:

```
const x = 10;
const y = 20;

const obj = {x, y}
obj = { x: 10, y: 20 }
```

Хороший пример использования этой фишки:

```
function makeUser(name, age) {
  return {
    name, // name: name
    age, // age: age
  };
}
```

При создании объекта, можно указать наименование ключа из переменной:

```
const n = 'name';
const m = 'married';
const a = 'age';

const user = { [n]: 'Vasya', [m]: true, [a]: 25 };
// {
  name: 'Vasya',
  married: true,
  age: 25
}
```

Можно делать префиксы:

```
const prefix = '_pre_';

const obj = {
  [prefix + 'first']: 'Glad',
  [prefix + 'second']: 'Valakas'
};

{
  _pre_first: 'Glad',
  _pre_second: 'Valakas'
}
```

Проверить наличие свойства

Оператор `in`

Оператор `in` возвращает `true`, если свойство содержится в указанном объекте или в его цепочке прототипов.

- Если свойству присвоено значение `undefined`, то для этого свойства `in` вернет значение `true`.
- Для свойств, которые унаследованы по цепочке прототипов, тоже возвращается `true`. Если вы хотите проверить только не наследованные свойства, используйте `Object.hasOwnProperty()`.
- Ищет свойство в прототипе.
- Если нет кавычек, то значит указана переменная, в которой находится имя свойства.

```
"key" in object

let user = { name: "John", age: 30 };

console.log( "age" in user );    // true
console.log( "blabla" in user ); // false
```

`Object.hasOwnProperty()`

Возвращает `true` или `false`, указывающее, содержит ли объект указанное свойство.

Не проверяет существование свойств в цепочке прототипов объекта.

```
object.hasOwnProperty('propertyName')

const obj = { a: 1, b: 2, c: 3 };
console.log(obj.hasOwnProperty('a'))
// true
```

`_has` из Lodash

Альтернатива встроенному методу `Object.hasOwnProperty`.

```
_has(object, path)
```

Опциональная цепочка ?.

Новая фича в JS (октябрь 2020) – это **опциональная цепочка «?»**

Опциональная цепочка – это безопасный способ доступа к свойствам вложенных объектов.

Если обратиться к несуществующему свойству объекта, ошибки не будет и вернётся `undefined`.

Но если обратиться к несуществующему свойству несуществующего свойства, то скрипт упадёт.

Такая же ошибка будет, если нужно получить данные об HTML-элементе, который отсутствует на странице.

[Optional chaining](#) в MDN.

В случае использования опциональной цепочки, просто вернётся `undefined`.

Синтаксис имеет три формы:

```
obj?.prop  
obj?.[prop]  
obj?.method?().()
```

также работает с функциями и квадратными скобками
`arr?.[index]`
`func?(args)`

Вот безопасный способ обратиться к свойству `user.address.street`:

```
let user = {};  
// пользователь без адреса  
  
alert( user?.address?.street );  
// undefined (без ошибки)
```

Чтение адреса с помощью конструкции `«user?.address»` выполняется без ошибок, даже если объекта `user` не существует:

```
let user = null;  
  
alert( user?.address );  
// undefined  
alert( user?.address.street );  
// undefined
```

Трансформации объекта

У объектов нет множества методов, которые есть в массивах, например `map`, `filter` и других.

Можно использовать `Object.entries` с последующим вызовом `Object.fromEntries`:

```
let prices = {  
    banana: 1,  
    orange: 2,  
    meat: 4,  
};  
  
let doublePrices = Object.fromEntries(  
    Object.entries(prices)  
        .map( ([key, value]) => [key, value * 2])  
);  
  
alert(doublePrices.meat); // 8
```

Обход объекта

в цикле for...of

Обход реализуется опосредованно:

1. получить массив ключей или массив пар ключ-значение (Object.keys или Object.entries);
2. по-взросленому обойти с дестракчерингом.

```
const myObject = { key1: 'value1', key2: 'value2' };

for (const [key, value] of Object.entries(myObject)) {
  console.log(key);
  console.log(value);
}
```

в цикле for...in

Это цикл по ключам объекта и ключам прототипа, если они перечислимые.

Числовые ключи сортируются, остальные - соответствует порядку объявления (+49 или 1.2)

Если записать так, то при каждой итерации i будет принимать значение ключа в объекте:

```
const object = {
  key1: 'value1',
  key2: 'value2',
  key3: {
    subKey1: 'subValue1',
    subKey2: 'subValue2',
  }
}

for (let i in object) {
  console.log(i);
}

key1 key2 key3
```

This

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, методу user.sayHi() может понадобиться имя пользователя, которое хранится в объекте.

This – ключевое слово для доступа к информации внутри объекта. Значение this – это объект «перед точкой», который использовался для вызова метода.

```
let user = {
  name: "Джон",
  age: 30,

  sayHi() {
    // this - это "текущий объект"
    alert(this.name);
  }
};

user.sayHi(); // Джон
```

Также возможно получить доступ к объекту без ключевого слова `this`, просто прописав наименование объекта: `user.name`.

Но такой код будет ненадёжным. Если мы решим скопировать ссылку на объект `user` в другую переменную, например, `admin = user`, и перезапишем переменную `user` чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из `admin`.

В JavaScript `this` является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»). Функция может быть повторно использована в качестве метода у различных объектов.

Заемствование метода – это использование метода одного объекта в рамках другого объекта.

this не является фиксированным

Значение `this` вычисляется во время выполнения кода и зависит от контекста.

Оно может использоваться в любой функции. В этом коде нет синтаксической ошибки:

```
function sayHi() {  
    alert( this.name );  
}
```

- При объявлении любой функции в ней можно использовать `this`, но этот `this` не имеет значения до тех пор, пока функция не будет вызвана.
- Эта функция может быть скопирована между объектами (из одного объекта в другой).
- Когда функция вызывается синтаксисом «метода» – `object.method()`, значением `this` во время вызова является объект перед точкой.

Пример контекста: здесь одна и та же функция назначена двум разным объектам и имеет различное значение «`this`» при вызовах:

```
let user = { name: "Джон" };  
let admin = { name: "Админ" };  
  
function sayHi() {  
    alert( this.name );  
}  
  
// используем одну и ту же функцию в двух объектах  
user.f = sayHi;  
admin.f = sayHi;  
  
// вызовы функции, приведённые ниже, имеют разное значение this  
user.f(); // Джон  
admin.f(); // Админ  
  
admin['f'](); // Админ
```

Потеря `this (undefined)`

Мы можем вызвать функцию без использования объекта:

```
function sayHi() {  
    alert(this);  
}  
  
sayHi(); // undefined
```

В строгом режиме ("use strict") в таком коде значением `this` будет являться `undefined`.

В нестрогом режиме значением this в таком случае будет **глобальный объект** (window для браузера). Это – исторически сложившееся поведение this, которое исправляется использованием строгого режима ("use strict"). Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

Если создать метод с this в объекте и обращаться к нему через объект (this.name), то всё будет работать хорошо. Но если этот метод записать во внешнюю переменную, то метод будет работать, но потеряет связь с объектом и будет вызываться сам по себе. Его значение будет undefined:

```
const obj = {
  name: 'Valera',
  hi() {
    console.log(`Hi, ${this.name}`);
  }
};

obj.hi()
// Hi, Valera

const a = obj.hi;
a();
// Hi, undefined
```

Внутренняя реализация: ссылочный тип

Некоторые способы вызова метода приводят к потере значения this.

Например:

```
let user = {
  name: "Valera",
  hi() { console.log(this.name); },
  bye() { console.log("Пока"); }
};

user.hi();
// Valera

(user.name == "Valera" ? user.hi : user.bye)(); // undefined!
```

В случае вызова через объект: user.hi(), всё работает хорошо.

Когда используется условный оператор «?», который определяет, какой будет вызван метод в зависимости от выполнения условия, this теряется, хотя внутри оператора прописан объект.

Так происходит из-за «ссылочного типа», называемого [Reference Type](#). Это чисто технический тип, к которому нельзя получить доступ напрямую, и про него нет страницы в руководстве MDN.

В выражении obj.method() происходят две операции:

1. Сначала оператор точка «.» возвращает свойство объекта – его метод (obj.method);
2. Затем скобки () вызывают этот метод (исполняется код метода).

И вот здесь главный смысл: точка «.» возвращает не саму функцию, а специальное значение «ссылочного типа», Reference Type. Это же справедливо для обращения через скобки user['hi']().

Reference Type – это комбинация из трёх значений: base, name, strict, где:

base – это объект;

name – это имя свойства объекта;

strict – это режим исполнения: true, если действует строгий режим (use strict).

Для user.hi из примера выше в строгом режиме Reference Type будет таким: (user, "hi", true).

Таким образом, в выражении «obj.method()» информация о this передаётся из первой части во вторую.

Когда скобки вызова «()» применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный this (=user в данном случае, по base).

Ссылочный тип – исключительно внутренний, промежуточный тип, используемый для передачи информации от точки . до вызывающих скобок (). Это же справедливо для обращения через квадратные скобки [].

При любой другой операции (например, присваивании hi = user.hi), ссылочный тип заменяется на собственно значение user.hi (функцию), и дальше работа уже идёт только с функцией, а не ссылочным типом. Поэтому дальнейший вызов происходит уже без this. Тернарный оператор – это именно про присваивание, а не про потоки выполнения кода.

С if всё работает нормально, потому что if – это про разделение потоков.

Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']().

За исключением вызова метода, любая другая операция (подобно операции присваивания = или сравнения через логические операторы, например ||) превращает это значение в обычное, которое не несёт информации, позволяющей установить this.

Именно поэтому коды типа этих возвращают undefined:

```
f = obj.go; // вычисляется выражение (переменная f ссылается на код функции)
f();          // undefined

(obj.go || obj.stop)();
// undefined
```

Есть несколько вариантов решения проблемы потери значения this. Например, такие как [func.bind\(\)](#).

Решение проблемы потери this

Это тема относится к функциям и здесь есть термины функций типа «замыкания», но я дублирую всё здесь, чтобы не искать в другом документе.

Функция обёртка

Самый простой вариант решения – это обернуть вызов в стрелочную анонимную функцию, создав замыкание:

```
const obj = {
  foo() {
    console.log(this);
  }
}

obj.foo()
// { foo: [Function: foo] }

setTimeout( () => obj.foo(), 1 );
// { foo: [Function: foo] }

setTimeout( obj.foo, 1 );
// Timeout { ... }
```

Код работает корректно, потому что объект «obj» достаётся из замыкания, а затем вызывается его метод «foo». У стрелочных функций нет своего this. Стрелочные функции подтягивают ближайшее внешнее значение this.

В случае, если не поставить стрелочную функцию, метод obj.foo потеряет контекст и в качестве this использует саму функцию-таймер.

Метод foo.bind

Метод создаёт новую функцию, которая при вызове устанавливает в качестве контекста выполнения this предоставленное ей значение. В метод также можно передать «предустановленные» аргументы (частичное применение), т.е можно применить к функции не только контекст, но и аргументы.

Документация [здесь](#).

bind создаёт обёртку над функцией, которая подменяет контекст этой функции. Похоже на call и apply, но, в отличие от них, bind не вызывает функцию, а лишь возвращает "обёртку", которую можно вызвать позже. Все аргументы передаются к обёртке как если бы вызывалась исходная функция.

```
let boundFoo = foo.bind(this);  
let boundFoo = foo.bind(this, arg1, arg2...);
```

this

Значение, передаваемое в качестве this в целевую функцию при вызове привязанной функции

arg1, arg2...

Аргументы целевой функции («предустановленные»), передаваемые перед аргументами привязанной функции

В этом примере функции «foo», которая печатает this, через bind присваивается контекст:

```
const obj = { name: 'this is object' };  
  
function foo() {  
  console.log(this);  
}  
  
foo();  
// global или undefined  
  
const bindFoo = foo.bind(obj);  
bindFoo();  
// { name: 'this is object' }
```

Методы фиксируются точно так же:

```
const obj = {  
  name: 'this is object',  
  method() {  
    console.log(this.name)  
  }  
};  
  
const bindMethod = obj.method.bind(obj)  
  
bindMethod()  
// this is object
```

this и стрелочные функции

В учебнике не верно написано - стрелочная функция как раз имеет this и он четко определен контекстом, в отличии от обычных функций. Стрелочная функция "замыкает" в себе this, как если бы это был обычный аргумент в области видимости, где её объявили.

Об этом написано в документации для [this](#): «Также в ES2015 представлены стрелочные функции, которые не создают собственные привязки к this (они сохраняют значение this лексического окружения, в котором были созданы)».

Наглядная разница между обычной функцией и стрелкой:

- обычная функция при вызове через метод всплывает (?) и this для неё – глобальный объект.
- стрелочная функция запоминает свой this.

```
const obj = {
  name: 'obj',
  arrow() {
    return () => this();
  },
  common() {
    return (function() {
      return this
    })();
  },
}
console.log(obj.arrow());
// { name: 'obj' ... }
console.log(obj.common());
// Object [global]
```

#? Единственное, что я до сих пор не понял, почему в таймере this – это сама функция-таймер, но при этом таким же образом нельзя получить this обычной функции? В таймере всё работает не так, как в обычной функции.

Преобразование объектов в примитивы

Статья из старой редакции лучше и понятнее, лучше читать её: [ссылка](#).

Бывают операции, при которых объект должен быть преобразован в примитив:

Строковое – объект выводится через `alert()`.
Численное – при арифметических операциях и при сравнении с примитивом.
Логическое – при `if(obj)` и других логических операциях.

Вычитание одного объекта из другого – это обычно `Nan`. Но объекты даты можно вычесть друг из друга и получится разница во времени.

Алгоритм преобразований объектов к примитивам

Движок JavaScript пытается найти и вызвать:

если этот метод существует, то вызывает его и передаёт хинт
`obj[Symbol.toPrimitive](hint)`

Если символьного метода не существует:
Хинты «number» или «default» => `obj.valueOf()`
Хинты «string» => `obj.toString()`

Если такого метода нет, то вызывается второй. На практике достаточно реализовать только `obj.toString()`, достаточный для логирования или отладки.

Ниже подробнее обо всём.

Хинт

Это строковый аргумент number, default или string.

Не существует хинта со значением «boolean».

На практике все встроенные объекты, исключая Date, реализуют "default" преобразования тем же способом, что и "number".

string

Для преобразования объекта к строке, когда операция ожидает получить строку:

```
явное  
String(obj)
```

```
автоматом в браузере  
alert(obj)
```

```
при использовании объекта в качестве имени свойства  
anotherObj[obj] = 123;
```

```
при использовании интерполяции  
`${obj1}`
```

number

Для преобразования объекта к числу, в случае математических операций.

Оператор больше/меньше <> может работать со строками и числами. По историческим причинам он использует хинт «number».

```
явное  
let num = Number(obj);
```

```
математическое, кроме бинарного +  
let n = +obj;  
let delta = date1 - date2;
```

```
сравнения больше/меньше  
let greater = user1 > user2;
```

default

для некоторых операций.

Бинарный плюс + склеивает строки или складывает числа.

Нестрого сравнить объект == можно со строкой, числом или символом.

Для этих случаев есть дефолтный вариант:

```
бинарный плюс  
let total = car1 + car2;  
  
obj == string/number/symbol  
if (user == 1) { ... };
```

Symbol.toPrimitive

Метод обязан возвращать примитив, иначе будет ошибка.

(в отличие от методов toValue и toString, результат которых в случае не примитива будет проигнорирован).

[MDN](#).

Определение не понятное, проще посмотреть код.

В зависимости от хина, символ вернёт то или иное примитивное значение.

Функции, типа alert, или унарный +, вызывают этот символ и передают ему хинт: string, number или default.

```
const object1 = {
  [Symbol.toPrimitive](hint) {
    if (hint === 'number') {
      return 42;
    }
    return null;
  }
};

console.log(+object1);
// 42
```

.toString

.valueOf

Методы `toString` и `valueOf` берут своё начало с древних времён. Они не символы, так как в то время символов ещё не существовало, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода `Symbol.toPrimitive`, движок JavaScript пытается найти эти методы и вызывать их следующим образом:

`toString` -> `valueOf` для хинта со значением «`string`».

`valueOf` -> `toString` – в ином случае.

Получится то же поведение, что и с `Symbol.toPrimitive`.

Довольно часто мы хотим описать одно «универсальное» преобразование объекта к примитиву для всех ситуаций. Для этого достаточно создать один `toString`. В отсутствие `Symbol.toPrimitive` и `valueOf`, `toString` обработает все случаи преобразований к примитивам.

```
let user = {
  name: "John",

  toString() {
    return this.name;
  }
};

alert(user);      // toString -> John
alert(user + 500); // toString -> John500
```

Логическое преобразование

Любой объект в логическом контексте – `true`, даже если это пустой массив `[]` или пустой объект `{}`

```
Boolean( [] ) // = true
Boolean( {} ) // = true
```

Численное преобразование

Для численного преобразования объекта используется метод `valueOf`, а если его нет – то `toString`. Поэтому если переназначать `toString`, то он должен возвращать что-то примитивное. Если будет возвращено не примитивное значение, то результат проигнорируется.

У большинства встроенных объектов такого `valueOf` нет, поэтому «численное преобразование» для них равно «строковое преобразование».

Если посмотреть в стандарт, то в пункте [15.2.4.4](#) говорится, что метод `valueOf` есть у любых объектов. Но он ничего не делает, просто возвращает сам объект (не примитивное значение!), а потому игнорируется.

Исключением является объект Date, который поддерживает оба типа преобразований. Если к строке, то он вернёт дату в виде читаемой строки, если к valueOf, то кол-во мсекунд:

```
const date = new Date();

date.toString()
// Tue Jun 02 2020 23:19:14 GMT+0300 (GMT+03:00)

date.valueOf()
// 1591129154464
```

Строковое преобразование

Объект автоматом приводится к строке при некоторых функциях (например, alert, но не console.log). Явно преобразовать к строке объект можно через object.**toString()**

```
const obj = {
  key: 'value'
}

console.log(obj.toString())
// [object Object]
```

Метод .toString для конкретного объекта можно переназначить и сделать так, чтобы он выводил что угодно. Есть ограничение: это должен быть именно метод, а не простое свойство, потому что все штатные инструменты toString вызывают как функцию.

При этом результатом метода .toString может быть любой примитив. Например, это может быть число. Поэтому мы и называем его здесь «строковое преобразование», а не «преобразование к строке». Если будет возвращено не примитивное значение, то результат проигнорируется.

```
const obj = {
  key: 'value',

  toString() {
    return this.key
  }
}

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

Все объекты, включая встроенные, имеют свои реализации метода **toString**, например:

```
alert( [1, 2] );          // toString для массивов выводит строку элементов через , : "1,2"
alert( new Date() );      // toString для дат выводит дату в виде строки
alert( function() {} );  // toString для функции выводит её код
```

Требование к возвращаемым типам данных

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно требуемый «хинтом» тип примитива.

Нет обязательного требования, чтобы **toString()** возвращал именно строку, или чтобы метод **Symbol.toPrimitive** возвращал именно число для хинта «number».

Единственное обязательное требование: методы должны возвращать примитив, а не объект.

По историческим причинам, если `toString` или `valueOf` вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало).
Метод `Symbol.toPrimitive`, напротив, обязан возвращать примитив, иначе будет ошибка.

Дескрипторы свойств

Свойство – это не только пара «ключ-значение». У каждого свойства есть ещё и дескрипторы. Обычно они скрыты:

```
(ключ: значение) : дескрипторы
```

Просто так дескрипторы не получить и не записать. Для работы с ними нужны специальные методы:

```
Object.getOwnPropertyDescriptor()
Object.getOwnPropertyDescriptors(obj)

Object.defineProperty()
Object.defineProperties()
```

Подробная информация о типах дескрипторов свойств и их атрибутах может быть найдена в описании метода [Object.defineProperty\(\)](#).

Дескрипторы описывают «служебные» свойства самого свойства объекта. Например, будет ли оно перезаписываемым, перечисляемым в цикле, есть ли у него сеттеры или геттеры.

Дескрипторы бывают двух основных типов: дескрипторы данных и дескрипторы доступа. Дескриптор может быть только чем-то одним из этих двух типов; он не может быть одновременно обоими.

Фактически, дескриптор – это объект с набором определённых ключей.

Если ты сам создаёшь свойства при инициализации объекта или дозаписываешь их в объект, то значение этих ключей по умолчанию – `true`.

Если свойства создаются через метод `defineProperty`, то значение флагов по умолчанию – `false`.

Обязательные ключи

```
const descriptor = {
  enumerable: true/false,
  configurable: true/false,
}
```

enumerable

`true` – свойство будет перечисляться в циклах.

`false` – циклы и метод `Object.keys` будут игнорировать свойство.

configurable

`true` – свойство можно удалить, а флаги можно изменять.

`false`:

- нельзя менять флаги этого дескриптора, в т.ч. сам `configurable`.

- свойство нельзя удалить (изменять `value` можно). Попытки сделать это приведут к ошибке в строгом режиме.

Определение свойства как неконфигурируемого – это дорога в один конец. Для некоторых встроенных объектов `configurable` иногда установлен в `false`. Например, для свойства `Math.PI`

Ключи дескриптора данных

```
const descriptor = {
  enumerable: true/false,
  configurable: true/false,
  writable: true/false
  value: <значение обычного свойства объекта>
}
```

writable

true – свойство объекта можно изменить

false – свойство только для чтения. Попытка изменить его приведёт к ошибке в строгом режиме.

value

это обычное значение свойства объекта. Его можно изменять и добавлять через дескриптор.

Дескрипторы доступа

Это функции, которые выполняют роль геттеров и сеттеров.

На практике можно реализовать геттеры и сеттеры другими, более удобными способами. Обо всех способах написано ниже в разделе «Геттеры и сеттеры».

```
const descriptor = {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

Object.defineProperty(myObject, 'fullName', descriptor);

myObject.fullName = 'Денис Детров';
myObject.fullName // Денис Детров
```

Методы для дескрипторов

`Object.getOwnPropertyDescriptor()`
`Object.getOwnPropertyDescriptors(obj)`

`Object.defineProperty()`
`Object.defineProperties()`

`Object.getOwnPropertyDescriptor()`

Возвращает объект-дескриптор для собственного свойства объекта.

Собственное свойство – это которое находится в объекте, а не получено через цепочку прототипов.

```
descriptor = Object.getOwnPropertyDescriptor(obj, 'prop')
```

`obj`

Объект

`'prop'`

Это ключ, который нужно передать как строку

Пример:

```
const myObject = {
  key1: 'value1',
  key2: 'value2'
}

const descr = Object.getOwnPropertyDescriptor(myObject, 'key1');

{
  value: 'value1',
  writable: true,
  enumerable: true,
  configurable: true
}
```

Object.getOwnPropertyDescriptors(obj)

Получить все дескрипторы собственных свойств объекта сразу, включая свойства-символы.

Возвращается объект, в котором ключи – это наименование (ключи) исходного объекта, а значения этих ключей – объекты-дескрипторы.

`Object.getOwnPropertyDescriptors(obj)`

`obj`

Объект, чьи дескрипторы будут возвращены

Пример:

```
const myObject = {
  key1: 'value1',
  key2: 'value2'
};

const descr = Object.getOwnPropertyDescriptors(myObject);

{
  key1: {
    value: 'value1',
    writable: true,
    enumerable: true,
    configurable: true
  },
  key2: {
    value: 'value2',
    writable: true,
    enumerable: true,
    configurable: true
  }
}
```

Этот метод можно объединить с `Object.defineProperties`, чтобы клонировать объект вместе с дескрипторами. Обычное копирование объекта через `[key, value]` не копирует флаги, а так всё скопируется полностью, в т.ч. дескрипторы и свойства-символы:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Object.defineProperty()

Определяет новое или изменяет существующее свойство непосредственно на объекте.

Возвращает этот объект.

По умолчанию, ставит флагам дескриптора значение `false`, если их явно не прописать.

```
Object.defineProperty(obj, 'prop', descriptor)

obj
Объект, на котором определяется свойство

prop
Имя определяемого или изменяемого свойства

descriptor
Объект-дескриптор определяемого или изменяемого свойства.
```

Метод создаёт новое свойство с указанным `value` и флагами, или обновит флаги существующего свойства.

Можно перезаписывать все существующие свойства объекта, в т.ч. и скрытые.

Например, свойство `.toString` скрыто и не выводится при переборе в цикле, но к нему можно добраться через `.defineProperty()` и сделать так, чтобы оно было перечислимым.

Descriptor должен быть объектом, иначе будет ошибка. Например:

```
const descriptor = {
  value: 'new value',
  writable: true,
  enumerable: true,
  configurable: true,
}
```

В этом примере в объект добавляется новое свойство с флагами. Если флагам явно не прописать `true`, то их значением будет `false`:

```
const myObject = { key1: 'value1' };

const descriptor = {
  value: 'new value',
  writable: true,
  enumerable: true,
  configurable: true,
};

Object.defineProperty(myObject, 'newKey', descriptor);

myObject;
// { key1: 'value1', newKey: 'new value' }
```

Object.defineProperties()

Определяет новые или изменяет существующие свойства непосредственно на объекте, возвращая этот объект. То же самое, что и `.defineProperty`, но позволяет менять флаги сразу нескольких свойств.

```
Object.defineProperties(obj, {props})
```

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
});
```

```
const myObject = {};
const props = {
```

```
key1: {
  value: 'value1',
  writable: true,
  enumerable: true,
  configurable: false,
},
key2: {
  value: 'value1',
  writable: true,
  enumerable: true,
  configurable: false,
}
};

Object.defineProperties(myObject, props);

myObject;
// { key1: 'value1', key2: 'value1' }
```

Вместе с Object.getOwnPropertyDescriptors этот метод можно использовать для клонирования объекта вместе с его флагами. Обычное копирование объекта через [key, value] не копирует флаги, а так всё скопируется полностью, в т.ч. флаги и свойства-символы:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Глобальное запечатывание объекта

Дескрипторы свойств работают на уровне конкретных свойств.
Есть методы, которые ограничивают доступ ко *всему* объекту:

Object.preventExtensions(obj)

Запрещает добавлять новые свойства в объект.

Object.seal(obj)

Запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств.

Object.freeze(obj)

Запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.

А также есть **методы для их проверки**:

Object.isExtensible(obj)

Возвращает false, если добавление свойств запрещено, иначе true.

Object.isSealed(obj)

Возвращает true, если добавление/удаление свойств запрещено и для всех существующих свойств установлено configurable: false.

Object.isFrozen(obj)

Возвращает true, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено configurable: false, writable: false.

Дескрипторы доступа

Дескриптор аксессора может иметь:

get() - функция без аргументов, которая сработает при чтении свойства
set(value) - функция, принимающая один аргумент, вызываемая при присвоении свойства,
Записываются они также через `Object.defineProperty()`

В пример ниже добавляется геттер/сеттер `fullname`. Он не перечисляется, не выводится при печати объекта, но к нему можно обращаться и использовать. После использования всего одного геттера, перезапишутся одновременно два свойства объекта: `this.name` и `this.surname`

```
const myObject = { name: 'Glad', surname: 'Valakas' };

const descriptor = {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

Object.defineProperty(myObject, 'fullName', descriptor);

myObject.fullName = 'Денис Детров';
myObject.fullName; // Денис Детров
```

Использование для совместимости

См. учебник [ТУТ](#).

Умные геттеры/сеттеры

В геттеры и сеттеры можно добавлять инструкции для проверки значений, а само значение хранить в отдельном свойстве `_name`. Свойство `_name` изначально не создаётся и к нему ничто не обращается, кроме геттера-сеттера.

Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью `user._name`, но существует широко известное соглашение: если перед свойством стоит нижнее подчёркивание `_`, значит что к нему не рекомендуется обращаться извне.

```
const myObject = {
  get name() {
    return this._name;
  },
  set name(value) {
    if (value.length < 4) {
      console.log('д.б. более 4 символов');
      return;
    }
    this._name = value;
  }
};

myObject.name = "Glad";
console.log(myObject.name);
// Glad

myObject.name = "PD";
// 'д.б. более 4 символов'

console.log(myObject)
// { name: [Getter/Setter], _name: 'Glad' }
```

То же самое, пример из курсеры с дескриптором:

```
const tweet = {
  _likes: 16
};

Object.defineProperty(tweet, 'likes', {
  get: function() {
    return this._likes;
  },
  set: function(value) {
    this._likes = parseInt(value) || 0;
  }
});

tweet.likes; // 16

tweet.likes = 17;
```

Геттеры и сеттеры

Геттеры и сеттеры ещё называют свойства-аксессоры (accessor properties). По сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта. При литеральном объявлении объекта они обозначаются `get` и `set`

Геттеры – методы для извлечения данных из объекта.

Подставить перед методом «`get`»

Сеттеры – методы для изменения данных в объекте

Подставить перед методом «`set`»

Есть несколько способов реализовать дескрипторы:

1. При литеральном создании

Они создаются как обычные свойства при литеральном создании объекта с префиксами `get` и `set`.

При использовании свойство-аксессор выглядит как обычное свойство, **не метод!**

Геттер не вызывается как функция через скобки `()`. Обращение к геттеру – это как обращение к обычному свойству, типа `.length`. При создании функции слева ставится оператор `get`.

Сеттер тоже не имеет скобок при использовании, ему просто присваивается значение через оператор `«=»`. При создании слева ставится оператор `set`.

В примере ниже я их записал в 2 разных свойства: `getFullName` и `setFullName` для удобства. На самом деле, можно назвать их одним и тем же именем и использовать его для обеих операций:

```
const myObject = {
  name: 'Glad',
  surname: 'Valakas',
  get getFullName() {
    return `${this.name} ${this.surname}`;
  },
  set setfullName(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

// работают как обычные свойства, без вызовов ()
```

```
myObject.setfullName = 'Денис Детров';
myObject.getFullName; // Денис Детров
```

2. Через дескриптор доступа

Дескриптор доступа добавляется только через `Object.defineProperty()` и может иметь:

`get()` - функция без аргументов, которая сработает при чтении свойства

`set(value)` - функция, принимающая один аргумент, вызываемая при присвоении свойства,

В пример ниже добавляется геттер/сеттер `fullname`. Он не перечисляется, не выводится при печати объекта, но к нему можно обращаться и использовать. После использования всего одного геттера, перезапишутся одновременно два свойства объекта: `this.name` и `this.surname`

```
const myObject = { name: 'Glad', surname: 'Valakas' };

const descriptor = {
  get() {
    return `${this.name} ${this.surname}`;
  },
  set(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

Object.defineProperty(myObject, 'fullName', descriptor);

myObject.fullName = 'Денис Детров';
myObject.fullName; // Денис Детров
```

3. Обычными методами

Объявляются как функции, вызываются как функции: круглые скобки и всё такое:

```
const myObject = {
  name: 'Glad',
  surname: 'Valakas',
  getFullName() {
    return `${this.name} ${this.surname}`;
  },
  setFullName(value) {
    [this.name, this.surname] = value.split(' ');
  }
};

myObject.setFullName('Денис Детров');
myObject.getFullName();
// Денис Детров
```

```
const myObject = {
  name: 'Glad',
  surname: 'Valakas',
  getFullName() {
```

```

        return `${this.name} ${this.surname}`;
    },
    setFullName(value) {
        [this.name, this.surname] = value.split(' ');
    }
};

myObject.setFullName('Денис Детров');
myObject.getFullName();
// Денис Детров

```

Фишки с объектами

Задача

Реализуйте функцию pick, которая извлекает из переданного объекта все элементы по указанным ключам и возвращает новый объект. Аргументы:

1. Исходный объект

2. Массив ключей, по которым должны быть выбраны элементы (ключ и значение) из исходного объекта, и на основе выбранных данных сформирован новый объект

```

const pick = (obj, arr) => {
    const result = {};
    const pairs = Object.entries(obj);

    for (const [key, value] of pairs) {
        if (arr.includes(key)) {
            result[key] = value;
        }
    }
    return result;
};

```

Копия массива с объектами внутри:

```

var arr = [ { key1: 'value1', key2: 'value2' } ]

var copy = arr.map(function(element) {
    var newObject = {};
    var originalObjectKeys = Object.keys(element);
    var originalObjectValues = Object.values(element);

    for (var i = 0; i < originalObjectKeys.length; i += 1) {
        var currentKey = originalObjectKeys[i];
        var currentValue = originalObjectValues[i];
        newObject[currentKey] = currentValue;
    }
    return newObject;
})

```

Пересечение по значениям свойств

Иногда надо найти из нескольких объектов такие, у которых есть общие свойства (значения свойств). Можно это сделать так:

```

const arr1 = [
    { key1: 'value1', id: 2, key2: 'value2'},
    { key1: 'value1', id: 8, key2: 'value2'},
    { key1: 'value1', id: 100, key2: 'value2'}
]

```

```
];
const arr2 = [
  { key1: 'value1', id: 2, key2: 'value2' },
  { key1: 'value1', id: 100, key2: 'value2' },
  { key1: 'value1', id: 7, key2: 'value2' }
];

const commonID = arr1.filter(
  (element1) => arr2.some(
    (element2) =>
      element2.id === element1.id));
[ { key1: 'value1', id: 2, key2: 'value2' },
  { key1: 'value1', id: 100, key2: 'value2' } ]
```

</>

Конструкторы объектов, "new"

Много однотипных объектов можно создать при помощи функции-конструктора и оператора "new". Для создания сложных объектов есть и более «продвинутый» синтаксис – [классы](#).

Функции-конструкторы являются обычными функциями. Два требования к конструкторам:

1. Имя начинается с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Синтаксис кратко:

```
function User(name) {
  // создание свойства
  this.name = name;

  // создание метода
  this.read = function() {
    console.log( "Меня зовут: " + this.name );
  };
}

// создание
let user = new User('Valakas');
```

Как работает?

Такой вызов через «new» создаёт пустой `this` в начале выполнения и возвращает заполненный в конце:

1. Создаётся новый пустой объект, и он присваивается `this`.
2. Выполняется код функции. Обычно он модифицирует `this`, добавляет туда новые свойства.
3. Возвращается значение `this`.

Когда нам необходимо будет создать других пользователей, мы можем использовать `new User("Маша")`. Данная конструкция удобнее, чем каждый раз создавать литерал объекта.

Отсутствие скобок ()

Можно не ставить скобки после new, если вызов конструктора идёт без аргументов. Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

```
let user = new User;  
// то же, что и  
let user = new User();
```

Обычно у конструкторов нет return.

Конструкторы ничего не возвращают явно, только заполняют this. Но если return всё же есть, то применяется простое правило:

1. При вызове return с объектом, будет возвращён объект, а не this.
 2. При вызове return с примитивным значением, примитивное значение будет отброшено.
- #? Вот это я не понял.

Проверка на вызов в режиме конструктора: new.target

Свойство new.target позволяет определить была ли функция или конструктор вызваны с помощью оператора new. [MDN](#).

В случае, если функция вызвана при помощи new, то в new.target будет сама функция, в противном случае undefined.

Свойство new.target это мета свойство которое доступно во всех функциях. В стрелочных - new.target ссылается на new.target внешней функции.

Это можно использовать, чтобы отличить обычный вызов от вызова «в режиме конструктора».

```
function Foo() {  
  if (!new.target) throw "Foo() must be called with new";  
  console.log("Foo instantiated with new");  
}  
  
new Foo(); // выведет "Foo instantiated with new"  
Foo(); // ошибка "Foo() must be called with new"
```

Прототипы

Прототипное наследование — это возможность языка, которая позволяет создавать одни объекты на основе других. Learn.js [здесь](#)

Объекты имеют скрытое свойство [[Prototype]], которое либо равно null, либо ссылается на другой родительский объект-прототип.

Прототипное наследование – механизм, когда отсутствующее в объекте свойство берётся из прототипа.

Перечисление всех методов и свойств из этой главы:

```
obj.__proto__  
Object.create(proto, [descriptors])  
Object.getPrototypeOf(obj)  
Object.setPrototypeOf(obj, prototype)
```

```
Function.prototype  
Function.prototype.constructor  
instance.constructor  
instance[[Prototype]]
```

Методы для прототипов

Свойство `[[Prototype]]` является скрытым, поэтому обращение к нему и изменение осуществляется через методы.

`obj.__proto__`

Это геттер/сеттер для `[[Prototype]]`. По спецификации `__proto__` должен поддерживаться только браузерами, но по факту его поддерживают все среды. Это API не было стандартизовано.

Внимание! Это не то же самое, что `[[Prototype]]`.

`Object.getPrototypeOf(obj)`

Метод возвращает прототип (то есть, внутреннее свойство `[[Prototype]]`) указанного объекта.

`Object.setPrototypeOf(obj, prototype)`

Метод устанавливает прототип (то есть, внутреннее свойство `[[Prototype]]`) указанного объекта в другой объект или `null`.

`obj` Объект, которому устанавливается прототип
`prototype` Новый прототип объекта (объект или `null`)

`Object.create(proto, [descriptors])`

Создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto`, и необязательными дескрипторами свойств `descriptors`.

`proto` Объект, который станет прототипом вновь созданного объекта
`descriptors` Необязательный. Дескрипторы создаваемых свойств.

В примере создаётся новый объект, которому назначается прототип и создаётся новое свойство «`jumps`»:

```
let animal = {  
    eats: true  
};  
  
let rabbit = Object.create(animal, {  
    jumps: {  
        value: true  
    }  
});
```

`Object.create` можно использовать для копирования объекта.

Такой вызов создаёт точную копию объекта `obj`, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством `[[Prototype]]`.

Тут комбинация сразу трёх методов:

- `Object.create`
- `Object.getPrototypeOf`
- `Object.getOwnPropertyDescriptors`

```
let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));
```

Свойство `[[Prototype]]`

Как было сказано ранее, смысл прототипов – это наследование свойств. Если у объекта нет какого-то свойства, то движок JS ищет это свойство в прототипе объекта. У прототипа может быть свой прототип и так далее по цепочке. Объект `child` можно так же назначить прототипом для третьего объекта, и третьему будут доступны все свойства `parent`.

Такие свойства называются **унаследованными**.

Есть несколько ограничений при создании прототипов:

1. Ссылки не могут идти по кругу, JavaScript выдаст ошибку;
2. Прототип – это либо Object, либо null. Другие типы игнорируются;
3. У объекта может быть только один прототип, он не может наследовать от двух одновременно.

В примере ниже объекту child будут доступны все свойства parent:

```
let parent = { eats: true };

let child = { jumps: true };

child.__proto__ = parent;    // назначить прототип для child
child.__proto__           // геттер прототипа

let child = {
  jumps: true,
  __proto__: parent
};
```

В примере выше я назначил прототип вне объекта. Альтернативный способ – назначить прототип прямо в объекте, но я не уверен, что так правильно делать.

Прототип используется только для чтения свойств. Операции записи новых свойств и удаления существующих работают напрямую с объектом, но не с прототипом.

Если дочернему объекту прописать такое же свойство, какое есть у прототипа, то это свойство добавится в дочерний элемент. Свойство в прототипе не перезапишется.

Работа с this и другими операторами

Если мы вызываем obj.method(), а метод при этом взят из прототипа, то this всё равно ссылается на obj. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.

Поиск свойства и исполнение кода – два разных процесса:

Методы, использующие this, ищут свойство в дочернем объекте. Если не находят – ищут в прототипе. Сеттеры с this записывают по общему правилу: в объект перед точкой, т.е. не в прототип.

Операция delete применяется только к собственным свойствам объекта. Если свойства нет в дочернем объекте, но есть в прототипе, то ничего не произойдёт.

Методы Object.keys, .values и прочие тоже работают только с собственными свойствами объекта.

Цикл for...in (не of!)

Цикл перебирает и собственные, и унаследованные свойства объекта.

Чтобы сделать обход через for...in с собственными ключами, можно использовать проверку методом obj.hasOwnProperty(key):

```
for (let i in child) {
  if (!child.hasOwnProperty(i)) {
    continue;
  }
  console.log(i)
}
```

Все методы, которые доступны объектам из коробки, наследуются от Object.prototype. Это «верховный» объект в иерархии. Эти методы не отображаются в цикле for..in, как другие унаследованные свойства, потому что им прописаны соответствующие дескрипторы: они неперечисляемые, внутренний флаг enumerable = false. Почти все остальные методы получения ключей/значений игнорируют унаследованные таким образом свойства.

Итого:

- В JavaScript все объекты имеют скрытое свойство `[[Prototype]]`, которое является либо другим объектом, либо `null`.
- Объект, на который ссылается `[[Prototype]]`, называется «прототипом».
- Если мы хотим прочитать свойство `obj` или вызвать метод, которого не существует у `obj`, тогда JavaScript попытается найти его в прототипе.
- Операции записи/удаления работают непосредственно с объектом, они не используют прототип (если это обычное свойство, а не сеттер).
- Если мы вызываем `obj.method()`, а метод при этом взят из прототипа, то `this` всё равно ссылается на `obj`. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.
- Цикл `for..in` перебирает как свои, так и унаследованные свойства. Остальные методы получения ключей/значений работают только с собственными свойствами объекта.

Конструктор и прототип

Глава в учебнике находится [тут](#) и называется `Foo.prototype`.

В конспекте я ещё что-то написал от себя.

Важно! Свойство функции `.prototype` и геттер-сеттер `__proto__` – это разные вещи. У функции есть свой прототип и он доступен через обычные геттеры.

Объекты можно создавать с помощью функций-конструкторов.

У функции-конструктора есть свойство `Function.prototype`. Это именно наименование ключа. По этому ключу хранится объект, который будет автоматически назначаться в качестве прототипа для вновь создаваемых конструктором объектов. Экземпляры объекта `Function` (т.е. функции-конструктора) наследуют методы и свойства из объекта `Function.prototype`. У обычных объектов свойства `.prototype` нет.

По умолчанию этот прототип имеет только одно свойство: `.constructor`. По этому ключу записана ссылка на функцию-конструктор.

MDN:

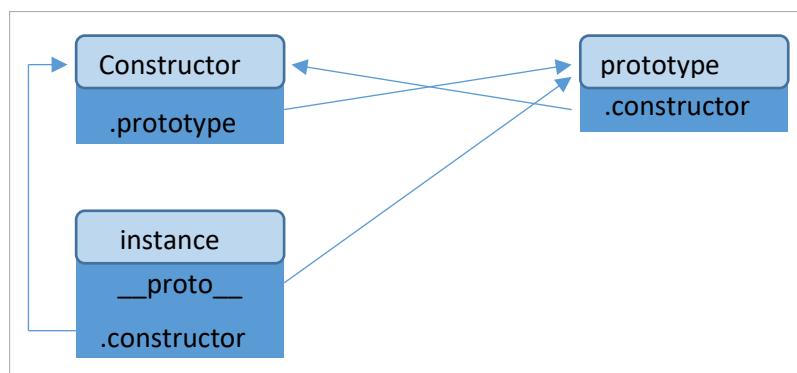
[Object.prototype](#)

Свойство представляет объект прототипа `Object`. В документации написано `Object`, но речь идёт именно о функции-конструкторе `Object()`, потому что обычные объекты не имеют свойства `.prototype`.

[Object.constructor](#)

Свойство возвращает ссылку на функцию [Object](#), создавшую прототип экземпляра. Обратите внимание, что значение этого свойства является ссылкой на саму функцию, а не строкой, содержащей имя функции. Для примитивных значений, вроде `1`, `true` или `"test"`, значение доступно только для чтения.

Все объекты наследуют свойство `constructor` из своего прототипа.



Поскольку этот объект будет назначаться вновь создаваемым объектам в качестве прототипа, то все создаваемые объекты также будут иметь свойство `.constructor`.

Если у тебя есть какой-то инстанс и ты не знаешь, как называется его конструктор, то можно вызвать конструктор прямо из инстанса и создать новый объект:

```

function Func() {}
const instance = new Func();

Func === Func.prototype.constructor; // true
Func === instance.constructor;      // true
  
```

```
const instance2 = new instance.constructor();
Func === instance2.constructor; // true
```

Конструкторы работают следующим образом:

1. прописывают создаваемому объекту будущий прототип из Function.prototype;
2. дозаписывают в создаваемый объект остальные свойства из своего тела;
3. создают новый объект.

Объект, указанный в Function.prototype, можно **заменять**.

Установка Function.prototype = object буквально говорит следующее: при создании объекта через new Function() запиши ему object в свойство [[Prototype]].

Объект по ссылке Function.prototype используется только в момент создания инстанса. Если этот объект в дальнейшем заменить на другой (т.е. указать в свойстве Function.prototype другой объект), то он по прежнему останется прототипом для старых инстансов, но не для новых.

Если прототип менялся, важно вручную прописать новому объекту-прототипу свойство .constructor, иначе не будет ссылки на функцию-конструктор.

Ещё один способ переназначить прототип для вновь создаваемых через конструктор объектов – внутрь конструктора записать this.__proto__, но я не уверен, что так правильно делать.

Рекомендуется не менять прототип после создания объекта. Двигок JS хорошо оптимизирован в том случае, когда прототип не меняется. Object.setPrototypeOf или obj.__proto__ = – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого.

Дополнение в прототип новых свойств.

В объект в Function.prototype можно добавлять и удалять новые свойства, которые будут доступны всем ранее созданным инстансам. Этот способ предпочтительнее, чем его полная замена.

Примеры замены объекта-прототипа.

```
const parent = { eats: true, д.б. ещё свойство .constructor };

function Function(name) {
  this.name = name;
  //this.__proto__ = parent;
}

Function.prototype = parent;

const instance = new Function("Name");
instance.eats; // true
```

Прототипы встроенных объектов

Все встроенные объекты следуют одному шаблону:

- Методы хранятся в прототипах (Array.prototype, Object.prototype, Date.prototype и т.д.).
- Сами объекты хранят только данные (элементы массивов, свойства объектов, даты).

Примитивы также хранят свои методы в прототипах объектов-обёртках: Number.prototype, String.prototype, Boolean.prototype. Только у значений undefined и null нет объектов-обёрток.

Встроенные прототипы могут быть изменены или дополнены новыми методами. Но не рекомендуется менять их. Единственная допустимая причина – полифил, т.е. добавление нового метода из стандарта, который ещё не поддерживается движком JavaScript.

Объект всегда создаётся с помощью обычной функции-конструктора. Использование краткой нотации – это просто синтаксический сахар, всё равно работает конструктор:

```
obj = new Object()
obj = {}
```

Функция-конструктор, как обычно, имеет свойство `.prototype`. Этот `Object.prototype` – объект, который хранит все встроенные в JS функции, которые подтягиваются к созданные объекты. Когда вызывается `new Object()`, свойство `[[Prototype]]` этого объекта устанавливается на `Object.prototype` по правилам, которые обсуждались в предыдущей части. У самого `Object.prototype` своего `.prototype` нет, он равен `'null'`.

В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования через `console.dir`.

`Array, Date, Function` – это всё объекты и они имеют свои конструкторы и прототипы.

Например, при создании массива `[1, 2, 3]` используется конструктор `Array`, поэтому прототипом массива становится `Array.prototype`, который и предоставляет свои методы.

В JS всё наследует от объектов. Получается такая иерархия:

```
          null
          |
Object.prototype
|
Array.prototype, Function.prototype, Number.prototype ...
|
[1, 2, 3]           foo()           179
```

Некоторые методы в прототипах могут пересекаться.

У `Array.prototype` есть свой метод `toString`, который выводит элементы массива через запятую, а у `Object.prototype` метод `toString` тоже есть и работает по-другому. Инстанс берёт ближайший метод, поэтому ничего страшного не происходит.

Примитивы

Хоть это и не объекты, но если мы попытаемся получить доступ к их свойствам, то тогда будет создан временный объект-обёртка с использованием встроенных конструкторов `String`, `Number` и `Boolean`, который предоставит методы и после этого исчезнет. Методы этих объектов также находятся в прототипах, доступных как `String.prototype`, `Number.prototype` и `Boolean.prototype`.

Как было сказано ранее, значения `null` и `undefined` не имеют объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

Изменение встроенных прототипов

Встроенные прототипы можно изменять. Если что-то добавить к встроенному прототипу для строк, то все строки будут иметь этот метод. Если что-то удалить из общего прототипа, то свойство исчезнет.

Добавлять и изменять методы считается плохой практикой, потому что разные библиотеки могут переписать или добавить один и тот же метод и будет коллизия.

Единственное исключение – это полифили.

```
if (!String.prototype.repeat) {
  // Если такого метода нет, то добавляем его в прототип

  String.prototype.repeat = function(n) {
    return new Array(n + 1).join(this);
  };
}

console.log( "La".repeat(3) ); // LaLaLa
```

на самом деле код должен быть немного более сложным, полный алгоритм можно найти в спецификации, но даже неполный полифил зачастую достаточно хорош для использования.

Заимствование методов у прототипов

В главе [Декораторы и переадресация вызова, call/apply](#) мы говорили о заимствовании методов, когда мы берём метод из одного объекта и копируем его в другой.

С прототипами есть такая же практика. Можно одолжить какие-то методы, которые есть у встроенных прототипов, для невстроенных объектов. Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

Например, если мы создаём объект, похожий на массив (псевдомассив), мы можем скопировать некоторые методы из Array в этот объект. Например, .join, потому что для алгоритма join важны только корректность индексов и свойство length, он не проверяет, является ли объект на самом деле массивом. И многие встроенные методы работают так же:

```
let obj = {  
  0: "Hello",  
  1: "world!",  
  length: 2,  
};  
  
obj.join = Array.prototype.join;  
  
console.log( obj.join(',') ); // Hello,world!
```

Ультра вариант – унаследовать все вообще методы, указав в качестве прототипа объекта – Array.prototype: obj.__proto__ = Array.prototype. Но Важно помнить, что прототип у инстанса может быть только один.

Объекты без прототипа

Если хочется создать простейший объект, то можно создать объект и прописать ему прототип null.

```
let obj = Object.create(null);
```

Недостаток в том, что у таких объектов не будет встроенных методов объекта. Но обычно это нормально для ассоциативных массивов. Кроме того, большая часть методов, связанных с объектами, имеют вид свойства Object.something(...), как Object.keys(obj). Подобные методы не находятся в прототипе (они - свойства Object), так что они продолжат работать для таких объектов.

Такие чистые объекты нужны для того, чтобы избежать уязвимостей.

В случае наследования, объект получает сеттер **proto** и это представляет уязвимость, потому что кто-то может сделать так:

```
let obj = {};  
  
let key = prompt("What's the key?", "__proto__");  
obj[key] = "some value";  
  
console.log(obj[key]);  
// [object Object], не "some value"
```

Если мы присваиваем объектные значения, то прототип будет изменён. В результате дальнейшее выполнение пойдёт совершенно непредсказуемым образом. Это типа уязвимость. Неожиданные вещи могут случаться также при присвоении свойства toString, которое по умолчанию функция, и других свойств, которые тоже на самом деле являются встроенными методами.

Встроенный геттер/сеттер __proto__ не безопасен, если мы хотим использовать созданные пользователями ключи в объекте. Как минимум потому, что пользователь может ввести "__proto__" как ключ. Так что мы можем использовать либо Object.create(null) для создания «простейшего» объекта, либо использовать коллекцию Map.

</>

Udemy, прототипы

Если у нескольких объектов есть одно и то же свойство, имеет смысл его вынести в прототип. Сделать это можно несколькими способами:

1. С помощью свойства Object.setPrototypeOf().

Использование свойства Object.setPrototypeOf() очень ресурсоёмко, его не рекомендуется использовать даже MDN.

```
const animal = {
  say: function() {
    console.log(` ${this.name} say ${this.voice}`);
  }
};

const dog = {
  name: 'dog',
  voice: 'woof'
};

const cat = {
  name: 'cat',
  voice: 'meow'
};

Object.setPrototypeOf(dog, animal);
Object.setPrototypeOf(cat, animal);
```

2. С помощью Object.create()

Вместо этого можно использовать метод Object.create.

Из минусов – придётся переписывать все свойства вручную.

```
const animal = {
  say: function() {
    console.log(` ${this.name} say ${this.voice}`);
  }
};

const dog = Object.create(animal);
dog.name = 'dog';
dog.voice = 'woof';
```

3. Через обычную функцию

Чтобы не вбивать все свойства вручную, можно сделать функцию, которая будет создавать и возвращать новый объект с использованием Object.create():

```
const animal = {
  say: function() {
    console.log(` ${this.name} say ${this.voice}`);
  }
};

function createAnimal(name, voice) {
  const result = Object.create(animal);
  result.name = name;
  result.voice = voice;
  return result;
}
```

```
}
```

```
const dog = createAnimal('dog', 'woof');
const cat = createAnimal('cat', 'meow');
```

4. Через функцию-конструктор

Ещё удобнее будет использовать синтаксис функции-конструктора. Их имена пишутся с большой буквы, а при использовании нужно ставить ключевое слово «new». Кажется, что это то же самое, но нет:

1. JS будет оптимизировать процесс создания нового объекта, не надо использовать Object.create();
2. Ссылки на свойства создаваемого объекта (в примере выше это был result) теперь даются через ключевое слово this.property.
3. Новый объект не надо возвращать через return.
4. Методы можно дописывать вручную через свойство Animal.prototype (а можно сразу записывать в функцию-конструктор). Прототип создаётся автоматом для конструктора.

```
function Animal(name, voice) {
  this.name = name;
  this.voice = voice;
}

Animal.prototype.say = function() {
  console.log(` ${this.name} say ${this.voice}`);
}

const dog = new Animal('dog', 'woof');
const cat = new Animal('cat', 'meow');
```

Тем не менее, этот паттерн не очень хороший по нескольким причинам:

- Мы показываем связи между объектами вместо того, чтобы показывать смысл. А смысл очень простой: у всех объектов Animal должно быть имя, голос и функция.
- Наследование и управление цепочками прототипов. Работая с прототипами напрямую, код для вызова функции по цепочке прототипов выше выглядит сложно и громоздко.

Синтаксис классов – это синтаксический сахар, делающий работу с прототипами просто более удобную.

Классы

Классы в справочнике MDN [здесь](#).

Свойства могут записываться:

- в экземпляры класса (собственные свойства);
- в прототип (унаследованные свойства);
- в функцию-конструктор (статические свойства).

Класс в объектно-ориентированном программировании – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

На практике приходится создавать много однотипных объектов (пользователи, товары и т.д.). С этим позволяет справляться конструктор new Foo(), но есть более продвинутый синтаксис: класс. Класс – это надстройка над конструктором, которая позволяет сразу же закидывать методы в прототип создаваемых объектов, а не делать это отдельными операциями через Foo.prototype и instance._proto.

Если прототип (который _proto) при создании через конструктор имел только один метод .constructor, то в классах ему добавляются все остальные необходимые методы «из коробки».

Отличия классов от обычных конструкторов:

1. функция, созданная с помощью class, помечена специальным внутренним свойством `[[FunctionKind]]:"classConstructor"`. У созданных вручную функций такого свойства нет.
2. Методы класса являются неперечислимыми. Определение класса устанавливает флаг `enumerable` в `false` для всех методов в "prototype". В отличие от обычного конструктора.
3. Классы всегда используют `use strict`. Весь код внутри класса автоматически находится в строгом режиме.
4. Отличие от функций – классы не всплывают (проверить самому).

В остальном, `class` – это «синтаксический сахар» (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что мы можем сделать всё то же самое без конструкции `class`, докинув нужные методы в прототип отдельной строкой.

Дописать

При объявлении класса, функции (методы), объявленные внутри, уходят в прототип. Если метод использует `this`, то, в отрыве от инстанса, он может не найти нужное свойство. Например, так теряется `this.props` в React.

Чтобы этого избежать, можно использовать поля класса (новая фишка) или конструктор (старая, не правильно).

```
// метод уходит в прототип и потеряет контекст
// будет работать только с bind
class TodoListItem extends Component {

  onLabelClick() {
    console.log('Done: ${this.props.label}');
  }

  render() {
    <span
      onClick={ this.onLabelClick } />
  }
}

// прописать в конструкторе
// всё, что указывается в конструкторе, будет храниться в инстансе, а не прототипе
class TodoListItem extends Component {

  constructor() {
    super();
    this.onLabelClick() = () => {
      console.log('Done: ${this.props.label}');
    };
  }

  render() {
    <span
      onClick={ this.onLabelClick } />
  }
}

// поля класса
// методы, объявленные через поля, останутся в инстансе, а не прототипе
class TodoListItem extends Component {

  onLabelClick() = () => {
    console.log('Done: ${this.props.label}');
  }
}
```

```

};

render() {
  <span
    onClick={ this.onLabelClick } />
}

```

Udemy, классы

- В классе первым делом создаётся конструктор объекта.
- Ключевое слово extends говорит о том, что данный класс включается в цепочку прототипов расширяемого класса.
- Если extends-объекту не создать свой конструктор, то он будет наследовать конструктор прототипа.

```

class Animal {
  constructor(name, voice) {
    this.name = name;
    this.voice = voice;
  }

  say() {
    console.log(` ${this.name} say ${this.voice}`);
  }
}

class Bird extends Animal {
}

// duck --> Bird.prototype --> Animal.prototype --> Object.prototype --> null
const duck = new Bird('duck', 'quack');

```

Если для дочернего класса надо добавить дополнительную функциональность и ему понадобится конструктор, то внутри этого конструктора первой строкой должен сначала вызываться родительский конструктор через ключевое слово super.

```

class Animal {
  constructor(name, voice) {
    this.name = name;
    this.voice = voice;
  }

  say() {
    console.log(` ${this.name} say ${this.voice}`);
  }
}

class Bird extends Animal {
  constructor(name, voice, canFly) {
    super(name, voice);
    this.canFly = canFly;
  }
}

const duck = new Bird('duck', 'quack', true);

```

Ключевое слово super даёт доступ не только к super-конструктору, но и к любому методу из super-класса.

```
class Bird extends Animal {  
    constructor(name, voice, canFly) {  
        super(name, voice);  
        super.say();  
        this.canFly = canFly;  
    }  
}
```

Если необходимо, в дочернем классе можно полностью переопределить метод родительского класса.

В примере ниже переопределяется метод say(). При этом сначала сработает родительский метод, к которому обратились через super, а потом дочерний, переопределённый.

```
class Animal {  
    constructor(name, voice) {  
        this.name = name;  
        this.voice = voice;  
    }  
  
    say() {  
        console.log(`${this.name} say ${this.voice}`);  
    }  
}  
  
class Bird extends Animal {  
    constructor(name, voice, canFly) {  
        super(name, voice);  
        super.say();  
        this.canFly = canFly;  
    }  
  
    say() {  
        console.log('birds don\'t like to talk');  
    }  
}  
  
const duck = new Bird('duck', 'quack', true);  
duck.say();  
  
// duck say quack  
// birds don't like to talk
```

Свойства класса

Можно инициализировать свойства прямо в теле класса.

Если свойства не зависят от внешних параметров, не имеет смысла передавать их через конструктор, можно сразу записать их в тело класса. Так же можно создавать методы.

```
class Counter {  
    count = 1;  
  
    inc = () => {  
        this.count += 1;  
        console.log(this.count);  
    }  
}
```

```
let a = new Counter();
a.inc(); // 2

let b = new Counter();
b.inc(); // 2

setTimeout(a.inc, 1000); // 3
```

Если в примере выше создать ещё инстансы этого класса, то свойство count для каждого из них будет своим, независимым.

Статические свойства

Статические свойства и методы принадлежат всему классу. Правило простое: обычные свойства – для конкретных экземпляров, статические – для всех экземпляров.

Обращение к ним – ClassName.property

```
class Counter {
  count = 1;

  inc = () => {
    this.count += Counter.incStep;
    console.log(this.count);
  }

  static incStep = 2;

  static incrementAll = function(arr) {
    arr.forEach(instance) => instance.increment();
  };
}
```

Пример выше можно написать на старом синтаксисе, без static.

Те свойства, которые инициализируются в теле класса и принадлежат конкретному экземпляру, можно создать в конструкторе. Что касается методов экземпляров, то Arrow-функция сохранит лексический this – это сам объект, который мы создаём.

Статические поля и функции выносятся за конструктор и прописываются вручную для конструктора класса. Функции – это тоже объекты, поэтому им можно записать свойства.

```
class Counter {

  constructor() {
    this.count = 0;
    this.increment = () => {
      this.count += Counter.incrementStep;
    }
  }

  Counter.incrementStep = 2;

  Counter.incrementAll = function(arr) {
    arr.forEach(inst) => inst.increment();
  }
}
```

Весь синтаксис кратко

Базовый. Конструктор вызывается автоматом при создании инстанса, в него передаются аргументы, которые становятся собственными свойствами создаваемого инстанса.

Между методами не ставится запятая.

```
class MyClass {  
  
    constructor(name, color) {  
        this.name = name;  
        this.color = color;  
    }  
  
    property = "Value";  
  
    method() {  
        console.log(this.name);  
    }  
    method2() {  
        console.log(this.color);  
    }  
}  
  
// Использование:  
let instance = new MyClass('Иван', 'Red');  
instance.method();
```

Геттеры и сеттеры

```
class MyClass {  
    constructor(name) {  
        // вызывает сеттер instance.name =  
        this.name = name;  
    }  
  
    get name() {  
        return this._name;  
    }  
  
    set name(value) {  
        this._name = value;  
    }  
}
```

Наследование классов, переопределение конструктора и переопределение метода:

```
class Parent {  
    constructor(name) {  
        this.name = name;  
    }  
  
    method() {  
        return `${this.name} стоит`;  
    }  
}  
  
class Child extends Parent {  
    constructor(name, surname) {  
        super(name);  
        this.surname = surname;  
    }  
}
```

```
}

method() {
  return super.method() + ` и ${this.surname} прячется`;
}
};

const instance = new Child('Валера', 'Жмых');
console.log( instance.method() ); // Валера стоит и Жмых прячется
```

Ключевое слово `super` используется для вызова функций, принадлежащих родителю объекта.

Документация [здесь](#).

```
super([arguments]);
// вызов родительского конструктора

super.functionOnParent([arguments]);
```

Ключевое слово `static` определяет статический метод класса. Обращение к ним происходит через `конструктор.метод`, они не могут быть вызваны у экземпляров (`instance`) класса. Статические методы часто используются для создания служебных функций для приложения.

```
class MyClass {
  static method() {
    console.log('this is static method');
    return;
  }
}

MyClass.method()
// this is static method
```

Это фактически то же самое, что присвоить метод напрямую как свойство функции:

```
MyClass.staticMethod = function() {...};
```

Создание приватного свойства, которое не наследуется и недоступно извне функции-конструктора класса:

```
#field
// приватное

field
// публичное
```

Синтаксис

Создание классов

```
class MyClass {

  constructor(name, color) {
    this.name = name;
    this.color = color;
  }

  method() {
    console.log(this.name);
  }
}
```

```

    method2() {
      console.log(this.color);
    }
}

// Использование:
let instance = new MyClass('Иван', 'Red');
instance.method();

```

В JavaScript класс – это разновидность функции.

Секция **constructor** идёт первой в описании класса

В нём надо инициализировать объект. Сюда записываются такие свойства создаваемого инстанса, которые являются уникальными лично для него.

Теоретически, в конструктор можно пихнуть не только уникальные для инстанса свойства, но и общие для всех инстансов методы, но это будет неправильно с т.з. семантики. У инстансов должны быть только такие свойства, которые индивидуально характеризуют именно их.

Например, для класса User это может быть имя, фамилия, которые у каждого инстанса свои. А вот методы у них всех общие - посчитать возраст, вывести полное имя и т.д. Общие методы можно один раз положить в прототип и все, они работают одинаково для всех инстансов с разными собственными данными.

Общие методы и свойства для всего класса

Если в тело функции-класса после конструктора записать метод, то он будет передан в прототип.

Если после конструктора записать обычное свойство, то такое свойство будет записано именно в инстанс, а не в прототип.

В примере видно, что свойство отправилось в инстанс, а метод – в прототип.

Этот код работает в браузере и немного иначе работает в node:

```

class User {
  name = "Аноним";

  sayHi() {
    alert(`Привет, ${this.name}!`);
  }
}

const instance = new User();

instance;
// Object { name: "Аноним" }

Object.getPrototypeOf(instance);
// constructor: class User {}
// sayHi: function sayHi()

```

Геттеры, сеттеры в классах

Все методы, в т.ч. геттеры и сеттеры записываются в Class.prototype.

В примере ниже интересный случай, который работает не очевидно: откуда берётся `_name` в коде ниже и куда пропадает `name`, объявленный в конструкторе?

При создании инстанса, `constructor` вызывает сеттер `this.name`. Т.е. `this.name` – это не запись переменной в свойство `name`, а вызов функции-сеттера, куда передаётся значение, которое сеттер записывает в новое свойство `_name`. Соответственно, сеттер `name` создаёт свойство `_name`. А свойства `name` не существует, это – метод-сеттер.

```

class MyClass {

  constructor(name) {
    // вызывает сеттер instance.name =

```

```

    this.name = name;
}

get name() {
    return this._name;
}

set name(value) {
    if (value.length < 4) {
        console.log("Имя слишком короткое.");
        return;
    }
    this._name = value;
}

let instance = new MyClass("Иван");

console.log(instance.name);
// Иван

instance.name = '1234';
console.log(instance.name);
// 1234

instance = new MyClass("");
// Имя слишком короткое.

```

В том, что геттер и сеттер name лежит в прототипе, можно убедиться с помощью этих команд:

```

console.log( Object.getOwnPropertyDescriptors(instance) );
{
    _name: {
        value: 'Иван',
        writable: true,
        enumerable: true,
        configurable: true
    }
}

console.log( Object.getOwnPropertyDescriptors(instance.__proto__) );
{
    constructor: {
        value: [Function: MyClass],
        writable: true,
        enumerable: false,
        configurable: true
    },
    name: {
        get: [Function: get name],
        set: [Function: set name],
        enumerable: false,
        configurable: true
    }
}

```

Это равносильно тому, если бы геттеры и сеттеры были записаны в имеющийся прототип через дескрипторы свойств:

```
Object.defineProperties(User.prototype, {
```

```
name: {
  get() {
    return this._name
  },
  set(name) {
    // ...
  }
});
});
```

Class Expression

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д.
Пример Class Expression (по аналогии с Function Expression):

```
let User = class {
  sayHi() {
    console.log("Привет");
  }
};

const a = new User();
console.log(a) // User {}
a.sayHi() // Привет
```

Class Expression может иметь имя, как и Named Function Expression.

Если у Class Expression есть имя, то оно видно только внутри класса и не видно снаружи:

```
let User = class MyClass {
  sayHi() {
    console.log(MyClass); // имя MyClass видно только внутри класса
  }
};

new User().sayHi();
// работает: [Function: MyClass]

console.log(MyClass);
// ошибка, имя MyClass не видно за пределами класса
```

Можно создавать класс динамически:

Тут речь идёт о том, что класс можно возвращать из функции.

```
function makeClass(phrase) {
  // объявляем класс и возвращаем его
  return class {
    sayHi() {
      console.log(phrase);
    };
  };
}

// Создаём новый класс
let User = makeClass("Привет");

new User().sayHi();
// Привет
```

Наследование классов

extends и super

Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {...}.

```
class Child extends Parent {...}
```

Ключевое слово extends работает, используя прототипы. Оно устанавливает Child.prototype.[[Prototype]] в Parent.prototype. Если метод не найден в Child.prototype, JavaScript берёт его из Parent.prototype.

После extends разрешены любые выражения.

Синтаксис создания класса допускает указывать после extends не только класс, но любое выражение.

В примере ниже обычная функция создаёт родительский класс, к которому привязывается дочерний.

Это может быть полезно для продвинутых приёмов проектирования, где мы можем использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

Здесь class User наследует от результата вызова f("Привет"):

```
function f(phrase) {
  return class {
    sayHi() { alert(phrase) }
  }
}

class User extends f("Привет") {}

new User().sayHi(); // Привет
```

У дочернего класса может не быть своего конструктора.

Согласно [спецификации](#), если класс расширяет другой класс и не имеет своего конструктора, то автоматически создаётся «пустой» конструктор, который использует конструктор родительского класса.

Сбор аргументов и вызов родительского конструктора условно выглядит так:

```
class Rabbit extends Animal {
  constructor(...args) {
    super(...args);
  }
}
```

Первый пример – самый примитивный. Здесь нет дочернего конструктора и нет переопределения методов. Так как у наследника нет собственного конструктора, будет автоматом использоваться родительский конструктор. В результате дочернему классу будет доступно всё из родительского:

```
class MyClass {
  constructor(name) {
    this.name = name;
  }
  m1() {
    console.log('Метод родителя');
  }
}

class Child extends MyClass {
  m2() {
    console.log('Метод наследника');
  }
}
```

```
const instance = new Child('Valakas');
instance.name; // Valakas
instance.m1(); // Метод родителя
instance.m2(); // Метод наследника
```

Super

Super – это ссылка на родителя, как this – ссылка на себя. Ключевое слово super используется для вызова родительского конструктора и родительских методов в дочернем классе. Стрелочные функции не имеют super. Документация [здесь](#).

```
super([arguments]);
// вызов родительского конструктора

super.method([arguments]);
// вызов родительского метода
```

Переопределение конструктора

Если у потомка есть свой конструктор, то конструктор обязан вызывать super(...) первой строчкой:

```
Родительский
constructor(param1, param2) {
  this.param1 = param1;
  this.param2 = param2;
}

Дочерний
constructor(param1, param2, param3) {
  super(param1, param2);
  this.param3 = param3;
}
```

Это связано с тем, что когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его this. Но если запускается конструктор унаследованного класса, он этого не делает. Пустой объект создаёт конструктор родительского класса, а дочерний докидывает туда что-то. Если не вызвать родительский конструктор первой строкой, то объект для this не будет создан, и мы получим ошибку.

В наследуемом классе собственная функция-конструктор помечена специальным внутренним свойством [[ConstructorKind]]: "derived".

Если в родительский конструктор ничего не надо класть (для него не предусмотрено параметров), то всё равно надо его прописать вот так: super().

Переопределение методов

В дочернем классе можно вызывать родительские методы через super.method. Сам по себе родительский метод может использовать инстанс без всяких super, но бывают ситуации, когда технически нужно вызвать родительский метод внутри дочерней функции-конструктора. Например, если используется одноимённый метод, который делает то же самое, что и родительский и что-то дополнительно ещё. Мы делаем что-то в нашем методе и вызываем родительский метод в процессе.

```
class Parent {
  method() {
    return `стоит`;
  }
}

class Child extends Parent {
  method() {
    return super.method() + ` и прячется`;
  }
};
```

Пример работы super

Тут используется и заинствование конструктора, и заинствование метода:

```
class Parent {
  constructor(name) {
    this.name = name;
  }

  method() {
    return `${this.name} стоит`;
  }
}

class Child extends Parent {
  constructor(name, surname) {
    super(name);
    this.surname = surname;
  }

  method() {
    return super.method() + ` и ${this.surname} прячется`;
  }
};

const instance = new Child('Валера', 'Жмых');
console.log( instance.method() ); // Валера стоит и Жмых прячется
```

У стрелочных функций нет super

стрелочные функции не имеют super.

При обращении к super стрелочной функции он берётся из внешней функции.

В примере super в стрелочной функции тот же самый, что и в stop(), поэтому метод отрабатывает как и ожидается.

Если бы мы указали здесь «обычную» функцию, была бы ошибка.

```
class Rabbit extends Animal {
  stop() {
    // вызывает родительский stop после 1 секунды
    setTimeout( () => super.stop(), 1000);
  }
}

// Unexpected super
setTimeout( function() { super.stop() }, 1000);
```

Устройство super, [[HomeObject]]

Почему this не заменит super

Чтобы использовать родительские методы, использовать .this не получится: .this указывает на сам объект, и если попытаться подняться вверх по цепочке прототипов выше, чем на 1 уровень вверх, то попадёшь в бесконечную рекурсию.

Первый наследник работает, второй попадает в рекурсию:

```
let parent = {
  name: 'parent',
  method() {
    console.log(this.name);
```

```

};

let child1 = {
  __proto__: parent,
  name: 'child',
  method() {
    this.__proto__.method.call(this); // (1)
  }
};

let child2 = {
  __proto__: child1,
  name: 'child2',
  method() {
    this.__proto__.method.call(this); // (2)
  }
};

child1.method();
// child

child2.method();
// RangeError: Maximum call stack size exceeded

```

Метод вызывается в контексте текущего объекта (поэтому присутствует .call). Простой вызов this.__proto__.method() будет выполнять родительский метод в контексте прототипа, а не текущего объекта.

Получается, что в обеих строках (1) и (2) значение this – это текущий объект. Таким образом, метод снова и снова вызывает себя, не поднимаясь по цепочке вызовов.

1. Внутри child2 строка (2) вызывает child1.method со значением this = child2
2. В строке (1) в child1.method мы хотим передать вызов выше по цепочке, но this = child2, поэтому this.__proto__.method снова равен child1.eat.
3. child1.method вызывает себя в бесконечном цикле, потому что не может подняться дальше по цепочке.

Интересно, что так получается только в том случае, если методы называются одинаково. Если бы они назывались method1, method2 и method3, то такого бы не произошло, потому что поиск нужного метода сразу бы лез наверх, а одноимённый запускается на второй ступени в контексте объекта this.

[[HomeObject]]

Когда функция объявлена как метод внутри класса или объекта, её свойство [[HomeObject]] становится равно этому объекту. Затем super использует его, чтобы получить прототип родителя и его методы.

Каждый метод знает свой [[HomeObject]] и получает метод родителя из его прототипа.

Методы запоминают свой объект во внутреннем свойстве [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.

```

let parent = {
  name: 'parent',
  method() { // [[HomeObject]] == parent
    console.log(this.name);
  }
};

let child1 = {
  __proto__: parent,
  name: 'child1',
}

```

```

method() { // [[HomeObject]] == child1
  super.method(); // (1)
}
};

let child2 = {
  __proto__: child1,
  name: 'child2',
  method() { // [[HomeObject]] == child2
    super.method(); // (2)
  }
};

child1.method(); // child1
child2.method(); // child2

```

Методы не «свободны»

Функции в JavaScript не привязаны к объектам: их можно копировать между объектами и вызывать с любым this. Существование [[HomeObject]] нарушает этот принцип, так как методы запоминают свои объекты. [[HomeObject]] нельзя изменить, эта связь – навсегда.

Единственное место в языке, где используется [[HomeObject]] – это super. Поэтому если метод не использует super, то мы все ещё можем считать его свободным и копировать между объектами. А вот если super в коде есть, то возможны побочные эффекты.

Вот пример неверного результата super после копирования.

В примере ниже animal – это прототип для rabbit, а plant – прототип для tree. Tree хочет вызвать метод sayHi, который лежит в rabbit, но этот метод реализован через super, поэтому tree в конечном итоге переходит в animal:

```

let animal = {
  sayHi() {
    console.log("Я животное");
  }
};

```

```

// rabbit наследует от animal
let rabbit = {
  __proto__: animal,
  sayHi() {
    super.sayHi();
  }
};

```

```

let plant = {
  sayHi() {
    console.log("Я растение");
  }
};

```

```

// tree наследует от plant
let tree = {
  __proto__: plant,
  sayHi: rabbit.sayHi // (*)
};

```

```

tree.sayHi(); // Я животное

```

В строке (*), метод tree.sayHi скопирован из rabbit. Возможно, мы хотели избежать дублирования кода?

Его [[HomeObject]] – это rabbit, ведь он был создан в rabbit. Свойство [[HomeObject]] никогда не меняется. В коде tree.sayHi() есть вызов super.sayHi(). Он идёт вверх от rabbit и берёт метод из animal.

Методы, а не свойства-функции

Свойство [[HomeObject]] определено именно для методов. Но методы должны быть объявлены именно как method(), а не «method: function()».

Для JavaScript это две большие разницы.

В примере ниже используется синтаксис свойства-функции. У него нет [[HomeObject]] и наследование не работает:

```
let parent = {
  foo: function() { // намеренно пишем так, а не foo() { ...
    console.log('foo');
  }
};

let child = {
  __proto__: parent,
  foo: function() {
    super.foo();
  }
};

child.foo();
// SyntaxError: 'super' keyword unexpected here
// потому что нет [[HomeObject]])
```

Методы запоминают свой объект во внутреннем свойстве [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.

Статические свойства и методы

Статические свойства – это общие свойства для всего класса. Они хранятся в конструкторе класса и доступны через него же. К ним нельзя обратиться через инстанс, только через класс.

В классе такие свойства и методы обозначаются ключевым словом static.

Статическими их можно назвать только в контексте того, что если использовать функции как конструкторы, то в создаваемых объектах это свойство будет отсутствовать, и **доступ к ним будет осуществляться только по имени класса.**

Т.к. конструкторы являются функциями, а функции – объектами, то в конструкторы можно добавлять свойства. Кроме свойств экземпляра и свойств прототипа (в конструктор записываются свойства конкретного инстанса!), JavaScript позволяет определить общие свойства класса в функции-конструкторе.

Статические свойства и методы наследуются через extends. Так происходит потому, что у функций есть свои прототипы и они наследуют друг от друга. Исключение – приватные свойства, о них ниже соответствующий раздел.

Если я правильно понял, в статические свойства записываются:

- универсальные свойства всего класса, существующие в одном экземпляре: Number.MAX_VALUE, Math.PI.
- универсальные методы для работы с экземплярами класса: String.parse(), Object.keys(obj).
- методы, легко создающие пустые шаблонные инстансы класса.

Статические методы

Документация про статические методы:

Ключевое слово [static](#), определяет статический метод для класса. Статические методы вызываются без инстанцирования их класса и не могут быть вызваны у экземпляров (*instance*) класса. Статические методы, часто используются для создания служебных функций для приложения.

Статичный метод – это метод, который не использует собственные свойства объекта. Например, если метод печатает одну и ту же строку, а не подставляет какое-нибудь свойство этого объекта.

Статические методы наследуются. Исключение – встроенные классы. Встроенные классы не наследуют статические методы друг друга.

Статические свойства

Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. В node не работают. Статические свойства выглядят как обычные свойства класса, но со static в начале.

```
class MyClass {  
  
    static method() {  
        console.log('this is static method');  
        return;  
    }  
  
    static name = 'Valakas';  
}  
  
MyClass.method();           // this is static method  
console.log(MyClass.name); // Valakas  
  
// Это фактически то же самое, что присвоить метод снаружи через точку  
// MyClass.staticMethod = function() {...};  
// MyClass.name = 'Valakas'
```

.this и статический метод

Если в статическом методе используется .this, то он **равен самой функции-конструктору**, а не экземпляру (правило «объект до точки»):

```
class MyClass {  
    static method() {  
        console.log(this === MyClass);  
    }  
}  
  
MyClass.method();  
// true
```

Наследование статических свойств и методов

Статические свойства и методы наследуются только при использовании extends.

Например, у Animal есть статический метод Animal.compare, а у Rabbit его нет, но он ему всё равно доступен, как если бы он у него был.

```
class Animal {  
    constructor(name, speed) {  
        this.speed = speed;  
        this.name = name;  
    }  
}
```

```

    run(speed = 0) {
      this.speed += speed; // нахуя?
      console.log(`#${this.name} бежит со скоростью ${this.speed}`);
    }

    static compare(animalA, animalB) {
      return animalA.speed - animalB.speed;
    }
}

class Rabbit extends Animal {
  hide() {
    console.log(`#${this.name} прячется!`);
  }
}

let instances = [
  new Rabbit('Белый', 10),
  new Rabbit('Серый', 9)
];

instances.sort(Rabbit.compare);
instances[0].run();
// Серый бежит со скоростью 9

```

Это работает с использованием прототипов: функция Rabbit прототипно наследует от функции Animal. Функции – это объекты, а объекты могут прототипно наследовать друг от друга

И их prototype тоже наследуют:

```

Animal()           Animal.prototype {}
(static methods)   (methods)
|                   |
Rabbit()          Rabbit.prototype {}

```

Пруфы:

```

console.log(Object.getPrototypeOf(Rabbit) === Animal);
// true

console.log(Object.getOwnPropertyNames(Animal))
// [ 'length', 'prototype', 'compare', 'name' ]

// для обычных методов
console.log(Rabbit.prototype.__proto__ === Animal.prototype);
// true

```

То, что написано дальше – это [задача](#) после темы. Но в ней ещё немного теории и примеров, поэтому я её помешу сюда. Важно не забывать, что всё, что здесь написано, касается статических методов, а не просто методов объектов.

Примеры использования

Класс Article может создавать объекты-статьи, и эти объекты нужно сравнивать между собой. Для этого создаётся статический метод Article.compare:

```
class Article {
```

```

constructor(title, date) {
  this.title = title;
  this.date = date;
}

static compare(articleA, articleB) {
  return articleA.date - articleB.date;
}

// использование
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];
articles.sort(Article.compare);

console.log(articles[0].title); // CSS

```

Ещё пример – «фабричный метод».

Допустим, надо создавать статьи различными способами:

1. через заданные параметры и вводные данные – это в конструктор
2. создание пустой статьи с сегодняшней датой – это в статику.

Статический метод в коде ниже – Article.createTodays(). Он использует возврат экземпляра класса из функции.

```

class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static createTodays() {
    return new this('Сегодняшний дайджест', new Date());
    //return new Article('Сегодняшний дайджест', new Date());
  }
}

const instance = Article.createTodays();
console.log(instance.title)
// Сегодняшний дайджест

```

Статические методы также используются в классах, относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных, например:

```

// предположим, что Article – это специальный класс для управления статьями
// статический метод для удаления статьи:
Article.remove({id: 12345});

```

Приватные и защищённые методы и свойства

Внутренний и внешний интерфейсы

Один из важнейших принципов объектно-ориентированного программирования – разделение внутреннего и внешнего интерфейсов. Это обязательная практика в разработке чего-либо сложнее, чем «hello world».

В объектно-ориентированном программировании **свойства и методы разделены на 2 группы**:

Внутренний интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса. Они не доступны для пользователя, но могут использоваться в доступных для пользователя методах в качестве вспомогательных механизмов.

Внешний интерфейс – методы и свойства, доступные снаружи класса.

Внутренний интерфейс. Если провести аналогию с кофеваркой – это то, что скрыто внутри: трубка кипятильника, нагревательный элемент и т.д. Внутренний интерфейс используется для работы объекта, его детали используют друг друга.

Внешний интерфейс – это то, что будет использовать пользователь. Снаружи кофеварка закрыта защитным кожухом, детали скрыты и недоступны. Мы можем использовать их функции через внешний интерфейс.

В JavaScript есть **два типа полей** (свойств и методов) объекта:

Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.

Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Можно сказать, что внутренний интерфейс реализуется через приватные и защищённые поля. Для скрытия внутреннего интерфейса мы используем защищённые или приватные свойства:

Защищённые поля

имеют префикс `_`. Это хорошо известное соглашение, не поддерживаемое на уровне языка. Программисты должны обращаться к полю, начинающемуся с `_`, только из его класса и классов, унаследованных от него.

Приватные поля

имеют префикс `#`. JavaScript гарантирует, что мы можем получить доступ к таким полям только внутри класса. Такие свойства не наследуются. В настоящее время приватные поля не очень хорошо поддерживаются в браузерах, но можно использовать полифил.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрешён доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что мы обычно хотим, чтобы наследующие классы получали доступ к внутренним полям. Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

защищённое свойство

Защищённые свойства начинаются с префикса `_`.

Это не синтаксис языка, это такое соглашение между кодерами, что такие свойства и методы не должны быть доступны извне. Т.к. это всего лишь соглашение, а не специальный синтаксис, такие свойства наследуются.

Допустим, есть класс-кофеварка, в нём есть такие свойства, которые сейчас являются публичными:

```
class CoffeeMachine {  
    waterAmount = 0;  
    constructor(power) {  
        this.power = power;  
    }  
}
```

В примере ниже эти же свойства становятся защищёнными через префикс `_`.

Изменим свойство `waterAmount` на защищённое. Пусть оно не устанавливается ниже нуля.

Здесь есть опять интересная штука с геттерами-сеттерами: если геттер `waterAmount` вернёт `this.waterAmount`, то он будет бесконечно возвращать сам себя, потому что `waterAmaunt` – это геттер. Поэтому запись делается в `_waterAmount`.

```
class CoffeeMachine {  
    _waterAmount = 0;  
  
    setWaterAmount(value) {  
        if (value < 0) throw new Error("Отрицательное количество воды");  
        this._waterAmount = value;  
    }  
}
```

```

getWaterAmount() {
  return this._waterAmount;
}
constructor(power) {
  this._power = power;
}

// создаём новую кофеварку
let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды
coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

```

Один из вариантов организации **защиты от перезаписи** – не создавать сеттер.

Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось.

Так как у кофеварки мощность никогда не меняется, мощность имеет смысл указать только при создании объекта. Для этого нам нужно создать только геттер, но не сеттер (ахуенная защита).

Уровень воды, напротив, всегда меняется, для него есть и сеттер, и геттер:

```

class CoffeeMachine {

  constructor(power) {
    this._power = power;
  }

  get power() {
    return this._power;
  }

  setWaterAmount(value) {
    if (value < 0) throw new Error("Отрицательное количество воды");
    this._waterAmount = value;
  }

  getWaterAmount() {
    return this._waterAmount;
  }
}

new CoffeeMachine().setWaterAmount(100);

```

приватное свойство

Эта возможность была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифил.

Приватные свойства и методы инстансов должны начинаться с **#**. На уровне языка **#** является специальным символом, который означает, что поле приватное. Мы не можем получить к нему доступ извне или из наследуемых классов. К ним можно без проблем обращаться внутри функции-конструктора класса, но снаружи функции к ним уже обратиться нельзя. Для этого нужно написать специальные геттеры и сеттеры. Приватные поля не наследуются.

В отличие от защищённых, функциональность приватных полей обеспечивается самим языком. Но если мы унаследуем что-то, то мы не получим прямого доступа к **#**приватным полям. Мы будем вынуждены полагаться на функции геттер/сеттер, которые создаются без **#** и, соответственно, наследуются.

Во многих случаях такое ограничение слишком жёсткое. Раз уж мы расширяем класс, у нас может быть вполне законная причина для доступа к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Приватные поля не конфликтуют с публичными. У нас может быть #приватное и публичное поля с одинаковым именем:

```
#field  
// приватное  
  
field  
// публичное
```

Квадратные скобки тоже не дают доступ к приватным полям:

```
#value = 0;  
simpleValue = 123;  
  
getValue() {  
    // return this.#value  
    const key = '#value'  
    return this[key];      // undefined  
}
```

Геттеры-сеттеры и приватные поля

Получить доступ к приватному свойству можно только через специально написанные для него геттеры-сеттеры. В примере ниже создаётся класс, в котором есть приватное свойство и обычное.

```
class MyClass {  
    #value = 0;  
    simpleValue = 123;  
  
    getValue() {  
        return this.#value  
    }  
    setValue(num) {  
        this.#value = num;  
    }  
}  
  
const instance = new MyClass();  
  
instance;           // MyClass {simpleValue: 123, #value: 0}  
instance.simpleValue; // 123  
instance.#value;     // Private field '#value' must be declared in an enclosing class  
  
// через геттеры и сеттеры всё работает  
instance.getValue(); // 0  
instance.setValue('new Value');  
instance;           // MyClass {simpleValue: 123, #value: "new Value"}
```

Расширение встроенных классов

От встроенных классов, таких как Array, Map и других, тоже можно наследовать.

Если от них наследовать и применить `.map` `.filter` и другие функции на экземпляре наследующего потомка, они будут возвращать экземпляр класса-потомка. Так происходит потому, что для создания пропущенного через родительскую функцию дочернего экземпляра вновь используется свойство дочернего экземпляра `.constructor`.

Т.е. JS сначала ищет у родителя функцию, прогоняет дочерний экземпляр через неё, а для получившегося результата вызывает `instance.constructor`.

Здесь речь идёт не о статических методах (таких как `Object.keys`, например), а о методах экземпляров. Если наследуется класс, то и прототип со всеми методами тоже наследуется.

В примере ниже видно, что использование родительского `.filter` возвращает экземпляр дочернего класса. Через свойство `.constructor` видно, что оба инстанса созданы дочерним классом.

Нужно вспомнить, что дочернему классу не обязательно иметь свой конструктор.

```
// расширим array и добавим метод
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }
}

const arr = new PowerArray(1, 2, 5, 10, 50);
console.log(arr.isEmpty()); // false

let filtered = arr.filter((item) => item >= 10);
console.log(filtered);      // PowerArray(2) [ 10, 50 ]

console.log(arr.constructor) // [Function: PowerArray]
console.log(filtered.constructor) // [Function: PowerArray]
```

Symbol.species

Этим поведением можно управлять при помощи статического геттера `Symbol.species`.

Symbol.species

Документация [здесь](#).

`Symbol.species` — известный символ, позволяющий определить конструктор, использующийся для создания порождённых объектов. Свойство `Symbol.species`, содержащее аксессор (геттер), позволяет подклассам переопределить конструктор, используемый по умолчанию для создания новых объектов.

Понятно, как это работает с родительскими функциями высшего порядка. А если бы у дочернего класса были свои функции, обрабатывающие и возвращающие новый экземпляр, они бы тоже становились родительскими?

```
class PowerArray extends Array {
  isEmpty() {
    return this.length === 0;
  }

  // перегружаем species для использования родительского конструктора Array
  static get [Symbol.species]() {
    return Array;
  }
}
```

Наследование статических свойств встроенных классов

У встроенных объектов есть собственные статические методы, например `Object.keys`, `Array.isArray` и т. д. Обычно, когда один класс наследует от другого, то наследуются и статические методы.

Иключение – встроенные классы. Встроенные классы **не наследуют** статические методы друг друга.

Например, классы Array и Date наследуют от Object. Их экземплярам доступны методы из Object.prototype. Но Array.[[Prototype]] не ссылается на Object, поэтому нет статических методов Array.keys() или Date.keys().

Проверка класса

В [старой](#) версии статьи есть про утиную типизацию.

Проверить, к какому классу принадлежит экземпляр, можно с помощью:

1. оператора instanceof

Оператор будет искать наименование класса в цепочке прототипов. Если экземпляр наследует от какого-то прототипа, то вернёт true.

Схема поиска будет другая, если используется статический метод static [Symbol.hasInstance](obj).

2. оператора isPrototypeOf

Это как instanceof, только наоборот.

3. метода .toString

Встроенный метод `toString` может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения. Можно использовать `{}.toString.call` вместо `instanceof` для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.

Например, для массивов это будет так:

```
Object.prototype.toString.call(arr) => [object Array]
```

4. Утиная типизация

в ООП-языках – определение факта реализации определённого интерфейса объектом без явного указания или наследования этого интерфейса, а просто по реализации полного набора его методов.

Смысл утиной типизации – в проверке отдельных необходимых методов и свойств.

Например, мы можем проверить, что объект – массив, не вызывая `Array.isArray`, а просто уточнив наличие важного для нас метода, например `splice`:

```
var something = [1, 2, 3];

if (something.splice) {
  console.log('Это массив');
}
```

instanceof

instanceof

Документация [здесь](#).

Оператор проверяет, принадлежит ли объект к определённому классу (конструктору).

Он проверяет, присутствует ли объект «`constructor.prototype`» в цепочке прототипов `object`.

Важно: `instanceof` не учитывает саму функцию-конструктор при проверке, а только `prototype`, который проверяется на совпадения в прототипной цепочке.

Работает как с классами, так и с конструкторами.

Оператор вернёт `true`, если `obj` принадлежит классу **Class или наследующему** от него.

Синтаксис

```
obj instanceof Class
obj instanceof Constructor

obj
```

Проверяемый объект

Class

Класс или функция-конструктор

Пример работы:

```
// с классами
class MyClass {}
const myObject = new MyClass();

myObject instanceof MyClass; // true

// с конструкторами
function MyConstructor() {}
const myObject2 = new MyConstructor();

myObject2 instanceof MyConstructor; // true
```

Алгоритм работы instanceof описан чуть ниже.

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Это работает так: «экземпляр» instanceof «Класс» означает, что JS будет искать «Класс» в цепочке прототипов всё выше и выше, пока не найдёт. Например, [array] instanceof Object => true, потому что класс Array наследует от Object.

Если изменится constructor.prototype, то эта цепочка может нарушиться. Результат оператора instanceof зависит от свойства constructor.prototype (у функции-конструктора), поэтому результат оператора может поменяться после изменения этого свойства. Также результат может поменяться после изменения прототипа object (или цепочки прототипов) с помощью Object.setPrototypeOf или __proto__ (у экземпляра).

```
.__proto__ = Object? Да
|
|
[array] instanceof Object  [].__proto__ = Object? Нет
// true
```

Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance. Если этот статический метод-символ присутствует у конструктора и отрабатывает, то JS не пойдёт выше по цепочке.

```
[array] instanceof Object  Array.[Symbol.hasInstance](obj)? Да
// true
```

Symbol.hasInstance

Symbol.hasInstance

Это символ, который используется для определения является ли объект экземпляром конструктора. О нём подробнее написано в «Типы данных». Используется для изменения поведения оператора [instanceof](#).

Документация [здесь](#).

Алгоритм instanceof

Алгоритм obj instanceof Class работает в таком порядке:

1. Если имеется статический метод Symbol.hasInstance, тогда вызвать его: Class[Symbol.hasInstance](obj). Он должен вернуть либо true, либо false, и это конец. Это как раз и есть возможность ручной настройки instanceof:

```
// проверка instanceof будет полагать, что всё со свойством canEat - животное Animal
class Animal {
  static [Symbol.hasInstance](obj) {
    if (obj.canEat) return true;
  }
}

let obj = { canEat: true };
alert(obj instanceof Animal); // true: вызван Animal[Symbol.hasInstance](obj)
```

2. Большая часть классов не имеет метода Symbol.hasInstance. В этом случае снизу вверх проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj:

```
obj.__proto__ === Class.prototype?
obj.__proto__.__proto__ === Class.prototype?
obj.__proto__.__proto__.__proto__ === Class.prototype?
...
// если какой-то из ответов true - возвратить true
// если дошли до конца цепочки - false
```

В примере ниже, совпадение будет на втором шаге:

```
class Animal {}
class Rabbit extends Animal {}

let rabbit = new Rabbit();
alert(rabbit instanceof Animal); // true

// rabbit.__proto__ === Rabbit.prototype
// rabbit.__proto__.__proto__ === Animal.prototype (совпадение!)
```

Последствия изменения прототипа

Это может приводить к интересным последствиям при изменении свойства prototype после создания объекта. Как, например, тут:

```
function Rabbit() {}

// создаём экземпляр
let rabbit = new Rabbit();

// заменяем прототип
Rabbit.prototype = {};

// ...больше не rabbit!
alert( rabbit instanceof Rabbit ); // false
```

В приложении к уроку задача, которая тоже помогает понять эту тему:

Почему instanceof в примере ниже возвращает true? Мы же видим, что «а» не создан с помощью B().

```
function A() {}
function B() {}

A.prototype = B.prototype = {};

let a = new A();

console.log( a instanceof B ); // true
```

`instanceof` не учитывает саму функцию при проверке, а только `prototype`, который проверяется на совпадения в прототипной цепочке.

И в данном примере `a.__proto__ == B.prototype`, так что `instanceof` возвращает `true`.

Таким образом, по логике `instanceof`, именно `prototype` в действительности определяет тип, а не функция-конструктор.

isPrototypeOf

Есть метод `objA.isPrototypeOf(objB)`, который возвращает `true`, если объект `objA` есть где-то в прототипной цепочке объекта `objB`.

Так что `obj instanceof Class` можно перефразировать как `Class.prototype.isPrototypeOf(obj)`.

Забавно, но сам конструктор `Class` не участвует в процессе проверки! Важна только цепочка прототипов `Class.prototype`.

.toString возвращает тип

Обычные объекты преобразуются к строке как `[object Object]`.

Так работает реализация метода `toString`. Но у `toString` имеются скрытые возможности, которые делают метод гораздо более мощным. Мы можем использовать его как расширенную версию `typeof` и как альтернативу `instanceof`.

Согласно [спецификации](#), встроенный метод `toString` может быть позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

- Для числа это будет `[object Number]`
- Для булева типа это будет `[object Boolean]`
- Для `null`: `[object Null]`
- Для `undefined`: `[object Undefined]`
- Для массивов: `[object Array]`
- ...и т.д. (поведение настраивается).

продемонстрируем:

```
// скопируем метод toString в переменную для удобства
let objectToString = Object.prototype.toString;

// какой это тип?
let arr = [];

console.log( objectToString.call(arr) ); // [object Array]
```

В примере мы использовали `call`, как описано в главе [Декораторы и переадресация вызова, call/apply](#), чтобы выполнить функцию `objectToString` в контексте `this=arr`.

Внутри, алгоритм метода `toString` анализирует контекст вызова `this` и возвращает соответствующий результат.

Больше примеров:

```
let s = Object.prototype.toString;

console.log( s.call(123) );           // [object Number]
console.log( s.call(null) );          // [object Null]
console.log( s.call(console.log) );    // [object Function]
```

Symbol.toStringTag

Поведение метода объектов `toString` можно настраивать, используя специальное свойство объекта `Symbol.toStringTag`. О нём подробнее написано в «Типы данных».

Например:

```
let user = {
  [Symbol.toStringTag]: "User"
};

console.log( {}.toString.call(user) ); // [object User]
```

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

```
// toStringTag для браузерного объекта и класса
console.log( window[Symbol.toStringTag] );           // window
console.log( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

console.log( {}.toString.call(window) );             // [object Window]
console.log( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]
```

Результат преобразования к строке – это значение свойства `obj.Symbol.toStringTag` (если он имеется), которое находится в `[object ...]`.

В итоге мы получили «`typeof` на стероидах», который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать `{}.toString.call` вместо `instanceof` для встроенных объектов, когда мы хотим получить тип в виде строки, а не просто сделать проверку.

Как мы можем видеть, технически `{}.toString` «более продвинут», чем `typeof`.

А оператор `instanceof` – отличный выбор, когда мы работаем с иерархией классов и хотим делать проверки с учётом наследования.

Утиная типизация

Это раздел из старой [страницы](#).

Альтернативный подход к типу – «утиная типизация», которая основана на одной известной пословице: «*If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck (who cares what it really is)*».

В переводе: «Если это выглядит как утка, плавает как утка и крякает как утка, то, вероятно, это утка (какая разница, что это на самом деле)».

Утиная типизация (англ. Duck typing) в ООП-языках – определение факта реализации определённого интерфейса объектом без явного указания или наследования этого интерфейса, а просто по реализации полного набора его методов.

Смысл утиной типизации – в проверке отдельных необходимых методов и свойств.

Например, мы можем проверить, что объект – массив, не вызывая `Array.isArray`, а просто уточнив наличие важного для нас метода, например `splice`

```
var something = [1, 2, 3];

if (something.splice) {
  console.log('Это утка! То есть, массив!');
}
```

в `if` мы не вызываем метод `something.splice()`, а пробуем получить само свойство `something.splice`. Для массивов оно всегда есть и является функцией, т.е. даст в логическом контексте `true`.

Аналогично, проверить на дату можно, определив наличие метода `getTime`:

```
var x = new Date();

if (x.getTime) {
  alert('Дата!');
  alert(x.getTime()); // работаем с датой
}
```

С виду такая проверка хрупка, её можно «сломать», передав похожий объект с тем же методом.

Но как раз в этом и есть смысл утиной типизации: если объект похож на дату, у него есть методы даты, то будем работать с ним как с датой (какая разница, что это на самом деле).

То есть мы намеренно позволяем передать в код нечто менее конкретное, чем определённый тип, чтобы сделать его более универсальным.

Проверка интерфейса

Если говорить словами «классического программирования», то «duck typing» – это проверка реализации объектом требуемого интерфейса. Если реализует – ок, используем его. Если нет – значит это что-то другое.

Примеси

В JavaScript можно наследовать только от одного объекта. Объект имеет единственный `[[Prototype]]`. И класс может расширять только один другой класс.

Иногда это может ограничивать нас. Например, у нас есть класс `StreetSweeper` и класс `Bicycle`, а мы хотим создать их смесь. Для таких случаев существуют «примеси».

Примесь – это класс, методы которого предназначены для использования в других классах, но без наследования от примеси. Примесь только содержит в себе методы, которые реализуют определённое поведение. Метод, хранящиеся в примеси, используются другими объектами, чтобы добавить функциональность другим классам.

Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть добавлены в прототип любого класса. Это не наследование, а просто копирование методов.

В примере ниже создаётся самостоятельный объект, в котором лежат свойства, использующие this. Эти свойства копируются в прототип класса через Object.assign(куда?, откуда?). Так как методы из примеси залиты в прототип класса, экземпляры класса могут их использовать.

Таким образом, класс User может наследовать от другого класса, но при этом также включать в себя примеси, «подмешивающие» другие методы:

```
// объект-примесь
let sayHiMixin = {

  sayHi() {
    console.log(`Привет, ${this.name}`);
  },

  sayBye() {
    console.log(`Пока, ${this.name}`);
  }
};

// создание класса
class User {
  constructor(name) {
    this.name = name;
  }
}

// копирование в прототип класса
Object.assign(User.prototype, sayHiMixin);

new User('Вася').sayHi();
new User('Вася').sayBye();
```

Тут используется метод Object.assign()

Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.

```
Object.assign(target, ...sources)

var object = { key1: 'value1', key2: 'value2' };
var copy = Object.assign({}, object);
```

Сами примеси могут наследовать друг друга. В примере ниже sayHiMixin наследует от sayMixin:

```
let sayMixin = {
  say(phrase) {
    console.log(phrase);
  }
};

let sayHiMixin = {
  __proto__: sayMixin,
  // для задания прототипа можно также использовать Object.create
```

```

sayHi() {
  // вызвать метод родителя
  super.say(`Привет, ${this.name}`);
},
sayBye() {
  super.say(`Пока, ${this.name}`);
}
};

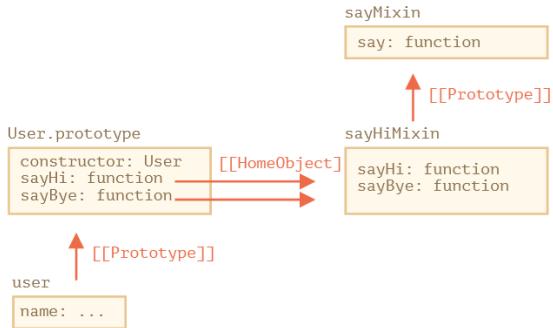
class User {
  constructor(name) {
    this.name = name;
  }
}

Object.assign(User.prototype, sayHiMixin);

new User('Вася').sayHi()
// Привет, Вася

```

При вызове родительского метода `super.say()` из `sayHiMixin` этот метод ищется в прототипе самой примеси, а не класса:



Это связано с тем, что методы `sayHi` и `sayBye` были изначально созданы в объекте `sayHiMixin`. Несмотря на то, что они скопированы в прототип класса `User`, их внутреннее свойство `[[[HomeObject]]]` ссылается на `sayHiMixin`, как показано на картинке выше.

Так как `super` ищет родительские методы в `[[[HomeObject]].[[Prototype]]]`, это означает `sayHiMixin.[[Prototype]]`, а не `User.[[Prototype]]`.

для задания прототипа можно также использовать:

[Object.create\(\)](#)

Создаёт пустой объект со свойством `[[[Prototype]]]`, указанным как `proto`, и необязательными дескрипторами свойств `descriptors`.

```
Object.create(proto[, propertiesObject])
```

`proto`

Объект, который станет прототипом вновь созданного объекта

`propertiesObject`

Необязательный. дескрипторы свойств.

??? EventMixin

Вообще ничего не понял.

Многие браузерные объекты могут генерировать события. События – отличный способ передачи информации всем, кто в ней заинтересован.

Давайте создадим примесь, которая позволит легко добавлять функциональность по работе с событиями любым классам/объектам.

Методы в примеси:

.trigger(name, [data])

Для генерации события. Аргумент name – это имя события, за которым могут следовать другие аргументы с данными для события.

.on(name, handler)

Назначает обработчик для события с заданным именем. Обработчик будет вызван, когда произойдёт событие с указанным именем name, и получит данные из .trigger.

.off(name, handler)

Удаляет обработчик указанного события.

После того, как все методы примеси будут добавлены, объект user сможет сгенерировать событие "login" после входа пользователя в личный кабинет. Другой объект – calendar, сможет использовать это событие, чтобы показывать зашедшему пользователю актуальный для него календарь.

Или menu может генерировать событие "select", когда элемент меню выбран, а другие объекты могут назначать обработчики, чтобы реагировать на это событие, и т.п.

Код примеси:

```
let eventMixin = {
  /**
   * Подписаться на событие, использование:
   * menu.on('select', function(item) { ... })
   */
  on(eventName, handler) {
    if (!this._eventHandlers) this._eventHandlers = {};
    if (!this._eventHandlers[eventName]) {
      this._eventHandlers[eventName] = [];
    }
    this._eventHandlers[eventName].push(handler);
  },
  /**
   * Отменить подписку, использование:
   * menu.off('select', handler)
   */
  off(eventName, handler) {
    let handlers = this._eventHandlers && this._eventHandlers[eventName];
    if (!handlers) return;
    for (let i = 0; i < handlers.length; i++) {
      if (handlers[i] === handler) {
        handlers.splice(i--, 1);
      }
    }
  },
  /**
   * Сгенерировать событие с указанным именем и данными
   * this.trigger('select', data1, data2);
   */
  trigger(eventName, ...args) {
    if (!this._eventHandlers || !this._eventHandlers[eventName]) {
      return; // обработчиков для этого события нет
    }

    // вызовем обработчики
    this._eventHandlers[eventName].forEach(handler => handler.apply(this, args));
  }
}
```

```
};
```

Итак, у нас есть 3 метода:

.on(eventName, handler)

назначает функцию `handler`, чтобы обработать событие с заданным именем. Обработчики хранятся в свойстве `_eventHandlers`, представляющим собой объект, в котором имя события является ключом, а массив обработчиков – значением.

.off(eventName, handler)

убирает функцию из списка обработчиков.

.trigger(eventName, ...args)

генерирует событие: все назначенные обработчики из `_eventHandlers[eventName]` вызываются, и `...args` передаются им в качестве аргументов.

Использование:

```
// Создадим класс
class Menu {
  choose(value) {
    this.trigger("select", value);
  }
}
// Добавим примесь с методами для событий
Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// Добавить обработчик, который будет вызван при событии "select":
menu.on("select", value => alert(`Выбранное значение: ${value}`));

// Генерирует событие => обработчик выше запускается и выводит:
menu.choose("123"); // Выбранное значение: 123
```

Теперь если у нас есть код, заинтересованный в событии "select", то он может слушать его с помощью `menu.on(...)`.

А `eventMixin` позволяет легко добавить такое поведение в любой класс без вмешательства в цепочку наследования.

Из комментов:

Разобрал [пример с событиями в песочнице](#) с подробными комментариями... Разбирался по ходу кода, поэтому сначала комментарии краткие, по действиям, а потом идут обобщения. Ну и пару примеров от себя, для закрепления.

Когда читал уже авторские примечания, уже после того, как полностью разобрался, и то, сразу понять что же делает эта хреновина и для чего создаётся сразу не совсем очевидно. То есть, на мой взгляд, тут надо либо СНАЧАЛА разбираться полностью в коде и уже потом говорить о возможных вариантах применения этому (а автор сразу говорит о применении). Либо УЖЕ иметь в голове задачи, о которых говорит автор, и следовать ходу его рассуждений чётко понимая какие проблемы подобный код призван решать. Без этого, по примечаниям, воссоздать что же именно делает код - по-моему, ну очень не интуитивно... Кстати, общая логика, тоже не такая уж сложная, когда понимаешь, что именно призваны реализовать те или иные функции...

Итого

Примесь – общий термин в объектно-ориентированном программировании: класс, который содержит в себе методы для других классов.

Некоторые другие языки допускают множественное наследование. JavaScript не поддерживает множественное наследование, но с помощью примесей мы можем реализовать нечто похожее, скопировав методы в прототип.

Мы можем использовать примеси для расширения функциональности классов, например, для обработки событий, как мы сделали это выше.

С примесями могут возникнуть конфликты, если они перезаписывают существующие методы класса. Стоит помнить об этом и быть внимательнее при выборе имён для методов примеси, чтобы их избежать.

Полиморфизм, инкапсуляция

В ИТ широко распространён термин "Интерфейс", который по смыслу похож на то, как мы используем это слово в повседневной жизни. Например, пользовательский интерфейс представляет собой совокупность элементов управления сайтом, телефоном и так далее. Интерфейсом пульта управления от телевизора являются кнопки. Резюмируя, можно сказать, что интерфейс определяет способ взаимодействия с системой.

В программировании всё устроено похожим образом. **Интерфейсом** называют набор функций (имена и их сигнатуры, то есть количество и типы входящих параметров, а также возвращаемое значение), не зависящих от конкретной реализации.

Как соотносятся между собой понятия **абстракция** и интерфейс? Абстракция – это слово, описывающее в первую очередь те данные, с которыми мы работаем. Например, почти каждое веб-приложение включает в себя абстракцию "пользователь". Интерфейсом же называется набор функций, с помощью которых можно взаимодействовать с данными.

Но функции бывают не только интерфейсные, но и вспомогательные, которые не предназначены для вызывающего кода и используются исключительно внутри абстракции (эти функции как раз хорошо рассмотрены на learn.js)

В сложных абстракциях (которые могут быть представлены внешними библиотеками), количество неинтерфейсных функций значительно больше, чем интерфейсных. Вплоть до того, что интерфейсом библиотеки могут являться одна или две функции, но в самой библиотеке их сотни. То, насколько хороша ваша абстракция, определяется в том числе тем, насколько удобен её интерфейс.

Интерфейсные функции – функции, которые использует пользователь.

Внутренние (вспомогательные) функции – функции, которые используются исключительно внутри абстракции и к которым у пользователя нет доступа.

абстрактный тип данных – математическая модель для типов данных, определяемая в терминах возможных операций над этими данными и их значений.

В программировании абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов. Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться. Такой подход соответствует принципу [инкапсуляции](#) в объектно-ориентированном программировании. Сильной стороной этой методики является именно сокрытие реализации. Раз вовне опубликован только интерфейс, то пока структура данных поддерживает этот интерфейс, все программы, работающие с заданной структурой абстрактным типом данных, будут продолжать работать. Разработчики структур данных стараются, не меняя внешнего интерфейса и [семантики](#) функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надежности и используемой памяти.

Инкапсуляция

+ хекслет

Инкапсуляция – объединение и скрытие от прямого обращения данных (в т.ч. функций) в рамках одной структуры. Функции называют «методами», а данные – «свойствами».

В терминах ООП отделение внутреннего интерфейса от внешнего называется [инкапсуляция](#). Это даёт следующие выгоды:

- Защита от дурака. Если пользователь класса изменит вещи, не предназначенные для изменения извне – последствия непредсказуемы.
- Поддерживаемость. Код постоянно подвергается разработке и улучшению. Если мы чётко отделим внутренний интерфейс, то разработчик сразу увидит приватные методы, беззапосно их изменит, переименует, потому что от них не зависит никакой внешний код. В новой версии вы можете полностью всё переписать, но пользователю будет легко обновиться, если внешний интерфейс остался такой же.
- Сокрытие сложности. Всегда удобно, когда детали реализации скрыты, и доступен простой, хорошо документированный внешний интерфейс.

Полиморфизм

+ Hexlet

Полиморфизм – это способность функции обрабатывать данные разных типов. На практике позволяет избежать дублирования кода.

Виды полиморфизма:

1. Специальный (ad hoc)
2. Параметрический
3. Полиморфизм подтипов (включения)

Параметрический полиморфизм

Это вызов одного и того же кода для всех допустимых подтипов (полиморфных) аргументов.

```
const numbers1 = l(1, 2, 3);
const numbers2 = l(4, 5, 6);
append(numbers1, numbers2);
// (1, 2, 3, 4, 5, 6)

const strings1 = l('cat', 'dog');
const strings2 = l('table', 'milk');
append(strings1, strings2);
// ('cat', 'dog', 'table', 'milk')
```

Операция append не зависит от того, что находится внутри списков. В одном случае список собирается из двух списков чисел, а во втором – из двух списков строк.

JS – динамический язык (динамическая типизация), поэтому он в любом случае позволяет так делать. В статических языках для каждого типа данных существует своя объединяющая функция. Для объединения списков из разных типов данных также существуют отдельные функции.

Ad hoc (специальный)

В зависимости от типа аргументов применяется разная реализация какой-то операции.

Если складываемые типы – числа, то будет арифметическое сложение.

Если строки – конкатенация.

```
1 + 1; // 2
'cat' + 'dog'; // catdog
```

В статических языках переписываются тела функций для разных типов аргументов, и компилятор сам выбирает, какую функцию в данном момент использовать. Внутри нет никаких if, внутри функции скрывается несколько разных тел.

В динамических языках это сводится к тому, что внутри функции есть много if-ов. Например, console.log устроен так, что внутри есть функция format, которая проверяет тип аргумента и в зависимости от типа делает разные действия.

```
console.log(1034.98);
console.log('hello');
```

Полиморфизм включения (подтипов)

Его в ООП как раз и называют полиморфизмом, он является ключевой штукой. Он позволяет вызывать разный код для разных иерархий типов. Подкурс хекслета про карточную игру как раз про него.

Для разных типов вызывается разная реализация одних и тех же функций с точки зрения интерфейса (чтобы не писать if-ы).

Динамическая диспетчеризация – один из способов реализации динамического полиморфизма.

```
const obj = new SimpleCard(); // or PercentCard()
obj.damage(health);
```

Пример полиморфной функции

Это пример со старой [страницы](#) на learn.js.

Полиморфизм – это когда одна функция может работать с разными типами данных. Как реализовать полиморфизм по-моему в хекслете хорошо написано, тут лишь пара слов и наглядный пример.

Пример полиморфной функции – sayHi(who), которая будет говорить «Привет» своему аргументу, причём если передан массив – то «Привет» каждому:

```
function sayHi(who) {
  if (Array.isArray(who)) {
    who.forEach(sayHi);
  } else {
    console.log('Привет, ' + who);
  }
}

// Вызов с примитивным аргументом
sayHi("Вася"); // Привет, Вася

// Вызов с массивом
sayHi(["Саша", "Петя"]); // Привет, Саша Петя

// и даже рекурсивно
sayHi(["Саша", "Петя", ["Маша", "Юля"]]);
// Привет Саша Петя Маша Юля
```

Проверку на массив в этом примере можно заменить на «утиную» – нам ведь нужен только метод forEach:

```
function sayHi(who) {
  if (who.forEach) {
    who.forEach(sayHi);
  } else {
    console.log('Привет, ' + who);
  }
}
```

JS – Обработка ошибок

Ошибки кратко

Обычно в случае ошибки скрипт «падает», т.е. программа останавливается, вылетает с выводом ошибки в консоль. Чтобы этого не происходило, можно эти ошибки перехватывать, обрабатывать и продолжать работу.

Есть одно требования: ошибки должны возникать в синтаксически правильном коде. Если забыли поставить закрывающую скобку – это синтаксически неправильный код, такая ошибка не обрабатывается, потому что JS просто не знает, как такой код выполнить.

Если делается ссылка на несуществующую переменную – такую ошибку можно обработать.

Исключения – это ошибки, которые возникают в синтаксически корректном коде. Их также называют «ошибками во время выполнения». Есть ещё близкое понятие `throw` (возбуждение исключения), о нём написано ниже.

Блоки инструкций

Мануал [здесь](#).

Конструкция `try...catch` состоит из 3-х блоков инструкций:

1. `try {...}`

В нём исполняется основной код. Здесь «должны» происходить ошибки.

2. `catch(err) {...}`

Этот блок будет вызван только в том случае, если в `try` произошла ошибка. Переменная `err` – это объект ошибки, который автоматом передаётся сюда как аргумент. Можно использовать любое имя для объекта. Подробнее об объекте ошибки будет ниже.

Если в блоке `try` нет ошибок, то блок `catch` игнорируется. Если ошибка есть, то выполнение `try` прерывается, и поток управления переходит в начало `catch(err)`. Таким образом, при ошибке в блоке `try {...}` мы получаем возможность обработать ошибку внутри `catch`.

3. `finally {...}`

Выполняется всегда, вне зависимости от того, была ошибка или нет.

Если ошибки не было, то он выполнится после `try`, если была – выполнится после блока `catch`.

Секцию `finally` часто используют, когда мы начали что-то делать и хотим завершить это вне зависимости от наличия или отсутствия ошибки. Например, для завершения измерений.

См. также «`finally`, особенности»

На практике это выглядит так:

```
try {  
    // код  
} catch (err) {  
    // обработка ошибки  
} finally {  
    // выполняется всегда  
}
```

Всё, что объявлено в `try`, находится в локальной зоне видимости. Если тут объявить переменные, их не будет видно снаружи.

Блоки `catch` или `finally` опциональны. Можно использовать только конструкции `try..catch` и `try..finally`.

Конструкция `try..catch` работает синхронно. Это означает, что исключение, которое произойдёт в асинхронном коде, запланированном «на будущее», конструкция `try..catch` не поймает.

Например, в setTimeout асинхронная функция выполнится в стеке позже, когда движок уже покинул конструкцию try..catch. Чтобы поймать исключение внутри запланированной в таймере функции, try..catch должен находиться внутри самой этой функции.

Т.е. неправильно помещать таймер в конструкцию try-catch, надо наоборот конструкцию засовывать в таймер:

```
setTimeout(() => {
```

```
  try {
```

```
    noSuchVariable;
```

```
  }
```

```
  catch {
```

```
    console.log('Поймана!');
```

```
}
```

```
, 100);
```

```
// Поймана!
```

Вот так делать нельзя

```
try {
```

```
  setTimeout(function() {
```

```
    noSuchVariable; // скрипт упадёт тут
```

```
  }, 1000);
```

```
} catch (e) {
```

```
  console.log( "catch" ); // не сработает
```

```
}
```

```
// ReferenceError: noSuchVariable is not defined
```

throw, исключения

Мануал [здесь](#), хорошая статья на professorweb [здесь](#).

Можно создавать собственные исключения. Например, если с т.з. JS код корректный и всё хорошо, но тебя не устраивает результат выполнения, можно создать собственное исключение и перехватить его.

Исключение - это сигнал, указывающий на возникновение какой-либо исключительной ситуации или ошибки.

throw (возбуждение исключения)

это способ просигнализировать о такой ошибке или исключительной ситуации.

catch (перехватить исключение)

значит обработать его, т.е. предпринять действия, необходимые или подходящие для восстановления после исключения.

В JavaScript обработка ошибок работает через механизм исключений. Одни функции их возбуждают, другие обрабатывают через try..catch.

Инструкция throw позволяет генерировать исключения, определяемые пользователем. Выполнение кода после throw не будет продолжено и управление будет передано в ближайший блок catch в стеке вызовов.

Если catch блоков среди вызванных функций нет, выполнение программы будет остановлено.

Сгенерированное исключение можно обработать в блоке catch, а можно не обрабатывать и передать дальше, т.е. «пробросить».

Работает предельно просто: указываешь оператор «throw» и что надо выбросить.

Сгенерировать и выбросить (не уверен, что это правильный термин) можно что угодно: число, строку и т.д., но чаще всего выбрасывается объект ошибки.

Синтаксис:

```
throw <объект ошибки>
```

```
throw "Error2"; // генерирует исключение, значением которого является строка
throw 42;        // ... число 42
```

```
throw true; // ... логическое значение true
```

В действии:

```
const a = 4;

try {
  if (typeof a !== 'string') {
    throw 'неправильный тип данных';
  }

} catch(err) {
  console.log(err);
}

// неправильный тип данных
```

Далее блок catch выполняет всю работу по обработке.

Как было сказано ранее, оператору throw чаще всего передаётся объект ошибки. Объекты могут быть встроенными или пользовательскими:

```
throw new Error('неправильный тип данных');
```

Подробнее об объектах ошибки в разделе «Объект ошибки»

Проброс исключения

Как было сказано ранее, исключение можно обработать в блоке catch, а можно «пробросить» - не обрабатывать, а вернуть наружу. Проброс исключения – это создание ещё одного исключения в блоке catch, которое никто уже ловить не будет.

В блок catch попадают все ошибки, которые появляются в блоке try. Правильно, когда блок catch работает только с теми ошибками, которые ему «известны». Мы пишем блок кода, который обрабатывает определённый тип ошибок. Получается не универсальный обработчик, который обрабатывает вообще все любые ошибки, а специальный, который по-разному работает с каждым известным типом ошибок.

Техника «проброс исключения» выглядит так:

1. Блок catch получает все ошибки.
2. В блоке catch(err) {...} мы анализируем объект ошибки err.
3. Если мы не знаем как её обработать, тогда делаем throw err уже в блоке catch.

Эта тема очень тесно связана с объектом ошибок, но я тут покажу простой пример. Catch в примере ниже умеет обрабатывать только ошибки типа ReferenceError, т.е. «переменная не определена». В случае, если в catch попадает другой тип ошибки, он этот же объект ошибки пробрасывает дальше:

```
try {
  constIsNotDefined;

} catch(err) {

  if (err instanceof ReferenceError) {
    console.log('Переменная не определена')
    // какой-то код для обработки ошибки

  } else {
    throw err;
    // пробросить эту ошибку наружу
  }
}
```

Подробнее о catch

Объект ошибки имеет 3 основных свойства:

name – имя ошибки

message – текстовое сообщение о деталях ошибки.

stack – текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки. Это нестандартное свойство, но поддерживается большинством окружений.

При возникновении ошибки, создаётся объект ошибки и передаётся как аргумент в блок `catch()`

В примере ниже распечатаны основные свойства одной из ошибок:

```
try {
  noSuchVariable
} catch(err) {
  console.log(err.name);      // ReferenceError
  console.log(err.message);   // noSuchVariable is not defined
  console.log(err.stack);     // большое описание, где и что произошло
}
```

Можно распечатать сам объект ошибки. У меня получается то же самое, как если распечатать стек.

С помощью обработки ошибок можно выполнять широкий круг действий.

Например, если пришёл некорректный JSON, можно вообще не связываться с ошибкой, а выполнить несколько сопутствующих действий:

- написать пользователю сообщение через `alert`, почему произошла ошибка, потому что консоль он не увидит;
- отправить новый сетевой запрос;
- предложить альтернативный способ ввода;
- отослать информацию об ошибке на сервер для логирования;
- ... всё лучше, чем просто «падение».

В блок `catch` можно не передавать объект ошибки, если тебе не надо его обрабатывать. Это новая возможность и не всеми браузерами поддерживается. Выглядит вот так, без скобок: `catch {}`

Глобальный catch

Если происходит ошибка вне блоков `try-catch`, её можно поймать и обработать. В JS нет такой функции, но обычно окружения предоставляют соответствующие инструменты.

В Node.js для этого есть [`process.on\("uncaughtException"\)`](#).

В браузере мы можем присвоить функцию специальному свойству [`window.onerror`](#), которая будет вызвана в случае необработанной ошибки.

```
window.onerror = function(message, url, line, col, error) {
  // ...
};
```

message

Сообщение об ошибке.

url

URL скрипта, в котором произошла ошибка.

line, col

Номера строки и столбца, в которых произошла ошибка.

error

Объект ошибки.

Роль глобального обработчика window.onerror обычно заключается не в восстановлении выполнения скрипта – это скорее всего невозможно в случае программной ошибки, а в отправке сообщения об ошибке разработчикам.

Существуют также веб-сервисы, которые предоставляют логирование ошибок для таких случаев, такие как <https://errorception.com> или <http://www.muscula.com>.

Они работают так:

1. Мы регистрируемся в сервисе и получаем небольшой JS-скрипт (или URL скрипта) от них для вставки на страницы.
2. Этот JS-скрипт ставит свою функцию window.onerror.
3. Когда возникает ошибка, она выполняется и отправляет сетевой запрос с информацией о ней в сервис.
4. Мы можем войти в веб-интерфейс сервиса и увидеть ошибки.

Подробнее о finaly

У finaly есть несколько особенностей.

finally и return

В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

```
function func() {  
  
    try {  
        return 1;  
  
    } catch (e) {  
        /* ... */  
    } finally {  
        console.log( 'finally' );  
    }  
}  
  
console.log( func() );  
// finally  
// 1
```

try finally

Конструкция try..finally без секции catch также полезна. Мы применяем её, когда не хотим здесь обрабатывать ошибки (пусть выпадут), но хотим быть уверены, что начатые процессы завершились.

В приведённом коде ошибка всегда выпадает наружу, потому что тут нет блока catch. Но finally отрабатывает до того, как поток управления выйдет из функции:

```
function func() {  
    // начать делать что-то, что требует завершения (например, измерения)  
    try {  
        // ...  
    } finally {  
        // завершить это, даже если все упадёт  
    }  
}
```

Объект ошибки

Встроенный Error

Документация [тут](#).

Во время выполнения кода ошибки приводят к созданию и выбрасыванию новых объектов Error.

Конструктор **Error** создаёт объекты ошибок, а также используется в качестве базового для создания пользовательских исключений с использованием наследования.

Синтаксис создания стандартного объекта ошибки:

```
Error(message)
// можно использовать без new
new Error(message)

message
Описание ошибки, обычная строка
```

Вообще, синтаксис создания ошибки шире. После message, можно также указать fileName, lineNumber, но на практике используют только message, а остальные значения подставляются автоматом. См. подробнее в документации.

Свойства объекта ошибки

.message

Описание и детали ошибки. Это то, что было указано в message при создании.

.name

Название ошибки. Как правило, это название функции-конструктора.

.stack

Текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки. Это нестандартное свойство, но поддерживается большинством окружений.

.constructor – определяет функцию, создающую прототип экземпляра ошибки.

Error.prototype – позволяет добавлять свойства в экземпляры объекта Error.

Условно (!) можно сказать, что класс Error внутри устроен так. Это нужно понимать, чтобы иметь представление, как создавать пользовательские ошибки:

```
// "Псевдокод" встроенного класса Error
class Error {
  constructor(message) {
    this.message = message;
    this.name = "Error"; // (разные имена для разных встроенных классов ошибок)
    this.stack = <стек вызовов>; // нестандартное свойство, но обычно поддерживается
  }
}
```

При возникновении ошибки, создаётся объект ошибки и передаётся как аргумент в блок catch()

В примере ниже распечатаны основные свойства одной из ошибок:

```
try {
  noSuchVariable
} catch(err) {
  console.log(err.name);      // ReferenceError
  console.log(err.message);   // noSuchVariable is not defined
  console.log(err.stack);     // большое описание, где и что произошло
}
```

Можно распечатать сам объект ошибки. У меня получается то же самое, как если распечатать стек.

Для встроенных ошибок свойство name – это в точности имя конструктора. А свойство message берётся из аргумента. Например:

```
let error = new Error("Ого, ошибка! о_0");

alert(error.name); // Error
alert(error.message); // Ого, ошибка! о_0
```

Всё это упаковывается в `throw`:

```
if (!user.name) {
    throw new SyntaxError("Данные неполны: нет имени"); // (*)
}
```

Встроенные конструкторы ошибок

Кроме общего конструктора `Error`, в JavaScript существует ещё семь других основных конструкторов ошибок. По обработке исключений смотрите раздел [Выражения обработки исключений](#).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: `Error`, `SyntaxError`, `ReferenceError`, `TypeError` и другие. Можно использовать и их для создания объектов ошибки.

Их синтаксис:

```
let error = new Error(message);
// или
let error = new SyntaxError(message);
let error = new ReferenceError(message);
// ...
```

ReferenceError

Создаёт экземпляр, представляющий ошибку, возникающую при разыменовывании недопустимой ссылки.

SyntaxError

Создаёт экземпляр, представляющий синтаксическую ошибку, возникающую при разборе исходного кода в функции [eval\(\)](#).

TypeError

Создаёт экземпляр, представляющий ошибку, возникающую при недопустимом типе для переменной или параметра.

EvalError

Создаёт экземпляр, представляющий ошибку, возникающую в глобальной функции [eval\(\)](#).

InternalError

Создаёт экземпляр, представляющий ошибку, возникающую при выбрасывании внутренней ошибки в движке JavaScript. К примеру, ошибки «слишком глубокая рекурсия» («too much recursion»).

RangeError

Создаёт экземпляр, представляющий ошибку, возникающую при выходе числовой переменной или параметра за пределы допустимого диапазона.

URIError

Создаёт экземпляр, представляющий ошибку, возникающую при передаче в функции [encodeURIComponent\(\)](#) или [decodeURIComponent\(\)](#) недопустимых параметров.

Пользовательские ошибки

Можно создавать собственные объекты ошибок на разные случаи, не предусмотренные стандартными конструкторами. Самый очевидный и правильный способ – это расширить стандартный класс Error.

```
class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = 'my Error';
  }
}

try {
  throw new MyError('Сообщение об ошибке');

} catch(err) {
  console.log(err.name);      // my Error
  console.log(err.message);   // Сообщение об ошибке
}
```

По правилам наследования, в конструкторе наследника обязательно должна быть родительская функция super. Родительский конструктор устанавливает свойство message.

This.name – тоже в конструкторе. Это нужно для того, чтобы переопределить свойство .name родительского класса. Если этого не сделать, то его значением будет просто стандартное 'Error', как и у родителя.

Как это используется на практике.

Допустим, есть функция, которая принимает строку JSON и конвертит его её объект. Нужно создать дочерний класс объекта ошибки, который будет проверять, все ли свойства создаваемого объекта получены через строку JSON.

```
// создание пользовательского класса ошибки
// через расширение встроенного класса Error
class MyValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'my Validation Error';
  }
}

// функция конвертит строку JSON в объект
// и проверяет наличие всех необходимых свойств
function readUser(json) {
  const user = JSON.parse(json);

  if (!user.age) {
    throw new MyValidationError('Нет поля .age');
  }
  if (!user.name) {
    throw new MyValidationError('Нет поля .name');
  }
  return user;
}

// запуск блока try...catch
try {
  let user = readUser('{ "age": 25 }');

} catch(err) {

  if (err instanceof MyValidationError) {
    console.log(`Некорректные данные: ${err.message}`);
  }
}
```

```

} else if (err instanceof SyntaxError) {
  console.log(`JSON ошибка синтаксиса: ${err.message}`);
}

} else {
  throw err; // в других случаях - пробросить исключение
}

// Некорректные данные: Нет поля .name

```

`instanceof` используется для проверки конкретного типа ошибки.

В некоторых примерах используется проверка на факт наличия какого-то конкретного свойства, типа `!obj.name`, но версия с `instanceof` лучше, потому что в будущем можно расширить `ValidationError`, сделав его подтипы, такие как `PropertyRequiredError`, и проверка `instanceof` продолжит работать для новых наследованных классов.

Также важно, что если `catch` встречает неизвестную ошибку, то он прорасывает её. Блок `catch` знает, только как обрабатывать ошибки валидации и синтаксические ошибки, а другие виды ошибок (из-за опечаток в коде и другие непонятные) он должен выпустить наружу.

Продвинутое наследование

Сейчас класс `MyValidationError` очень общий, его можно детализировать дальше и выделить из него класс ошибок, который будет отслеживать наличие всех необходимых свойств, переданных в JSON:

`PropertyRequiredError`.

Он будет нести дополнительную информацию о свойстве, которое отсутствует.

```

class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = 'ValidationError';
  }
}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super(`Нет свойства: ${property}`); // это св-во .message!
    this.name = 'PropertyRequiredError';
    this.property = property;
  }
}

const readUser = (json) => {
  const user = JSON.parse(json);

  if (!user.age) {
    throw new PropertyRequiredError('age');

  } else if (!user.name) {
    throw new PropertyRequiredError('name');
  }

  return user;
};

try {
  const user = readUser('{ "age": 25 }');

} catch(err) {
  if (err instanceof ValidationError) { // указан родительский класс
    console.log(`Неверные данные: ${err.message}`);
    console.log(err.name);
    console.log(err.property);
}

```

```

} else if (err instanceof SyntaxError) {
  console.log(`Ошибка синтаксиса: ${err.message}`);
}

} else {
  throw err;
}

// Неверные данные: Нет свойства: name
// PropertyRequiredError
// name

```

В этом примере есть непонятное место.

В блоке catch реализуется проверка на материнский класс, хотя экземпляр ошибки принадлежит к дочернему. По умолчанию, все дочерние классы пройдут проверку instanceof на материнский класс. Идея состоит в том, чтобы ловить сразу всю группу ошибок, это делается через материнский класс. При этом сами ошибки – инстансы дочернего класса, и свойства .name записаны в них как предписывает дочерний класс. Получается, что все ошибки можно отловить с помощью материнского класса, а потом вывести их индивидуальные свойства.

свойство this.name в конструкторе PropertyRequiredError снова присвоено вручную. Правда, немного утомительно – присваивать this.name = <class name> в каждом классе пользовательской ошибки. Можно этого избежать, если сделать наш собственный «базовый» класс ошибки, который будет ставить this.name = **this.constructor.name**. И затем наследовать все ошибки уже от него.

Назовём его MyError.

Вот упрощённый код с MyError и другими пользовательскими классами ошибок.

Теперь пользовательские ошибки стали намного короче, особенно ValidationError, так как мы избавились от строки "this.name = ..." в конструкторе.

```

class MyError extends Error {
  constructor(message) {
    super(message);
    this.name = this.constructor.name;
  }
}

class ValidationError extends MyError {}

class PropertyRequiredError extends ValidationError {
  constructor(property) {
    super(`Нет свойства: ${property}`);
    this.property = property;
  }
}

console.log( new PropertyRequiredError('field').name );
// PropertyRequiredError

```

Обёртывание исключений

Какая-то хрень.

В последнем примере для каждого вида ошибки в конструкции readUser создавался свой блок if чтобы выбросить исключение. Так делать оч долго, если пользовательских ошибок много.

Исключения можно обрачивать.

ReadError будет распознавать ошибки пользовательских классов.

```

class ReadError extends Error {
  constructor(message, cause) {
    super(message);
    this.cause = cause;
    this.name = 'ReadError';
  }
}

class ValidationError extends Error { /*...*/ }
class PropertyRequiredError extends ValidationError { /* ... */ }

function validateUser(user) {
  if (!user.age) {
    throw new PropertyRequiredError("age");
  }

  if (!user.name) {
    throw new PropertyRequiredError("name");
  }
}

function readUser(json) {
  let user;

  try {
    user = JSON.parse(json);
  } catch (err) {
    if (err instanceof SyntaxError) {
      throw new ReadError("Синтаксическая ошибка", err);
    } else {
      throw err;
    }
  }

  try {
    validateUser(user);
  } catch (err) {
    if (err instanceof ValidationError) {
      throw new ReadError("Ошибка валидации", err);
    } else {
      throw err;
    }
  }
}

try {
  readUser('{bad json}');
} catch (e) {
  if (e instanceof ReadError) {
    // Исходная ошибка: SyntaxError:Unexpected token b in JSON at position 1
  } else {
    throw e;
  }
}

```

В коде `readUser` распознаёт синтаксические ошибки и ошибки валидации и выдаёт вместо них ошибки `ReadError`. Неизвестные ошибки, как обычно, прорасыываются.

Внешний код проверяет только `instanceof ReadError`. Не нужно перечислять все возможные типы ошибок. Этот подход называется «обёртывание исключений», потому что мы берём «исключения низкого уровня» и «оборачиваем» их в `ReadError`, который является более абстрактным и более удобным для использования в вызывающем коде. Такой подход широко используется в объектно-ориентированном программировании.

Обёртывание исключений является распространённой техникой: функция ловит низкоуровневые исключения и создаёт одно «высокоуровневое» исключение вместо разных низкоуровневых. Иногда низкоуровневые исключения становятся свойствами этого объекта, как err.cause в примерах выше, но это не обязательно.

///

JS - Промисы, async/await

Статья на английском про асинхронность в книге «Вы не знаете JS» [здесь](#).

Цикл курсов про асинхронность на хабре [здесь](#).

Видео про Event Loop [здесь](#).

Основные понятия асинхронного программирования, мануал – [здесь](#).

Статья на Проглиб: [здесь](#).

Есть специальные библиотеки для работы в асинхронном стиле:

FS (file system) – встроенный модуль для ноды: [оригинал](#), [перевод](#).

[async](#) – содержит десятки функций для большого числа задач, связанных с упорядочиванием асинхронных операций.

Документация по Path: [оригинал](#), [перевод](#).

Введение: колбэки

Пример синхронного кода для начала темы:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}

loadScript('test.js');
console.log('текст после вызова функции');
```

Функция loadScript создаёт тег скрипт и загружает его на страницу.

На практике, дальнейший код в текущем файле не будет ждать, пока элемент будет добавлен на страницу и продолжит выполняться дальше.

Так, если запустить пример выше, то сначала выполнится печать «текст после вызова функции», а потом выполнится скрипт test.js. Если бы в этом же документе надо было использовать данные загружаемого скрипта, то произошла бы ошибка: скрипт к этому времени ещё не загружен.

Чтобы решить эту проблему, можно сделать асинхронную функцию, которая выполнится не сразу же, как только до неё дойдёт интерпретатор, а потом, при достижении определённых условий.

В качестве такого условия, можно прописать событие onload: как только скрипт загрузится на страницу, можно запускать какую-то другую функцию, использующую его данные.

В асинхронную функцию передаётся callback – функция обратного вызова, которая запускается при достижении нужных условий. Сама же функция callback принимает 2 аргумента: ошибку и собственно исполняющий код. Если в результате выполнения произошла ошибка, то она передаётся первым аргументом, callback обрабатывает её и завершает работу.

Если ошибки не было, то её значение равняется null и callback работает в штатном режиме.

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
```

```
script.onload = () => callback(null, script);
script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));
document.head.append(script);
}
```

Hexlet

Асинхронные функции никогда не возвращают результат асинхронной операции. Единственный способ получить результат — описать логику в колбеке.

Использование return всегда делает выход из АФ и возвращает undefined.

При этом инструкция return не вернёт результат, а выполнит выход из функции. Return можно использовать как guard expression, прекращая дальнейшие вычисления.

Как только происходит ошибка, мы вызываем основной колбек и отдаём туда ошибку. Если ошибка не возникла, то мы всё равно вызываем исходный колбек и передаём туда null. Вызывать его обязательно, иначе внешний код не дождётся окончания операции. Следующие вызовы, после вызова с ошибкой, больше не выполняются:

```
import fs from 'fs';

const unionFiles = (inputPath1, outputPath, cb) => {
  fs.readFile(inputPath1, 'utf-8', (error1, data1) => {
    if (error1) {
      cb(error1);
      return;
    }
    fs.readFile(inputPath2, 'utf-8', (error2, data2) => {
      if (error2) {
        cb(error2);
        return;
      }
      fs.writeFile(outputPath, `${data1}${data2}`, (error3) => {
        if (error3) {
          cb(error3);
          return;
        }
        cb(null); // не забываем последний успешный вызов
      });
    });
  });
}
```

Последний вызов можно сократить. Если в самом конце не было ошибки, то вызов cb(error3) отработает так же, как и cb(null), а значит, весь код последнего колбека можно свести к вызову cb(error3):

```
fs.writeFile(outputPath, `${data1}${data2}`, cb);
// что равносильно fs.writeFile(outputPath, `${data1}${data2}`, error3 => cb(error3));
```

Промисы

Объект promise

Promise - это объект, возвращаемый функцией, которая ещё не завершила свою работу.

Когда вызванная функция асинхронно завершает работу, этот объект переходит в соответствующее состояние и вызывает обработчик для дальнейшей работы с результатом асинхронной операции.

Объект Promise может находиться в трёх состояниях (свойство state, нет прямого доступа):

pending, ожидание	начальное состояние, не исполнен и не отклонён.
fulfilled, исполнено	операция завершена успешно.

```
rejected, отклонено    операция завершена с ошибкой.
```

Результат выполнения записывается в свойство `result` (нет прямого доступа) и может быть равен:

```
undefined  с самого начала
value      при вызове resolve(value)
error      при вызове reject(error)
```

Синтаксис создания промиса:

```
let promise = new Promise(executor);
let promise = new Promise(function(resolve, reject) { ... });

promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

Самому прописывать аргументы не надо: `resolve` и `reject` – функции, встроенные в JS.

В функции `executor` описывается выполнение асинхронной работы, а потом передаётся наружу через функции `resolve` или `reject`. Если что-то возвращать из функции через `return`, то результат будет проигнорирован.

[Promise.resolve\(value\)](#)

Возвращает промис, исполненный с результатом `value`. Вызывается, когда операция завершилась успешно и вернула результат своего исполнения в виде значения.

[Promise.reject\(reason\)](#)

Возвращает промис, отклонённый из-за `reason`. Вызывается, когда операция не удалась и возвращает значение, указывающее на причину неудачи, чаще всего объект ошибки. Если в промисе произошла ошибка, она вернётся через `.reject`

Промисы для любых функций

Чтобы снабдить свою функцию функциональностью промисов, нужно просто вернуть в ней объект `Promise`:

```
function myAsyncFunction(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
  });
}
```

Пример из учебника (тут ошибка по ходу, `.append` должен быть за пределами функции)

```
function loadScript(src) {

  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));
  });
}
```

```
    document.head.append(script);
  });
}
```

Обработчики решения

Результат, возвращённый из промиса, можно продолжить обрабатывать дальше через **обработчики решения**. Это методы промиса .then, .catch и .finally.

.then(onFulfilled, onRejected)

У этого обработчика два аргумента, они должны быть функциями-обработчиками результата выполнения промиса. Какая из них запустится, зависит от статуса выполненного промиса (value или err).

В первую функцию-обработчик передаётся результат успешного выполнения промиса, во вторую – ошибку выполнения промиса.

.then возвращает новый промис.

Если промис не был обработан (т.е. если аргументы .then – это не функции), то вернётся оригинальное значение.

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

.catch(onRejected)

Добавляет функцию для обработки ошибки и возвращает новый промис.

.catch(f) является аналогом .then(null, f).

.finally()

Переданный в .finally обработчик выполнится в любом случае, успешно ли завершился промис или с ошибкой. Хорошо подходит, например, для остановки индикатора загрузки, потому что его нужно остановить вне зависимости от результата.

Отличия от then(f,f):

- Обработчик, вызываемый из finally, не имеет аргументов. В finally мы не знаем, как был завершён промис. Это правильно, потому что обычно наша задача – выполнить «общие» завершающие процедуры.
- Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

Цепочка промисов

В обработчиках можно создавать промисы и возвращать их как обычный результат.

Явное создание промисов

Вообще, обработчики сами по себе возвращают результат в виде промиса, но в этом примере промис создаётся и возвращается явно вручную:

```
new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
})

.then(function(result) {
  alert(result); // 1
  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });
});
```

```
}).then(function(result) { // (**)
  alert(result); // 2
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });
}).then(function(result) {
  alert(result); // 4
});
```

Хороший комментарий: «Явно мы сами возвращаем промис, когда хотим не просто вернуть ответ, а еще и выполнить какую-то работу, которая занимает время. Тогда then будет ждать, когда наш созданный явный промис выполнится и отдаст ей результат».

Таким образом, выводятся 1, 2, 4, но сейчас между вызовами alert существует пауза в 1 секунду.

Цепочка .then

Можно выстраивать обработчики друг за другом. Так, например, скрипты будут загружены по очереди и в последнем скрипте можно использовать замыкание из первого:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));
    document.head.append(script);
  });
}

loadScript("/article/promise-chaining/one.js")
  .then(script => loadScript("/article/promise-chaining/two.js"))
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // скрипты загружены, мы можем использовать объявленные в них функции
    one();
    two();
    three();
});
```

«Пирамида» .then

Технически мы бы могли добавлять .then напрямую к каждому вызову loadScript, как в примере ниже. Этот код делает то же самое: последовательно загружает 3 скрипта. Но он «растёт вправо», так что возникает такая же проблема, как и с колбэками.

Иногда всё же приемлемо добавлять .then напрямую, чтобы вложенная в него функция имела доступ к внешней области видимости. В примере выше самая глубоко вложенная функция обратного вызова имеет доступ ко всем переменным script1, script2, script3. Но это скорее исключение, чем правило (какое-то противоречие, потому что пример выше тоже вызывает функции из вложенных скриптов).

```
loadScript("/article/promise-chaining/one.js").then(script1 => {
  loadScript("/article/promise-chaining/two.js").then(script2 => {
    loadScript("/article/promise-chaining/three.js").then(script3 => {
      // эта функция имеет доступ к переменным script1, script2 и script3
      one();
      two();
      three();
    });
});
```

```
});  
});
```

Использование внешних переменных для «протаскивания» данных

Если соединять несколько then друг с другом, то не получится протянуть какие-то аргументы от первого вызова к последнему:

```
import { promises as fs } from 'fs';  
  
const unionFiles = (inputPath1, outputPath) => {  
  const result = fs.readFile(inputPath1, 'utf-8')  
    .then((data1) => fs.readFile(inputPath2, 'utf-8'))  
    .then((data2) => fs.writeFile(outputPath, `${data1}${data2}`));  
  return result;  
};
```

Data 1 сюда уже не попадёт, она находится в локальной области видимости функции 1.

Выход из этой ситуации – после then создать переменные и протащить их от 1 вызова до конца:

```
import { promises as fs } from 'fs';  
  
const unionFiles = (inputPath1, outputPath) => {  
  let data1;  
  
  return fs.readFile(inputPath1, 'utf-8')  
    .then((content) => {  
      data1 = content; //  
    })  
    .then(() => fs.readFile(inputPath2, 'utf-8'))  
    .then((data2) => fs.writeFile(outputPath, `${data1}${data2}`));  
};
```

В этом месте содержимое промиса уже точно записывается в константу и не пропадает.

thenable

Обработчик может возвращать не промис, а любой объект, содержащий метод .then, такие объекты называют «thenable», и этот объект будет обработан как промис.

Смысл в том, что сторонние библиотеки могут создавать свои собственные совместимые с промисами объекты. Они могут иметь свои наборы методов и при этом быть совместимыми со встроенными промисами, так как реализуют метод .then.

Это позволяет добавлять в цепочки промисов пользовательские объекты, не заставляя их наследовать от Promise.

Вот пример такого объекта:

```
class Thenable {  
  constructor(num) {  
    this.num = num;  
  }  
  then(resolve, reject) {  
    alert(resolve); // function() { native code }  
    // будет успешно выполнено с аргументом this.num*2 через 1 секунду  
    setTimeout(() => resolve(this.num * 2), 1000); // (**)  
  }  
}  
  
new Promise(resolve => resolve(1))  
  .then(result => {  
    return new Thenable(result); // (*)  
  })  
  .then(alert); // показывает 2 через 1000мс
```

Промисы и ошибки

Управление в ближайший .catch

Если промис завершается с ошибкой, то управление переходит в ближайший обработчик ошибок. .catch не обязательно должен быть сразу после ошибки, он может быть после нескольких .then. Самый лёгкий путь перехватить все ошибки – это добавить .catch в конец цепочки.

```
fetch('url')
  .then()
  .then()
  .then()
  .catch(error => alert(error.message));
```

Невидимый try..catch

Вокруг функции промиса и обработчиков находится "невидимый try..catch". Если происходит исключение, то оно перехватывается, и промис считается отклонённым с этой ошибкой. Иными словами, любая ошибка в промисе автоматом делает его отклонённым.

Два кода ниже работают одинаково:

```
new Promise((resolve, reject) => {
  throw new Error("Ошибка!");
}).catch(alert);

new Promise((resolve, reject) => {
  reject(new Error("Ошибка!"));
}).catch(alert);
```

Можно нормально завершить цепочку промисов, даже если в ней была ошибка.

Для этого нужно после обработчика .catch продолжить по цепочке другие обработчики, даже если из .catch вернуть переданный в неё объект ошибки. В коде ниже считается, что ошибка обработана и всё ок, можно продолжать выполнение:

```
new Promise((resolve, reject) => {
  throw new Error("Ошибка!");
})
.catch(error => console.log("ошибка обработана"))
.then(() => console.log("then работает нормально"));

// ошибка обработана index.js:4:25
// then работает нормально
```

А в этом примере блок .catch тоже перехватывает ошибку, но не может обработать её, поэтому ошибка пробрасывается далее:

```
new Promise((resolve, reject) => {
  throw new Error("Ошибка!");
})
.catch(function(error) {
```

```
        console.log("Не могу обработать ошибку");
        throw error;
    })

    .then(function() {
        // не сработает
    })

    .catch(error => {
        // ничего не возвращаем => выполнение продолжается в нормальном режиме
    });
}
```

Необработанные ошибки

Если забыть добавить .catch и не обработать ошибку в промисе, то скрипт упадёт, как если бы это была обычная необработанная ошибка: JavaScript-движок отслеживает такие ситуации и генерирует в этом случае глобальную ошибку.

В браузере мы можем поймать такие ошибки, используя событие unhandledrejection.

Если происходит ошибка, и отсутствует её обработчик, то генерируется это событие и соответствующий объект event содержит информацию об ошибке.

```
window.addEventListener('unhandledrejection', function(event) {
    // объект события имеет два специальных свойства:
    alert(event.promise); // [object Promise] - промис, который сгенерировал ошибку
    alert(event.reason); // Error: Ошибка! - объект ошибки, которая не была обработана
});
```

Promise API

В классе [Promise](#) есть 5 статических методов.

Promise.all

Ожидает исполнения всех промисов или отклонения любого из них.

Возвращает промис, который исполнится после исполнения всех промисов в iterable.

В случае, если любой из промисов будет отклонён, Promise.all будет также отклонён.

```
let promise = Promise.all([promise1, promise2, ...]);
```

Принимает массив промисов (любой перебираемый объект). Новый промис завершится, когда завершится весь переданный список промисов, и его результатом будет массив их результатов.

Если передан объект, который не является промисом, то он будет преобразован с помощью метода `Promise.resolve`.

При этом результат будет собран в таком же порядке, в каком промисы были переданы. Даже если передать несколько неодинаковых таймеров, результат будет с соблюдением очерёдности передачи:

```
Promise.all([
    new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
    new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
    new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(console.log);

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

.map и Promise.all

Часто применяемый трюк – пропустить массив данных через map-функцию, которая для каждого элемента создаст задачу-промис, и затем обернёт получившийся массив в Promise.all.

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// Преобразуем каждый URL в промис, возвращённый fetch
let requests = urls.map(url => fetch(url));

// Promise.all будет ожидать выполнения всех промисов
Promise.all(requests)
  .then(responses => responses.forEach(
    response => console.log(`#${response.url}: ${response.status}`))
);
```

Если попробовать распечатать этот массив после map, то он будет выглядеть так:

```
const promises = filepaths.map((filepath) => fs.readFile(filepath, 'utf-8'));
console.log(promises);
[
  Promise { <pending> },
  Promise { <pending> },
  Promise { <pending> },
  Promise { <pending> },
]
```

Ошибки и Promise.all

Если хотя бы один промис завершился с ошибкой, весь результат Promise.all будет помечен как ошибочный. Чтобы этого избежать, можно передавать в Promise.all промисы с повешенными на них обработчиками .catch, из которых уже возвращаются данные с пометкой об успешности:

```
const promises = filepaths.map((filepath) => fs.readFile(filepath, 'utf-8')
  .then((v) => ({ result: 'success', value: v }))
  .catch((e) => ({ result: 'error', error: e })));
const promise = Promise.all(promises);
```

Promise.allSettled

Ожидает завершения всех полученных промисов (как исполнения так и отклонения).

Возвращает промис, который исполняется когда все полученные промисы завершены, содержащий массив результатов исполнения полученных промисов.

Массив будет содержать объекты в формате { status: 'fulfilled' / 'rejected', value: 'value' }.

```
let promise = Promise.allSettled([promise1, promise2, ...]);

выполнен
{ status: 'fulfilled', value: 'value' }
отклонён
{ status: 'rejected', reason: 'value' }
```

Promise.race

Возвращает результат или ошибку только того промиса, который выполнится быстрее всех. После этого остальные промисы игнорируются.

[Promise.resolve\(value\)](#)

[Promise.reject\(reason\)](#)

Редко используются в современном коде из-за синтаксиса `async/await`.

Возвращает исполненный/отклонённый промис с результатом `value / reason`.

`Promise.resolve(value)` создаёт успешно выполненный промис с результатом `value`. Этот метод используют для совместимости: когда ожидается, что функция возвратит именно промис.

```
let promise = new Promise(resolve => resolve(value));
```

`Promise.reject(error)` создаёт промис, завершённый с ошибкой `error`. На практике почти не используется.

```
let promise = new Promise((resolve, reject) => reject(error));
```

Промисификация

Промисификация – это преобразование функции на колбэках в промисы.

Многие библиотеки основаны на колбэках, есть смысл их «промисифицировать», потому что промисы удобнее.

Существуют модули с гибкой промисификацией, например, [es6-promisify](#) или встроенная функция `util.promisify` в Node.js.

? Помните, промис может иметь только один результат, но колбэк технически может вызываться сколько угодно раз. Поэтому промисификация используется для функций, которые вызывают колбэк только один раз.

Последующие вызовы колбэка будут проигнорированы.

```
// было
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

  document.head.append(script);
}

// использование: loadScript('path/script.js', (err, script) => {...})

// стало
let loadScriptPromise = function(src) {
  return new Promise((resolve, reject) => {
    loadScript(src, (err, script) => {
      if (err) reject(err)
      else resolve(script);
    });
  })
}

// использование: loadScriptPromise('path/script.js').then(...)
```

Функция-обёртка для промисификации

Можно сделать универсальную обёртку для промисификации. Эта обёртка такая же, как функция выше, но если исходная `f` ожидает колбэк с большим количеством аргументов `callback(err, res1, res2, ...)`, то при вызове `promisify(f, true)` результатом промиса будет массив результатов `[res1, res2, ...]`:
(не понимаю, как это работает)

```
// promisify(f, true), чтобы получить массив результатов
function promisify(f, manyArgs = false) {
    return function (...args) {

        return new Promise((resolve, reject) => {
            function callback(err, ...results) { // наш специальный колбэк для f
                if (err) {
                    return reject(err);
                } else {
                    // делаем resolve для всех results колбэка, если задано manyArgs
                    resolve(manyArgs ? results : results[0]);
                }
            }

            args.push(callback);

            f.call(this, ...args);
        });
    };
}

// использование:
f = promisify(f, true);
f(...).then(arrayOfResults => ..., err => ...)
```

Микрозадачи

Обработчики `.then/catch/finally` всегда асинхронны, поэтому даже если в них передаётся готовое решение, они вызываются только после выполнения текущего синхронного кода (который находится под ними). Пример:

```
let promise = Promise.resolve(console.log('1')); // выполнится сразу

promise.then(() => console.log('2')); // всегда асинхронный

console.log('3'); // выполнится вторым

// 1, 3, 2
```

Почему `.then` срабатывает позже?

Асинхронные задачи требуют правильного управления, для этого стандарт предусматривает внутреннюю очередь `PromiseJobs`, более известную как «очередь микрозадач» (`microtask queue`).

Когда промис выполнен, его обработчики `.then/catch/finally` попадают в конец этой очереди. Если есть цепочка с несколькими `.then/catch/finally`, то каждый из них ставится в очередь, а потом выполняется, когда добавленные ранее в очередь обработчики выполнены.

Как сказано в [спецификации](#):

- задачи, попавшие в очередь первыми, выполняются тоже первыми (FIFO).
- выполнение задачи происходит только в том случае, если ничего больше не запущено (?).

Теперь мы можем описать, как именно JavaScript понимает, что ошибка не обработана: необработанная ошибка возникает в случае, если ошибка промиса не обрабатывается в конце очереди микрозадач. Если мы забудем добавить `.catch`, то, когда очередь микрозадач опустеет, движок генерирует событие `«unhandledrejection»`. Событие `unhandledrejection` возникает, когда очередь микрозадач завершена: движок проверяет все промисы и, если какой-либо из них в состоянии `«rejected»`, то генерируется это событие.

Async/Await

Связка `async/await` нужна для того, чтобы работать с АФ как с обычными функциями. Это «синтаксический сахар» для получения результата промиса, более наглядный, чем `promise.then`.

Хотя при работе с `async/await` можно обходиться без `promise.then/catch`, иногда всё-таки приходится использовать эти методы (на верхнем уровне вложенности, например). Также `await` отлично работает в сочетании с `Promise.all`, если необходимо выполнить несколько задач параллельно.

async

Ключевое слово `async` ставится перед какой-нибудь функцией и такая функция всегда будет возвращать промис. Например, эта функция возвратит промис с результатом 1:

```
async function f() {  
    return 1;  
}  
  
console.log(f());  
// Promise { 1 }  
  
f().then(console.log);  
// 1
```

await

Ключевое слово `await` используется только внутри `async`-функций. `Await` заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от `await` не выполнится. После чего он вернёт его результат и выполнение кода продолжится.

Подряд идущие `await` в рамках одной функции всегда выполняются строго друг за другом. Проще всего это понимать если представлять код как цепочку промисов, где каждая следующая операция выполняется внутри `then`.

Получается, что они выполняются последовательно, а не параллельно. Чтобы выполнение было параллельным, надо использовать функцию `promise.all`. Используя только `async/await` невозможно одновременно запускать несколько промисов.

```
let value = await promise;
```

В этом примере промис успешно выполнится через 1 секунду:

```
async function f() {  
    let promise = new Promise((resolve, reject) => {  
        setTimeout(() => resolve("готово!"), 1000)  
    });  
  
    let result = await promise; // будет ждать, пока промис не выполнится (*)  
    console.log(result); // "готово!"  
}  
  
f();
```

В данном примере выполнение функции остановится на строке (*) до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную `result` будет записан результат выполнения промиса.

Стрелочные функции

Тоже используются:

```
самостоятельная функция  
const foo = async () => {...}
```

```
метод объекта
foo = async () => {...}
```

IIFE

Можно обернуть код в анонимную async-функцию:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Если у объекта можно вызвать метод `then`, этого достаточно, чтобы использовать его с `await`.

Обратите внимание, хотя `await` и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.

await нельзя использовать на верхнем уровне вложенности

из-за того, что `await` работает только внутри async-функций, так сделать не получится:

```
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();

// SyntaxError
```

Но можно сделать анонимную функцию-обёртку, и всё заработает:

```
(async () => {
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();
  ...
})();
```

Использование с promise.all

Можно сделать так с использованием `promise.all`, и файлы начнут читаться одновременно, а не последовательно:

```
const unionFiles = async (inputPath1, inputPath2, outputPath) => {
  // Эти вызовы начинают чтение почти одновременно и не ждут друг друга
  const promise1 = fs.readFile(inputPath1, 'utf-8');
  const promise2 = fs.readFile(inputPath2, 'utf-8');

  // Теперь дожидаемся когда они оба завершатся
  // Данные можно сразу разложить
  const [data1, data2] = await Promise.all([promise1, promise2]);
  await fs.writeFile(outputPath, `${data1}${data2}`);
};
```

await работает с «thenable»-объектами

Как и `promise.then`, `await` позволяет работать с промис-совместимыми объектами. Идея в том, что если у объекта можно вызвать метод `then`, этого достаточно, чтобы использовать его с `await`.

Асинхронные методы классов

Для объявления асинхронного метода достаточно написать `async` перед именем:

```
class Waiter {  
  async wait() {  
    return await Promise.resolve(1);  
  }  
}  
  
new Waiter()  
.wait()  
.then(alert); // 1
```

Обработка ошибок

Когда промис завершается успешно, `await promise` возвращает результат. Когда завершается с ошибкой – будет выброшено исключение, как если бы на этом месте находилось выражение `throw`. Если забыть добавить `.catch`, то будет сгенерирована ошибка «`Uncaught promise error`» и информация об этом будет выведена в консоль.

```
// такой код:  
async function f() {  
  await Promise.reject(new Error("Упс!"));  
}  
  
// Делает тоже самое, что и такой:  
async function f() {  
  throw new Error("Упс!");  
}
```

Можно использовать с конструкцией `try-catch` и ошибки будут отловлены:

```
import { promises as fs } from 'fs';  
  
const unionFiles = async (inputPath1, inputPath2, outputPath) => {  
  try {  
    const data1 = await fs.readFile(inputPath1, 'utf-8');  
    const data2 = await fs.readFile(inputPath2, 'utf-8');  
    await fs.writeFile(outputPath, `${data1}${data2}`);  
  } catch (e) {  
    console.log(e);  
    throw e; // снова бросаем, потому что вызывающий код должен иметь возможность отловить  
ошибку  
  }  
};
```

Можно использовать метод промисов `.catch` уже после синтаксиса `async`:

```
async function f() {  
  let response = await fetch('http://no-such-url');  
}  
  
// f() вернёт промис в состоянии rejected  
f().catch(alert); // TypeError: failed to fetch
```

Hexlet

Традиционная задача про объединение файлов:

```
// Код на колбеках
import fs from 'fs';

fs.readFile('./first', 'utf-8', (error1, data1) => {
  if (error1) {
    console.log('boom!');
    return;
  }
  fs.readFile('./second', 'utf-8', (error2, data2) => {
    if (error2) {
      console.log('boom!');
      return;
    }
    fs.writeFile('./new-file', `${data1}${data2}`, (error3) => {
      if (error3) {
        console.log('boom!');
      }
    });
  });
});
```

```
// Код на промисах
import { promises as fsPromises } from 'fs';

let data1;
fsPromises.readFile('./first', 'utf-8')
  .then((d1) => {
    data1 = d1;
    return fsPromises.readFile('./second', 'utf-8');
  })
  .then((data2) => fsPromises.writeFile('./new-file', `${data1}${data2}`))
  .catch(() => console.log('boom!'));
```

```
// То же самое с Async/Await
const unionFiles = async (inputPath1, inputPath2, outputPath) => {
  // Как и в примере выше, эти запросы выполняются строго друг за другом,
  // хотя при этом не блокируется программа, это значит, что другой код тоже может
  // выполняться во время этих запросов)
  // В реальной жизни чтение файлов лучше выполнять параллельно через Promise.all

  const data1 = await fsPromises.readFile(inputPath1, 'utf-8');
  const data2 = await fsPromises.readFile(inputPath2, 'utf-8');
  await fsPromises.writeFile(outputPath, `${data1}${data2}`);
};
```

Промисы и fetch

Сложная тема, смотреть её вместе с fetch темой.

Комментарий:

«Чтобы понять примеры, внимание обратите на след. цитаты. Из этого и предыдущего уроков:
Если промис в состоянии ожидания, обработчики в .then/catch/finally будут ждать его. Однако, **если промис уже завершён, то обработчики выполняются сразу.**
Обработчик handler, переданный в .then(handler), может вернуть промис. В этом случае дальнейшие обработчики ожидают, пока этот промис выполнится, и затем получают его результат.

Важно понимать, что асинхронный код обрабатывает только промис. .then/catch/finally - обычные функции. в примерах же, все функции которые делают асинхронные запросы, уже либо обернуты в промис (лоадСкрипт) либо возвращают промис как результат реализации (фетч). Поэтому в цепочке .then они выполняются друг за

другом, но если сделать к примеру пару .then с синхронным кодом, потом отправить в .then запрос на сервер, без промиса, и снова продолжить .then с синхронным - паравоз будет нестись не ожидая ответа».

Промисы часто используются, чтобы делать запросы по сети. Базовый синтаксис:

```
let promise = fetch(url);
```

Запрос работает так:

1. код запрашивает url по сети и возвращает промис.
2. Промис ждёт, пока сервер вернёт пришлёт заголовки ответа и возвращает объект response. У этого объекта много своих методов, например: response.text(), response.json()
3. Чтобы прочитать полный ответ, надо вызвать метод response.text(). Он тоже возвращает промис, который выполняется, когда данные полностью загружены с удалённого сервера, и возвращает эти данные.

```
fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then( (response) => response.text() )
  .then( (text) => console.log(text) ); // текст

fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then((response) => response.json())
  .then((user) => console.log(user) ); // объект
```

Полученные данные можно отправить на Гитхаб и скачать аватар пользователя:

```
fetch('https://learn.javascript.ru/article/promise-chaining/user.json')
  .then((response) => response.json())
  // запрос на гитхаб
  .then(obj => fetch(`https://api.github.com/users/${obj.name}`))
  .then(response => response.json())
  // на основе полученных данных сделать изображение
  .then(githubUser => new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    // через 3 секунды удалить
    setTimeout(() => {
      img.remove();
      resolve(githubUser); // (**)
    }, 3000);
  }))

  .then(githubUser => alert(`Закончили показ ${githubUser.name}`))
};
```

обработчик .then в строке // 1 будет возвращать new Promise, который перейдёт в состояние «выполнен» только после того, как в строке // 2 таймер вызовет resolve(githubUser). Соответственно, следующий по цепочке .then будет ждать этого.

Как правило, все асинхронные действия должны возвращать промис.

///

fs.promises API

Документация по FS: [перевод](#).

Для того, чтобы работать с функциями было ещё проще, в нодовской библиотеке [File System](#) есть соответствующий раздел:

«The fs.promises API provides an alternative set of asynchronous file system methods that return Promise objects rather than using callbacks. The API is accessible via require('fs').promises»

Очень полезная и удобная штука. К ней можно обращаться так:

```
import { promises as smth } from 'fs';

или https://nodejs.org/api/fs.htmls
let fs = require('fs');
let fsPromises = fs.promises;

или
fs.promises.method(arg, option)
```

У этого метода есть свои такие же методы, как и у обычного fs, только в них не надо передавать колбек. Просто аргументы и свойства, а затем обработчик .then

Схема

```
fs.promises.method(arg, option).then( callback_ok, callback_error )
```

Пример работы

```
const copy = (src, dest) =>
  promises.readFile(src, 'utf-8').then((content) => fs.writeFile(dest, content));
```

fs.readFile

fs.writeFile

[fs.readdir](#) - чтение содержимого директории

```
fs.readdirSync(dirpath);
```

[fs.stat](#) - информация о файле

```
fs.statSync(path.join(dirpath, fname))]
```

Задача

Реализуйте и экспортируйте асинхронную функцию reverse, которая изменяет порядок расположения строк в файле на обратный:

```
// one
// two
reverse(filepath);

// two
// one
```

```
import { promises as fsPromises } from 'fs';

export const reverse = (file) => fsPromises.readFile(file, 'utf-8')
  .then((data) => String(data))
  .then((str) => str.split('\n').reverse().join('\n')) // здесь готовый для записи текст
  .then((str) => fsPromises.writeFile(file, str));
```

препод

```
export const reverse = (filepath) => fs.readFile(filepath, 'utf-8')
  .then((data) => {
    const preparedData = data.split('\n').reverse().join('\n');
    return fs.writeFile(filepath, preparedData);
  });
}
```

JS – Асинхронное программирование

Понятие

В синхронном коде каждая операция ожидает окончания предыдущей. Поэтому вся программа может зависнуть, пока одна из команд выполняется очень долго.

Асинхронный код убирает блокирующую операцию из основного потока программы, так что она продолжает выполняться, но где-то в другом месте, а обработчик может идти дальше. Проще говоря, главный "процесс" ставит задачу и передает ее другому независимому "процессу". В этом случае главный поток выполнения разделяется на две ветви.

Асинхронные функции при своей работе не блокируют программу во время ввода-вывода информации. Например, при копировании или чтении файлов. Как правило, асинхронные функции всегда имеют дело с внешней средой. Например, операционной системой или сетью при копировании файлов. Асинхронная функция отработает потом, а не сразу же. При этом выполнение основной программы продолжится и не будет ждать отработки этой функции, потому что у асинхронных функций свой стек вызовов.

Асинхронное программирование и многопоточное – это разные вещи. Асинхронность базируется на неблокирующем IO и выполняется в одном потоке в рамках event loop. Асинхронно значит "в другое время" а не "параллельно" ??? Другие лекторы говорят, что параллельно. Сам код выполняется последовательно.

Реализовать АФ можно путём использования:

1. callback функций
2. promise
3. связки async/await

Отличие от синхронных функций

В синхронном коде выполнение функций происходит в том же месте, где они были вызваны, и в тот момент, когда происходит вызов. Примеры рассматриваются с модулем `fs` для Node. Синхронные функции этого модуля заканчиваются на Sync:

```
import fs from 'fs';

// Обязательно передавать вторым параметром `utf-8`,
// только тогда данные возвращаются в строковом представлении
const content = fs.readFileSync('./myfile', 'utf-8');
fs.writeFileSync('./myfile-copy', content);
```

Сначала читается содержимое файла в константу `content`, затем оно же записывается в другой файл. Каждая строчка приводит к **блокировке**, то есть выполнение программы ждёт, пока операционная система прочитает файл (а это делает именно она) и отдаст его содержимое программе, и только затем выполняется следующая строчка. Любые файловые операции занимают много времени. В течение этого времени процесс ожидает ответ от ядра о результате операции, не делая ничего другого. Синхронный подход в случае файловых операций очень неэффективно утилизирует ресурсы.

Асинхронный код продолжает выполняться во время файловых операций. Код никогда не блокируется на I/O операциях, но может узнать об их завершении. Асинхронный код лучше утилизирует ресурсы компьютера, процессор не простояивает в ожидании операций ввода/вывода.

Важно вот ещё что: АФ выполняет операцию не сразу, поэтому у неё невозможен возврат результата выполнения асинхронной операции. Чтобы вернуть результат, ей нужна функция обратного вызова или просто **колбек** (callback). Она будет вызвана, когда операция закончится (в т.ч. с ошибкой).

У асинхронных функций свой стек, поэтому ошибки надо ловить в нём, а не во внешнем окружении. Поэтому АФ нельзя заключить в try-catch.

Завершение асинхронной операции

Тут возникает проблема. Когда запрос завершится в дополнительной ветви, как об этом узнает главная? Как вернуть полученное значение в основной поток, если это необходимо? Для этого существуют события и механизм обратного вызова.

Если запрос выполняется асинхронно, то он может оповестить всех желающих о своем окончании. Программа подписывается на это сообщение и регистрирует для него обработчик. Когда придет время, запрос создаст событие и уведомит подписчиков.

Обработчик продолжает выполнять последующий код, пока не получит сообщение. Тогда он прервется и обработает его.

Далее – в статье на [проглиб](#).

Ещё понятия

Когда речь заходит об асинхронности, всплывают еще три близких понятия. Это:

- конкурентность (concurrency)
- параллелизм (parallel execution)
- многопоточность (multithreading).

Все они связаны с одновременным выполнением задач, однако это не одно и то же.

Конкурентность

Понятие конкурентного исполнения самое общее. Оно буквально означает, что множество задач решаются в одно время. Можно сказать, что в программе есть несколько логических потоков – по одному на каждую задачу.

При этом потоки могут физически выполняться одновременно, но это не обязательно.

Задачи при этом не связаны друг с другом. Следовательно, не имеет значения, какая из них завершится раньше, а какая позже.

Используется для решения независимых задач, когда важно, чтобы выполнился хотя бы один запрос. Например, отправка идентичных запросов на разные сервера.

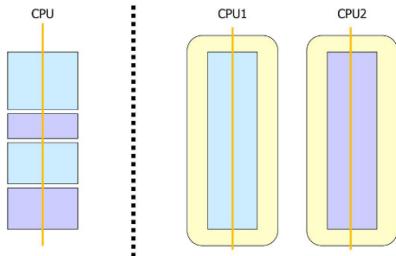
Параллелизм

Параллельное исполнение обычно используется для разделения одной задачи на части для ускорения вычислений.

Например, нужно сделать цветное изображение черно-белым. Обработка верхней половины не отличается от обработки нижней. Следовательно, можно разделить эту задачу на две части и раздать их разным потокам, чтобы ускорить выполнение в два раза.

Наличие двух физических потоков здесь принципиально важно, так как на компьютере с одним вычислительным устройством (процессорным ядром) такой прием провести невозможно.

Конкуренция Параллелизм



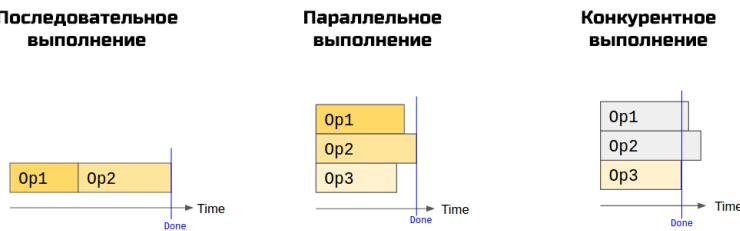
Многопоточность

Здесь поток является абстракцией, под которой может скрываться и отдельное ядро процессора, и тред ОС. Некоторые языки даже имеют собственные объекты потоков. Таким образом, эта концепция может иметь принципиально разную реализацию.

Асинхронность

Идея асинхронного выполнения заключается в том, что начало и конец одной операции происходят в разное время в разных частях кода. Чтобы получить результат, необходимо подождать, причем время ожидания непредсказуемо.

Можно выделить три самые популярные схемы асинхронных запросов.



Стек вызовов

Асинхронные функции тесно связаны со стеком вызовов, поэтому сначала кратко про него.

Стек (стопка) – абстрактный тип данных, представляющий собой список элементов, организованных по принципу «последним пришёл — первым вышел» (*LIFO, last in — first out*).

Возможны три операции со стеком: добавление элемента (проталкивание, *push*), удаление элемента (*pop*) и чтение головного элемента (*peek*).

Стек вызовов (call stack) – цепочки функций, вызывающих друг друга. Стек вызовов – это очередьность выполнения цепи запущенных программ.

Каждый внутренний вызов добавляет текущую функцию внутрь стека — и так до самой глубокой функции. Затем, когда происходит возврат, начинается раскрутка стека — из него по очереди (в обратном порядке, ведь это стек) извлекаются функции и продолжают своё выполнение с того места, где внутренняя функция вернула результат.

Пример стека вызовов – вывод ошибок. **Backtrace** (обратная трассировка) — не что иное, как стек вызовов, записанный в обратном порядке. Механизм исключений опирается на наличие стека вызовов. Возникающее исключение поднимается вверх по стеку вызовов до тех пор, пока не наткнется на конструкцию **try/catch** либо до тех пор пока стек вызовов не закончится.

```
const data = [16, 64, 4];
const data2 = data.map(Math.sqrt); // [4, 8, 2]
const predicate = (v) => unkown > 2;
const data3 = data2.filter(predicate); // ReferenceError
```

Результат вызова
index.js:3
const predicate = (v) => unkown > 2;
^
ReferenceError: unkown is not defined
at predicate (index.js:3:32) // указывает функцию
at Array.filter (<anonymous>) // функция выше
at Object.<anonymous> (index.js:4:21)

Вот так файлы читаются параллельно:

```
// глобальное состояние
const state = {
  count: 0, // сколько функций завершили чтение
  results: [], // здесь хранится прочитанная инфа из файлов
}

// функция записи нового файла
const tryWriteNewFile = () => {
  // guard expression
  // проверит, все ли функции завершили чтение
  if (state.count !== 2) {
    return;
  }
  // если всё готово, то начнётся запись
  fs.writeFile('./new-file', state.results.join('')), (error) => {
```

```

        if (error) {
            return;
        }
        console.log('finished!');
    });
}

// первая функция чтения
console.log('first reading was started');
fs.readFile('./first', 'utf-8', (error1, data1) => {
    console.log('first callback');
    if (error1) {
        return;
    }
    // сообщить о завершении чтения
    state.count += 1;
    // запись в определённый индекс
    state.results[0] = data1;
    // попробовать записать новый файл
    tryWriteNewFile();
});

// вторая функция чтения
console.log('second reading was started');
fs.readFile('./second', 'utf-8', (error2, data2) => {
    console.log('second callback');
    if (error2) {
        return;
    }
    state.count += 1;
    state.results[1] = data2;
    tryWriteNewFile();
});

```

Event Loop

Видео по теме [здесь](#).

Браузер работает по так называемой *событийной модели*. Он загружает страницу и ждёт действий от пользователя: клики, набор текста или движение мышкой. А код, загруженный на страницу, реагирует на эти события.

Такая организация взаимодействия невозможна в синхронном коде, у которого есть понятия "запуск" и "завершение" работы. Код в браузере не может завершиться совсем, он проходит стадию инициализации, а затем ждёт событий для реакции на них. Технически это выглядит как колбек, который соединён с определённым типом события. Когда событие срабатывает, то колбек вызывается.

Организация асинхронного взаимодействия требует наличия *событийного цикла* (Event Loop).

Событийный цикл всегда работает в однопоточном режиме (понимание этой темы кроется в операционных системах). Это значит, что события обрабатываются строго последовательно. Причём, запуск одного события может приводить к выполнению довольно тяжёлого кода, который работает достаточно долго. Такое периодически встречается, когда страницы сайта вдруг начинают подвисать. В какой момент начнёт обрабатываться следующее событие? В тот момент, когда текущий стек вызовов опустеет. Пока текущий стек не пуст, все остальные ждут его завершения. Исключение составляют [Workers](#), но это отдельная тема.

Hexlet Промисы

в модуле fs промисы доступны как свойство [promises](#).

Документация [здесь](#).

Можно сказать, что промис связывает асинхронный код и код, потребляющий результат асинхронного кода.

Промис – это не результат асинхронной операции. Это объект, который отслеживает выполнение операции. Операция по-прежнему асинхронна и выполнится когда-нибудь потом. С технической точки зрения, промис – это объект, который описывает состояние операции выполняющейся внутри него.

Интерфейс Promise представляет собой обертку для значения, неизвестного на момент создания промиса. Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается *обещание* (промис) получить результат в некоторый момент в будущем.

Что касается конструктора new Promise, то он нужен для того, чтобы оборачивать функции, которые работают на колбеках, потому что не все АФ имеют свои промисы-аналоги.

В конечном итоге конструкция new Promise() возвращает объект, с которым можно работать привычным для нас способом через методы .then и .catch.

В промис нельзя оборачивать больше 1-й асинхронной операции. Одна асинхронная операция — один конструктор new Promise.

Импорт:

```
var fs = require('fs');
const fsPromises = fs.promises;
```

Исполнитель выполняет задачу, затем вызывает resolve или reject, чтобы изменить состояние соответствующего Promise.

Когда пишешь executor, в resolve передаёшь value – вызывается в том случае, если функция завершилась успешно. Value будет её результатом, а в reject – error, это объект ошибки.

Вызывать reject можно с любым типом данных, но рекомендуется это делать с объектом Error.

Вот пример:

```
let fs = require('fs');

const a = new Promise(function(resolve, reject) {
  fs.readFile('test\\first.txt', 'utf-8', (err, data) => {
    if (err) {
      reject(err);
    }
    resolve(data);
  })
});

console.log(a)
// Promise { <pending> } - в ожидании
```

Вот пример с таймером. Спустя одну секунду «обработки» исполнитель вызовет resolve('выполнено'), чтобы передать результат:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('выполнено');
  }, 300);
});
```

Обычно исполнитель делает что-то асинхронное, но это не обязательно и resolve/reject могут быть вызваны сразу. Это может случиться, например, когда мы начали выполнять какую-то задачу, но тут же увидели, что ранее

её уже выполняли, и результат закеширован. Мы сразу получим успешно завершённый Promise без вычислений,. И результат не будет болтаться в стеке вызовов:

```
let promise = new Promise(function(resolve, reject) {  
    // задача, не требующая времени  
    resolve(123); // мгновенно вернёт результат: 123  
});
```

Встречаются задачи, когда асинхронного кода нет, но нужен промис чтобы начать цепочку промисов. Такой промис обычно создаётся так:

```
const promise = new Promise((resolve) => resolve());  
// promise.then ...
```

Тоже самое для промиса, который завершается неуспешно:

```
const promise = new Promise((resolve, reject) => reject());  
// promise.catch ...
```

Для этих задач добавили специальные сокращения, с которыми код становится чище:

```
const promise1 = Promise.resolve();  
// promise1.then  
  
const promise2 = Promise.reject();  
// promise2.catch ...
```

Далее будут использоваться примеры с «**fs.promises API**», у которого есть свои готовые промисы. Как было сказано ранее, не все у всех коллбеков есть аналогичный интерфейс на промисах, поэтому их нужно запихивать в конструктор new Promise.

Обработчики: `then`, `catch`, `finally`

Обработчики ждут завершения промиса и работают со свойством `.result` возвращаемого объекта.

`.then`

Метод `.then` первым параметром принимает функцию-обработчик, в которую передаются только данные из АФ при успешном её завершении, а вторым параметром – функцию, которая обрабатывает ошибки.

Синтаксис:

```
promise.then(  
    function(result) { обработает успешное выполнение },  
    function(error) { обработает ошибку }  
);
```

Первый аргумент метода – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

Если нужен только результат успешного выполнения задачи, то можно передать только одну, первую функцию.

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента:

`.then(null, errorHandlingFunction)`.

.then тоже возвращает ещё один промис, а не результат выполнения асинхронки. Поэтому к нему можно сразу же применять ещё один .then. Данные из предыдущего колбека .then пойдут в новый .then. Их можно соединять друг с другом и делать цепочки.

Даже если в then передаётся синхронная функция, .then всё равно вернёт промис:

```
let fs = require('fs');
let fsPromises = fs.promises;

// тут then вернёт Number, не связанный с чтением файла
let a = fsPromises.readFile('Test\\first.txt', 'utf-8')
.then(() => 123);

console.log(a);
// Promise { <pending> }

console.log(typeof(a));
// object
```

Синхронные функции, которые используют промисы, автоматически сами становятся промисом и поэтому должны возвращать наружу промис. Например, функция «copy» делает возврат промиса. Только в этом случае вызывающий код сможет встроить эту функцию и контролировать ход выполнения асинхронной операции:

```
const copy = (src, dest) => {
  return fsPromises.readFile(src, 'utf-8')
    .then((content) => fs.writeFile(dest, content));
};
```

Пример работы .then

```
let fs = require('fs');

const a = new Promise(function(resolve, reject) {
  fs.readFile('test\\first.txt', 'utf-8', (err, data) => {
    if (err) {
      reject(err);
    }
    resolve(data);
  })
});

console.log(a)
// Promise { <pending> } - в ожидании

a.then((item) => console.log(item));
// This is text from first file

a.then(() => console.log(a))
// Promise { 'This is text from first file\r\n' }
```

Пример с таймером:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});
```

```
// resolve запустит первую функцию, переданную в .then
promise.then(
  result => console.log(result),
  error => console.log(error)
);
```

.catch

Тоже возвращает промис, поэтому его можно встраивать в цепочки и комбинировать с then. Вызов .catch(f) – это сокращённый, «укороченный» вариант .then(null, f).

```
let a = new Promise((resolve, error) => {
  throw new Error('!!!!'); // я его тут не передал в error даже
});

a.catch((err) => console.log(err));
// Error: !!!!

a.then(
  (data) => console.log(data),
  (err) => console.log(err)
);
// Error: !!!
```

В случае с использованием библиотеки fs.Promises всё то же самое:

```
const promise = fsPromises.readFile('unkownfile');

promise.catch((e) => console.log('error!!!', e));

// => error!!! { описание ошибки }
```

В большинстве ситуаций не имеет значения, на какой из операций возникла ошибка. Любое "падение" должно прерывать текущее выполнение и уходить в блок обработки ошибки. Если возникла ошибка, то она передаётся по цепочке первому встреченному catch, а все встречающиеся на пути then игнорируются.

Поэтому первый код показывает возможность чередования, а второй – то же самое, только упрощение первого:

```
const promise = fsPromises.readFile('unkownfile')
  .catch(console.log)
  .then(() => fsPromises.readFile('anotherUnknownFile'))
  .catch(console.log); // можно прям так бросать данные в функцию
```

Достаточно одного .catch в конце. Всё равно в случае ошибки все then будут игнорироваться

```
const promise = fsPromises.readFile('unkownfile')
  .then(() => fsPromises.readFile('anotherUnknownFile'))
  .catch(console.log);
```

Семантически эти версии кода не эквивалентны. В первом случае вторая операция чтения начнёт выполняться обязательно, независимо от того, как закончилась предыдущая. В последнем – если упадёт первое чтение файла, то второе не будет выполнено.

Получается, что пока не будет найден **ближайший** .catch, все .then выключаются. А после catch - включаются. Промис продолжает выполнять то, что было добавлено в then уже после catch.

Иногда ошибку нужно генерировать самостоятельно. Самый простой способ сделать это — бросить исключение. Промис сам их преобразует, как надо, и отправит по цепочке в поиске вызова catch. Этую технику можно использовать для того, чтобы выключать и обходить определённые звенья then из цепи. То же самое можно делать с reject, о котором говорилось в самом начале.

В этом примере асинхронная функция touch проверяет, существует ли файл. Если нет – создает его.

fs.access - проверка существования файла

```
import { touch } from './file.js';

export const touch = (filepath) => fsPromises.access(filepath)
  .catch(() => fsPromises.writeFile(filepath));

touch('/myfile').then(() => console.log('created!'));
```

Пример с таймером:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});

// .catch(f) это тоже самое, что promise.then(null, f)
promise.catch(console.log);

// Error: Ошибка!
```

.finally

По аналогии с блоком try-catch, у промисов также есть метод finally.

Он выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

.finally хорошо подходит для очистки, например остановки индикатора загрузки, его ведь нужно остановить вне зависимости от результата.

Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам. Т.е. .finaly ничего не делает с результатом промиса, он может сделать только какие-то вспомогательные вещи, не трогая промис. Это очень удобно, потому что finally не предназначен для обработки результата промиса.

```
new Promise((resolve, reject) => {
  /* сделать что-то, что займет время, и после вызвать resolve/reject */
})

// выполнится, когда промис завершится, независимо от того, успешно или нет
.finally(() => остановить индикатор загрузки)
.then(result => показать результат, err => показать ошибку)
```

Использование .reduce

Если кол-во файлов заранее неизвестно, то нужно применять reduce и в качестве аккумулятора поставить `Promise.resolve()`. Метод возвращает Promise выполненный с переданным значением.

Документация по нему [здесь](#).

```
const filePaths = /* список путей до файлов */;

// В then отдается функция, а не ее вызов!
const promise = filePaths.reduce((acc, path) => (
    acc.then(/* содержимое предыдущего прочитанного файла */) => fs.readFile(path))
), Promise.resolve\(\));

// Если надо, продолжаем обработку
promise.then...
```

</>

Сетевые запросы

Для сетевых запросов используется термин AJAX (Asynchronous JavaScript And XML). Это старое понятие, здесь есть указание на XML, это один из способов сделать запрос.

Сетевые запросы можно сделать с помощью:

- [Fetch](#)
- [XML](#)

Fetch

[Fetch API](#) предоставляет интерфейс для получения ресурсов (в том числе по сети). Fetch обеспечивает обобщённое определение объектов [Request](#) и [Response](#).

```
let promise = fetch(url, [{options}]);
```

`url` Страна с прямым указанием адреса или Request объект (объект ответа).
`options` Объект с опциями, которые нужно применить к запросу.

Некоторые опции

`method`: метод запроса (GET, POST и т.д.).
`headers`: заголовки, содержащиеся в объекте.
`body`: тело запроса, может быть: Blob, BufferSource, FormData, URLSearchParams, или USVString объектами. GET или HEAD запрос не может иметь тела.

Обёртка для запросов

```
const getResource = async (url) => {
    const response = await fetch(url);

    if (!response.ok) {
        throw new Error(`Could not fetch ${url}, received ${res.status}`);
    }

    const body = await response.json();
    return body;
};
```

Типичная реализация

```
let response = await fetch(url, options); // завершается с заголовками ответа
let result = await response.json(); // читать тело ответа в формате JSON
```

```
fetch(url, options)
  .then(response => response.json())
  .then(result => /* обрабатываем результат */)
```

Принцип работы и методы

1. `fetch(url)` возвращает промис.

Работать с ним надо с помощью методов `.then` или `async-await`. Когда промис успешно выполнен, он возвращает объект `response`.

Промис завершается с ошибкой, если `fetch` не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 не являются ошибкой.

2. Объект `response`.

`Response` – это самостоятельный API с набором методов.

Сначала объект получает только заголовки и статус запроса (тело ответа приходит потом).

Можно применить следующие методы:

<code>Response</code>	API в MDN
<code>Response.ok</code>	<code>true/false</code> , выполнился ли запрос успешно (коды ответа 200–299).
<code>Response.status</code>	код ответа.
<code>Response.headers</code>	объект <code>Headers</code> , который содержит заголовки ответа (у него свои методы).

Headers API

`Headers()`

Creates a new `Headers` object.

`Headers.get()`

Returns a `ByteString` sequence of all the values of a header within a `Headers` object with a given name.

`Headers.entries()`

Returns an iterator allowing to go through all key/value pairs contained in this object.

`Headers.has()`

Returns a boolean stating whether a `Headers` object contains a certain header.

`Headers.keys()`

Returns an iterator allowing you to go through all keys of the key/value pairs contained in this object.

```
response.headers.get('Content-Type')
```

```
for (let [key, value] of response.headers) {
  alert(` ${key} = ${value}`);
}
```

Чтобы работать с телом ответа, нужно применить к объекту `response` методы из интерфейса `Body`.

В результате будет возвращён ещё один промис, с содержимым которого уже можно работать.

Можно выбрать только один метод чтения ответа.

<code>Body</code>	API в MDN предоставляет методы, относящиеся к телу запроса/ответа.
<code>Body.json()</code>	Декодирует ответ в формате JSON.
<code>Body.text()</code>	Читает ответ и возвращает как обычный текст.
<code>Body.blob()</code>	Возвращает объект как <code>Blob</code> (бинарные данные с типом).

`Body.arrayBuffer()`

Возвращает ответ как `ArrayBuffer` (низкоуровневое представление данных).

`Body.formData()`

Takes a `Response` stream and reads it to completion. It returns a promise that resolves with a `FormData` object.

`Body.body`

A simple getter used to expose a `ReadableStream` (en-US) of the body contents. Это объект `ReadableStream`, с помощью которого можно считывать тело запроса по частям.

Опции запроса

Для любых запросов, кроме GET, нужно использовать опции запроса.

Про опции можно прочитать в документации про fetch [здесь](#).

```
let promise = fetch(url, [{options}]);
```

Некоторые опции

method	метод запроса (GET, POST и т.д.)
headers	заголовки, содержащиеся в объекте.
body	тело запроса: Json, FormData, Blob/BufferSource и другие. GET или HEAD запрос не может иметь тела.
mode	режим, например, cors, no-cors или same-origin.

Headers

Для установки заголовка запроса нужно использовать опцию headers. Её значение – это объект с определёнными ключами и значениями.

Посмотреть возможные HTTP-заголовки этого объекта можно в Документации [здесь](#).

```
const options = {
  headers: {
    Authentication: 'secret'
    'Content-Type': 'application/json; charset=utf-8'
  },
};

const response = fetch(url, options);
```

Content-Type

Строка для определения типа ресурса. Список таких типов [здесь](#).

Есть [список](#) запрещённых HTTP-заголовков, которые нельзя установить. Они обеспечивают корректную работу протокола HTTP, поэтому контролируются только браузером.

Body

данные для отправки (тело запроса) в виде текста, FormData, BufferSource, Blob или UrlSearchParams

Другие опции

Объект с опциями, содержащий пользовательские настройки запроса, может содержать:

credentials:

Полномочия: omit, same-origin или include. Для автоматической отправки куки для текущего домена, эта опция должна быть указана. Начиная с Chrome 50, это свойство также принимает экземпляр класса [FederatedCredential \(en-US\)](#) или [PasswordCredential \(en-US\)](#).

cache:

Режим кеширования запроса default, no-store, reload, no-cache, force-cache или only-if-cached.

redirect:

Режим редиректа: follow (автоматически переадресовывать), error (прерывать перенаправление ошибкой) или manual (управлять перенаправлениями вручную). В Chrome по дефолту стоит follow (ранее, в Chrome 47, стояло manual).

referrer:

[USVString](#), определяющая no-referrer, client или a URL. Дефолтное значение - client.

referrerPolicy:

Определяет значение HTTP заголовка реферера. Может быть: no-referrer, no-referrer-when-downgrade, origin, origin-when-cross-origin, unsafe-url.

integrity:

Содержит значение целостности субресурсов ([subresource integrity](#)) запроса (например, sha256-BpfBw7ivV8q2jLit13fxDYAe2tJllusRSZ273h2nFSE=).

keepalive:

Эта опция может быть использована, чтобы разрешить запросу "пережить" страницу. Получение ресурсов с флагом keepalive - это альтернатива [Navigator.sendBeacon\(\)](#) API.

signal:

Экземпляр объекта [AbortSignal](#); позволяет коммуницировать с fetch запросом и, если нужно, отменять его с помощью [AbortController](#).

POST-запросы

```
const options = {
  method: 'post',
  body: json // это указание на переменную
  headers: { 'Content-Type': 'application/json; charset=utf-8' },
};

const promise = fetch(url, options);
```

body

Данные, которые надо отправить

headers

Сообщить браузеру, как правильно интерпритировать данные. В примере указано, что это - json.

[Content-Type](#)

Строка для определения типа ресурса. Список таких типов [здесь](#).

По умолчанию будет text/plain; charset=UTF-8.

Отправка изображения

Пропустил это, потому что не шарю в Blob.

Примеры

Запрос и работа с заголовками

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => console.log(response.headers));
```

Запрос тела ответа и его декодирование в формате json

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => console.log(commits[0].author.login))
};
```

Обработать сразу несколько адресов + обработать ошибки

Если возникает какая-то ошибка - возвратить null

```
async function getUsers(names) {
```

```

// перегнать все имена в промисы-запросы
let requests = names.map(el => fetch(`https://api.github.com/users/${el}`))
  .then(
    // correctResponse
    response => {
      if (response.status !== 200) return null;
      return response.json();
    },
    // errorResponse
    error => null)
  );

// дождаться выполнения всех запросов и их обработки
let result = await Promise.all(requests);
return result;
}
console.log(getUsers(['iliakan', 'remy', 'no.such.users']));

```

Чтение и перебор заголовков

```

(async () => {
  let response = await fetch('https://api.github.com/repos/javascript-
tutorial/en.javascript.info/commits');
  // чтение одного заголовка
  console.log(response.headers.get('Content-Type'));
  // перебрать все заголовки
  for (let [key, value] of response.headers) {
    console.log(`${key} = ${value}`);
  }
})();

```

FormData

`FormData` – это API, которое автоматом создаёт объект из полей формы.

Объекты `FormData` позволяют конструировать наборы пар ключ-значение, представляющие поля формы и их значения, которые в дальнейшем можно отправить с помощью метода `send()`.

Объект будет соответствующим образом закодирован и отправлен с заголовком `Content-Type: form/multipart`. Будет такой же формат на выходе, как если бы отправлялась обыкновенная форма с `encoding` установленным в `"multipart/form-data"`.

Конструктор

```

let formData = new FormData(form);

const options = {
  method: 'POST',
  body: new FormData(formElem)
}

```

Методы `FormData`

[FormData в MDN](#)

`FormData.append(name, value)`
`formData.append(name, blob, fileName)`
Создать или обновить пару ключ-значение.

`FormData.delete(name)`

удаляет поле.

FormData.get(name)

Возвращает значение поля с именем name.

FormData.getAll(name)

Возвращает массив всех значений ассоциированных с переданным ключом.

FormData.has(name)

true / false

FormData.set()

formData.set(name, blob, fileName)

Как append, но сначала удаляет все уже имеющиеся поля с ключом name.

Можно перебирать в цикле:

```
for(let [name, value] of formData) {...}
```

FormData.keys()

Возвращает iterator , который позволяет пройтись по всем ключам для каждой пары "ключ-значение" , содержащимся внутри объекта FormData

FormData.entries()

Возвращает iterator который позволяет пройтись по всем парам "ключ-значение" , содержащимся внутри объекта FormData

FormData.values()

Возвращает iterator , который позволяет пройтись по всем значениям , содержащимся в объекте FormData

Технически форма может иметь много полей с одним и тем же именем name, поэтому несколько вызовов append добавят несколько полей с одинаковыми именами.

Примеры

Отправка простой формы

В этом примере серверный код не представлен. Он принимает POST-запрос с данными формы и отвечает сообщением.

```
<form id="form-elem" />

formElem.onsubmit = async (e) => {
  e.preventDefault();

  const options = {
    method: 'POST',
    body: new FormData(formElem)
  }

  let response = await fetch(url, options);
  let result = await response.json();

  console.log(result.message);
};
```

Отправка формы с файлом

Файл удобнее всего отправлять вместе с формой. В этом примере ничего не меняется.

```
<form id="form-elem">
  <input type="file" name="picture" accept="image/*">
</form>

formElem.onsubmit = async (e) => {
  e.preventDefault();
```

```
const options = {
  method: 'POST',
  body: new FormData(formElem)
}

let response = await fetch(url, options);
let result = await response.json();

console.log(result.message);
};
```

Отправка формы с Blob-данными

Кликай на ссылку.

Обратите внимание на то, как добавляется изображение Blob:

```
formData.append("image", imageBlob, "image.png");
```

Это как если бы в форме был input type="file" со значениями:
name = "image"
наименование файла – image.png
imageBlob – сам файл.

Сервер прочитает и данные и файл, точно так же, как если бы это была обычная отправка формы.

Fetch: ход загрузки

Статья [тут](#), не читал.

Fetch: прерывание запроса

Не читал

Fetch: запросы на другие сайты

Статья [тут](#).

SWAPI

Получение данных

Чтобы получить данные с сервера, надо выполнить два вызова:

```
const getResource = async(url) => {
  const res = await fetch(url);
  const body = await res.json();
  return body;
}

getResource('https://swapi.dev/api/people/1')
  .then((body) => {
    console.log(body);
 });
```

Обработка ошибок

Есть 2 типа ошибок: сетевые ошибки и ошибки, которые сообщает сервер.

Сетевые происходят, когда мы вызываем сервер, но сервер не ответил. Возможно, проблема с интернетом или сервер недоступен (сломался, лёг).

В этом случае промис в первой строке (в примере выше) будет rejected.

```
const getResource = async(url) => {
  const res = await fetch(url);
  const body = await res.json();
  return body;
}

getResource('https://swapi.dev/api/people/1404')
  .then((body) => {
    console.log(body);
  })
  .catch((err) => {
    console.error('Could not fetch', err);
  });
}
```

Ошибки, которые сообщает сервер.

Если персонаж с указанным id не был найден, сервер должен вернуть 404.

В этом случае блок catch из примера выше не сработает. Логика fetch очень простая: мы вызвали сервер, получили ответ 404, но при этом сервер вернул валидный ответ. С т.з. fetch его функция выполнена отлично: мы послали запрос и мы получили ответ.

А вот интерпретация ответа – это вопрос клиентского кода. Очевидно, что надо сделать так, чтобы ошибочные коды сервера давали такой же результат, как и ошибка сети. Чтобы написать такой код, надо проверить статус ответа.

```
const getResource = async(url) => {
  const res = await fetch(url);
  if (!res.ok) {
    throw new Error(`Could not fetch ${url}, received ${res.status}`);
  }
  const body = await res.json();
  return body;
}

getResource('https://swapi.dev/api/people/1404')
  .then((body) => {
    console.log(body);
  })
  .catch((err) => {
    console.error('Could not fetch', err);
  });
}
```

Клиент для API

Классы-сервисы нужны для того, чтобы инкапсулировать весь сетевой код и изолировать его от остальных частей приложения в отдельный класс-сервис. Это хорошая практика при разработке больших приложений, потому что весь код работы с данными находится в одном месте и в будущем можно реализовать что-нибудь типа кеширования, сменить источник данных или реализовать дополнительную логику фильтрации.

Для остальной части приложения этот класс-сервис – это просто асинхронный источник данных. Компоненты не должны знать, откуда именно берутся данные.

Это та часть сетевого адреса, которая присутствует во всех запросах. Чтобы её каждый раз не дублировать, достаточно сохранить её в приватном поле и приклеивать к передаваемому url.

async getAllPeople()

Метод автоматом вычленяет массив персонажей и возвращает его. Если не поставить `async`, то пришлось бы использовать промисы вне класса при использовании этого метода.

Строчка `res.results` – это свойство возвращаемого сервером объекта, в котором хранится массив объектов-персонажей.

В примере ниже будет распечатываться список имён персонажей:

```
export default class SwapiService {

  _apiBase = 'https://swapi.dev/api';

  async getResource(url) {
    const res = await fetch(`.${this._apiBase}${url}`);
    if (!res.ok) {
      throw new Error(`Could not fetch ${url}, received ${res.status}`);
    }
    const body = await res.json();
    return body;
  }

  async getAllPeople() {
    const res = await this.getResource('/people/');
    return res.results;
  }
  async getPerson(id) {
    const res = await this.getResource(`/people/${id}/`);
    return res;
  }

  async getAllPlanets() {
    const res = await this.getResource('/planets/');
    return res;
  }
  async getPlanet(id) {
    const res = await this.getResource(`/planets/${id}`);
    return res;
  }

  async getAllStarships() {
    const res = await this.getResource('/starships/');
    return res;
  }
  async getStarship(id) {
    const res = await this.getResource(`/starships/${id}`);
    return res;
  }

};

const swapi = new SwapiService();

swapi.getAllPeople().then((people) => {
  people.forEach((p) => console.log(p.name));
});

swapi.getPerson(3).then((p) => {
```

```
    console.log(p.name);
});
```

Трансформация данных

API не всегда передаёт данные в нужном формате. Например, вместо camelCase может использоваться нижнее подчёркивание, или приложению нужно использовать только несколько свойств, а не все свойства, которые передаются с сервера.

В этом случае, в клиенте можно трансформировать данные, полученные с сервера, чтобы передавать их компонентам приложения в нужном формате. Для этого надо дописать ещё несколько приватных функций.

Такие трансформации нельзя делать в одной функции, которая получает данные от API, потому что надо отделять модель данных от API и модель данных приложения. Такая практика применяется для крупных проектов со сложными моделями данных, которые могут изменяться.

Swapi не передаёт id, поэтому его надо вытащить из свойства url через регулярное выражение.

```
_extractId(item) {
  const idRegExp = /\//([0-9]*)\$/;
  return item.url.match(idRegExp)[1];
}

_transformPlanet = (planet) => {
  return {
    id: this._extractId(planet),
    name: planet.name,
    population: planet.population,
    rotationPeriod: planet.rotation_Period,
    diameter: planet.diameter
  }
}
```

Итоговый вариант класса:

```
export default class SwapiService {

  _apiBase = 'https://swapi.dev/api';

  async getResource(url) {
    const res = await fetch(`.${this._apiBase}${url}`);
    if (!res.ok) {
      throw new Error(`Could not fetch ${url}, received ${res.status}`);
    }
    const body = await res.json();
    return body;
  }

  async getAllPeople() {
    const res = await this.getResource('/people/');
    return res.results.map(this._transformPerson);
  }

  async getPerson(id) {
    const person = await this.getResource(`/people/${id}/`);
    return this._transformPerson(person);
  }

  async getAllPlanets() {
```

```

    const res = await this.getResource('/planets/');
    return res.results.map(this._transformPlanet);
}
async getPlanet(id) {
    const planet = await this.getResource(`/planets/${id}`);
    return this._transformPlanet(planet);
}

async getAllStarships() {
    const res = await this.getResource(`/starships/`);
    return res.results.map(this._transformStarship);
}
async getStarship(id) {
    const starship = await this.getResource(`/starships/${id}`);
    return this._transformStarship(starship);
}

_extractorId(item) {
    const idRegExp = `/([0-9]*)$/`;
    return item.url.match(idRegExp)[1];
}

_transformPlanet = (planet) => {
    return {
        id: this._extractId(planet),
        name: planet.name,
        population: planet.population,
        rotationPeriod: planet.rotation_Period,
        diameter: planet.diameter
    }
}

_transformStarship= (starship) => {
    return {
        id: this._extractId(starship),
        name: starship.name,
        model: starship.model,
        manufacturer: starship.manufacturer,
        costInCredits: starship.costInCredits,
        length: starship.length,
        crew: starship.crew,
        passengers: starship.passengers,
        cargoCapacity: starship.cargoCapacity
    }
}

_transformPerson = (person) => {
    return {
        id: this._extractId(person),
        name: person.name,
        gender: person.gender,
        birthYear: person.birthYear,
        eyeColor: person.eyeColor
    }
}
};


```

Поскольку сервис возвращает готовый объект с нужными полями, то в state можно передавать информацию так:

```

state = {
    planet: {}
}
```

```

};

componentDidMount() {
  this.updatePlanet();
}

updatePlanet() {
  const id = Math.floor(Math.random()*21) + 1;
  this.swapiService.getPlanet(id)
    .then(this.onPlanetLoaded);
}

onPlanetLoaded = (planet) => {
  this.setState({planet})
}

```

Спиннер

Скачать спиннер можно тут:

[loading.io](#)

Готовые индикаторы загрузки. Мой любимый [тут](#).

Простой вариант

Для того, чтобы спиннер работал только когда компонент ждёт данные с сервера:

В state компонента, который подгружает данные, добавить ещё одно поле: loading. Как только компонент инициализируется, будет показан спиннер, потому что он только обращается к данным.

Первый способ:

в функции Render прописать условие показа спиннера: если loading – true, то возвращать элемент спиннер. Проблема – спиннер будет заменять собой весь элемент.

Второй способ:

чтобы спиннер отображался внутри основного элемента, нужно все дочерние элементы, которые ожидают данные с сервера, убрать в отдельный приватный фрагмент.

В JSX если значение какого-либо элемента – null, то он не отображается. Вместо if-else можно использовать тернарный оператор для спиннера.

В функции, которая срабатывает при получении данных сервера, поменять true на false одновременно с разбросом этих данных.

```

state = {
  planet: {},
  loading: true
};

onPlanetLoaded = (planet) => {
  this.setState({
    planet,
    loading: false
  });
}

render() {
  const { planet, loading } = this.state;

  const spinner = loading ? <Spinner /> : null;
  const content = !loading ? <PlanetView planet={planet} /> : null;

```

```

return (
  <div className="random-planet jumbotron rounded">
    {spinner}
    {content}
  </div>
);
}

const PlanetView = ({ serverData }) => {
  return (
    <React.Fragment>
      <img>
        <div>
      </React.Fragment>
    );
}

```

Обработка ошибок

Чтобы из-за неудачного запроса приложение не падало целиком, надо добавить блок .catch в компонент. Дополнительно надо создать компонент, который будет заниматься отображением ошибки.

Компонент-ошибка:

```

import icon from './death-star.png';

const ErrorIndicator = () => {
  return (
    <div className='error-indicator'>
      <img src={icon} alt='error icon'/>
      <span className='boom'>BOOM!</span>
      <span>
        something has gone terribly wrong
      </span>
      <span>
        (but we already sent droids to fix it)
      </span>
    </div>
  );
}

```

В state компонента добавляем поле error, которое по умолчанию false.

Добавляем функцию onError, которая занимается изменением state и помещается в catch.

hasData – данные будут отображаться, когда нет ни ошибки, ни загрузки.

```

onPlanetLoaded = (planet) => {
  this.setState({
    planet,
    loading: false,
    error: false
  });
}

onError = (err) => {
  this.setState({
    error: true,
    loading: false
  });
}

updatePlanet() {

```

```

const id = Math.floor(Math.random()*21) + 1;
this.swapiService.getPlanet(id)
  .then(this.onPlanetLoaded)
  .catch(this.onError);
}

render() {
  const { planet, loading, error } = this.state;

  const hasData = !(loading || error);

  const errorMessage = error ? <ErrorIndicator /> : null;
  const spinner = loading ? <Spinner /> : null;
  const content = hasData ? <PlanetView planet={planet} /> : null;

  return (
    <div className="random-planet jumbotron rounded">
      {errorMessage}
      {spinner}
      {content}
    </div>
  );
}
}

```

Все методы жизненного цикла

Mounting

Когда срабатывает

Компонент создаётся и первый раз отображается на странице.

Если этот метод вызван, значит элементы уже гарантированно находятся на странице.

Что делать (не использовать для этого всего конструктор):

- проводить инициализацию компонента,
- делать сетевые запросы,
- получать данные в компонент,
- работать с DOM.

Что вызывается:

`constructor() => render() => componentDidMount()`

Updates

Когда срабатывает

Компонент получает обновления, т.е. это либо:

- пришли новые свойства
- изменился state (вызван setState)

При этом функция срабатывает после того, как state обновлён.

Что делать:

Запрашивать новые данные для обновлённых свойств.

Что вызывается:

```

render() => componentDidUpdate(prevProps, prevState)
  if (this.props.personId !== prevProps.personId) {
    this.updatePerson();
  }
}

```

Unmounting

Когда срабатывает

Во время удаления компонента со страницы . В момент вызова DOM-элемент всё ещё будет находиться на странице.

Что делать:

- очищать те ресурсы, с которыми работал компонент

- останавливать запущенные таймеры
- останавливать запросы к серверу

Что вызывается:

`componentWillUnmount()`

Error

В компоненте произошла ошибка, которая не была поймана раньше.

Работает с ошибками жизненного цикла и метода rendering.

Это не замена try-catch.

Что делать:

Отключить ветку, в которой произошла ошибка.

Что вызывается:

`componentDidCatch(error, info)`

error – типичная ошибка JS

info – детали ошибки, характерные для React

`componentDidMount()`

Получение данных для itemList

В State хранится список предметов, который будет запрошен через сеть. По умолчанию этот список – null.

В render этот список передаётся из state.

В Render есть условие: если список – null (т.е. ещё не получен по сети), то отобразить spinner.

`renderItems(arr)`

Метод, который получает массив персонажей (от сетевого запроса) и через map делает из них li-элементы.

`onItemSelected(id)`

Функция определена в App передаётся в ItemList через props.

```
export default class ItemList extends Component {  
  
  swapiService = new SwapiService();  
  
  state = {  
    peopleList: null  
  };  
  
  componentDidMount() {  
    this.swapiService.getAllPeople()  
      .then((peopleList) => {  
        this.setState({  
          peopleList  
        });  
      });  
  }  
  
  renderItems(arr) {  
    // return arr.map((person) => {  
    return arr.map(({id, name}) => {  
      return (  
        <li  
          className="list-group-item"  
          key={id}  
          onClick={() => this.props.onItemSelected(id)}  
        >  
          {name}  
        </li>  
      );  
    });  
  }  
}
```

```

render() {
  const { peopleList } = this.state;

  if (!peopleList) {
    return <Spinner />
  }

  const items = this.renderItems(peopleList);

  return (
    <ul className="item-list list-group">
      {items}
    </ul>
  );
}
}

```

componentDidUpdate()

Компонент PersonDetails должен обновляться каждый раз при выборе персонажа из списка слева.

В State записывается текущий выбранный персонаж. По умолчанию – null. Текущий персонаж передаётся из App. В App он хранится в State.

updatePerson()

Функция, которая занимается сетевым запросом и отрисовкой данных. Также перезаписывает state: поле person, которое обновляется после получения данных с сервера, и поле loading.

componentDidUpdate

Если этот метод будет менять что-то, что в конечном счёте повлечёт обновление props компонента (в моём случае props компонента обновляется, потому что props получает информацию из App State), то обязательно внутри должно быть условие на проверку предыдущего props и нового. Иначе обновления будут запускаться по кругу бесконечно.

```

export default class PersonDetails extends Component {

  swapiService = new SwapiService();

  state = {
    person: null,
    loading: false
  }

  componentDidMount() {
    this.updatePerson();
  }

  componentDidUpdate(prevProps) {
    if (this.props.personId !== prevProps.personId) {
      this.updatePerson();
    }
  }

  updatePerson() {
    const { personId } = this.props;
    if (!personId) {
      return;
    }

    this.setState({
      loading: true
    })
  }
}

```

```

    });

    this.swapiService.getPerson(personId)
      .then((person) => {
        this.setState({ person, loading: false });
      });
  }

render() {
  const { person, loading } = this.state;

  const selectMessage = !person ? <span>Select a person from a list</span> : null;
  const spinner = loading ? <Spinner /> : null;
  const personData = (!loading && person) ? PersonView(person) : null;

  return (
    <div className="person-details card">
      {selectMessage}
      {spinner}
      {personData}
    </div>
  );
}

const PersonView = (props) => {
  const { id, name, gender, birthYear, eyeColor } = props;

  return (
    <React.Fragment>
      <img className="person-image"
        src={`https://starwars-visualguide.com/assets/img/characters/${id}.jpg`}
        alt="character"/>

      <div className="card-body">
        <h4>{name}</h4>

        <ul className="list-group list-group-flush">
          <li className="list-group-item">
            <span className="term">Gender</span>
            <span>{gender}</span>
          </li>
          <li className="list-group-item">
            <span className="term">Birth Year</span>
            <span>{birthYear}</span>
          </li>
          <li className="list-group-item">
            <span className="term">Eye Color</span>
            <span>{eyeColor}</span>
          </li>
        </ul>

      </div>
    </React.Fragment>
  )
}

```

componentWillUnmount()

Вызывается перед тем, как компонент удалится (компонент всё ещё находится в дом-дереве). Используется для того, чтобы очистить ресурсы, с которыми работал компонент.

Например, если был запущен таймер и его надо отменить. Можно отменять текущие запросы к серверу или отписываться от веб-сокетов.

Если компонент работает со сторонней библиотекой и этой библиотеке тоже надо сделать очистку, то этот метод – самое лучшее место, чтобы написать такой код.

```
componentDidMount() {
  this.updatePlanet();
  this.interval = setInterval(() => this.updatePlanet(), 5000);
}

componentWillUnmount() {
  clearInterval(this.interval);
}
```

componentDidCatch()

Отлавливает ошибки из функций, которые отвечают за корректный рендеринг компонента.

Если в одном из компонентов на странице выбросить ошибку, то всё приложение ляжет: будет показан пустой экран. В консоле будет виден код ошибки, но пользователь ничего не поймёт.

React даёт возможность обрабатывать ошибки и ограничивать их область действия.

В случае, если ошибка произойдёт, то отключится только вышестоящий по иерархии компонент, в котором был метод componentDidCatch, а всё остальное будет работать.

У метода есть 2 аргумента: (error, info)

В state компонента добавляется новый флаг hasError: false, который говорит, была ли ошибка или нет. А когда componentDidCatch – true.

```
state = {
  showRandomPlanet: true,
  selectedPerson: null,
  hasError: false
};

componentDidCatch() {
  console.log('componentDidCatch()');
  this.setState({ hasError: true });
}

render() {
  if (this.state.hasError) {
    return <ErrorIndicator />
  }
}
```

Выводы

1. React ничего не знает о работе с сервером, это задача других библиотек.

2. Сетевой код следует изолировать от кода компонентов
3. Если необходимо, трансформировать данные надо до того, как их получит компонент
4. Приложение должно уметь обрабатывать состояния «загрузка» и «ошибка»
5. Разделять ответственность компонентов: логику и рендеринг

</>

React

React

JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов. И это именно библиотека для создания UI (в смысле, инструменты для создания, а не готовые компоненты), а не фреймворк.

JSX

JavaScript XML (JSX) — это расширение синтаксиса JavaScript, которое позволяет использовать HTML-подобный синтаксис для описания структуры интерфейса. В курсе сказано, что он позволяет комбинировать JS с разметкой. Таким образом, код и UI находятся рядом и это помогает описывать логику.

Создание проекта

Установка приложения из репозитория

```
npm install -g create-react-app  
Установить приложение для использования React
```

```
npx create-react-app todo  
Создать react-проект  
Будет создан каталог с react-проектом, в него скачиваются все необходимые пакеты.
```

Команды после создания проекта

```
npm start      Запустить компиляцию, открыть проект в браузере, запустится локальный сервер  
npm run build Bundles the app into static files for production.  
npm test       Starts the test runner.
```

Ограничение на установку скриптов

Если команды выше не работают, надо отключить ограничение на установку скриптов

```
Set-ExecutionPolicy RemoteSigned -Scope CurrentUser    Отключить  
Set-ExecutionPolicy Restricted -Scope CurrentUser     Вернуть назад
```

Подключение библиотек

```
import React from 'react';  
import ReactDOM from 'react-dom';  
  
ReactDOM.render( element, container, [callback] )
```

element элемент для отрисовки
container где отрисовывать элемент

React

Библиотека, которая говорит трансплаеру, как перегнать синтаксис JSX в обычный JS-код.

ReactDOM

Библиотека, которая может преобразовать virtual dom в реальный дом. Рендерит элементы на странице, чтобы их отобразил браузер.

Этот код написан на JSX, его перегоняет babel. Чтобы переписать этот код на чистом JS, надо сделать так:

```
const el = React.createElement('h1', null, 'Hello World!!');
```

Структура проекта

Компоненты выносят в отдельные файлы и сохраняют в каталоге components. Один компонент – один файл.

HTML, CSS и JS компоненты называются одинаково.

Используется kebab-case для наименования файлов.

Не забыть импортировать react и сделать export default.

Каталоги

Public хранит статичные ресурсы
Src действующий код

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import TodoList from './components/todo-list';
```

Элементы, компоненты, фрагменты

Элемент – это самый маленький кирпич.

По сути, это присваивание в переменную элемента, который описан через JSX.

Создать один элемент

```
const el = <h1>Hello, World</h1>;
```

Создать сразу несколько вложенных элементов

```
const el = (
  <div>
    <h1>My Todo List</h1>
    <input placeholder="search"/>
    <ul>
      <li>Learn React</li>
      <li>Build App</li>
    </ul>
  </div>
);
```

Компоненты – это независимые блоки пользовательского интерфейса, которые могут иметь своё поведение.

Создать реакт-компонент – это создать функцию, которая возвращает элемент. Они нужны для того, чтобы легко переиспользовать компонент в других частях приложения и прикручивать им логику.

Внутри такой функции может быть только один элемент-корень, но внутри корня можно вложить сколько угодно других элементов.

Наименование функции-компонента пишется с большой буквы. Это требование реакта, чтобы отличать обычные теги от react-компонентов.

Вставку null / undefined, true / false react проигнорирует.

Контролируемые элементы (в компонентах).

Элемент является контролируемым, если он управляет из state. По сути, это связь между state и данными элемента. Для того, чтобы сделать элемент контролируемым, нужно сделать так, чтобы значение элемента устанавливалось из state компонента. Например, если текстовое поле в форме очищается из state после события onSubmit.

Создание компонента

```
const TodoList = () => {
  return (
    <ul>
      <li>Learn React</li>
      <li>Build App</li>
    </ul>
  );
};
```

Использование компонента

используется так, будто это обычный элемент

```
const el = (
  <div>
    <TodoList />
  </div>
);
```

Fragment

Во фрагментах элементы могут иметь несколько корневых элементов (в отличие от компонентов).

Поэтому фрагмент – это специальный элемент, который служит обёрткой, когда надо поместить несколько элементов в один контейнер.

Можно импортировать вместе с React

```
import React, { Fragment } from 'react';

const PlanetView = () => {
  return (
    <Fragment>
      <img className="planet-image" alt="" src={`https`} />
      <div> ... </div>
    </Fragment>
  );
};
```

Можно использовать без именованного импорта

```
const PlanetView = () => {
  return (
    <React.Fragment>
      <img className="planet-image" alt="" src={`https`} />
      <div> ... </div>
    </React.Fragment>
  );
};
```

Использование JSX

В JSX можно использовать как обычные html-теги, так и имена react-компонентов.

Кроме обычной html-разметки, можно вставлять js-выражения или переменные.
Такой код пишется в фигурных скобках.

Можно вставлять только str, num, bool, null, undefined и react-элементы.

Игнорируется вставка null, undefined, true, false. Это можно использовать с тернарным оператором.

Вставлять объекты нельзя.

Вставлять массивы с элементами можно.

Все переданные значения становятся обычным текстом. Это позволяет делать безопасные вставки.

```
<TodoList /> Компонент добавляется в JSX
{ loginBox } Переменная JS, которая хранит реакт-элемент или текст, вставляется в JSX
{ newDate() } Не сработает, потому что объект. Надо привести к строке.
```

Использование JS выражений

```
const TodoList = () => {
  const items = ['Learn React', 'Build App'];
  return (
    <ul>
      <li>{ items[0] }</li>
      <li>{ (new Date()).toString() }</li>
    </ul>
  );
};
```

null сочетается с тернарным оператором

```
const App = () => {
  const loginBox = <span>Log in please</span>
  return (
    <div>
      { isLoggedIn ? null : loginBox }
    </div>
  );
};
```

Вставка массива

```
const elements = [li, li, li...];
<ul> { elements } </ul>
```

JSX безопасно вставляет HTML как текст

```
const value = '<script> alert(!) </script>';
return (
  <div> { value } </div>
);
```

Атрибуты и свойства компонентов

Атрибуты

Наименование атрибутов – строго в camelCase.

Стандартные атрибуты прописываются как обычно.

Кроме того, атрибуты можно передавать как переменные.

Если свойству или атрибуту не передать значение явно, то его значение будет true.

```
2 атрибута, которые называются в JSX по-другому
className вместо class
htmlFor вместо for
```

Значения из переменных

```
const searchText = 'Type here to search';
<input placeholder={ searchText }/>;
```

Значение атрибута по умолчанию – true

```
return <input
  disabled
  disabled = { true }
/>;
```

Свойства

В реакт-компоненты можно передавать свойства. Они записываются в компонент так же, как атрибуты, но значением свойства может быть всё, что угодно. После передачи, свойства доступны в функции-конструкторе через пропсы. В функцию-конструктор props передаются первым параметром в виде объекта.

Наименование свойств – в camelCase.

Если свойству или атрибуту не передать значение явно, то его значение будет true.

Набор свойств можно сначала сохранить как объект в переменную, а потом распаковать его через spread.

Передача свойств

```
<Component key1={ value1 } key2={ value 2 } />
```

Использование в функции-конструкторе

```
const Component = (props) => {
  props.key1;
  props.key2;
}

const component = (props) => {
  const {key1, key2} = props;
}

const Component = ( {key1, key2} ) =>
```

Передача всех свойств через spread

Переменная хранит набор свойств

```
const item = { label: 'Make Awesone App', important: true }
```

Обычная передача

```
<TodoList Item label={item.label} important={item.important} />
```

Через ...rest

```
<TodoList Item { ...item } />
```

CSS и стили

Инлайн-стили описываются в объекте, сохраняются в переменную.

Переменная вставляется в атрибут style.

```
const searchStyle = { fontSize: '20px' };

return <input style={ searchStyle } placeholder={ searchText } />;
```

Webpack позволяет импортировать **css-файлы** со стилями прямо в JS файл.

```
import './todo-list.css';
```

Bootstrap

CDN подключается [здесь](#).

Жизненный цикл

lifecycle hooks – специальные функции, которые React автоматом вызывает на каждом этапе жизненного цикла.

Этапы жизненного цикла:

Mounting

Когда срабатывает

Компонент создаётся и первый раз отображается на странице.

Если этот метод вызван, значит элементы уже гарантированно находятся на странице.

Что делать (не использовать для этого всего конструктор):

- проводить инициализацию компонента,
- делать сетевые запросы,
- получать данные в компонент,
- работать с DOM.

Что вызывается:

```
constructor() => render() => componentDidMount()
```

Updates

Когда срабатывает

Компонент получает обновления, т.е. это либо:

- пришли новые свойства
- изменился state (вызван setState)

При этом функция срабатывает после того, как state обновлён.

Что делать:

Запрашивать новые данные для обновлённых свойств.

Что вызывается:

```
render() => componentDidUpdate(prevProps, prevState)
  if (this.props.personId !== prevProps.personId) {
    this.updatePerson();
  }
}
```

Unmounting

Когда срабатывает

Во время удаления компонента со страницы . В момент вызова DOM-элемент всё ещё будет находиться на странице.

Что делать:

- очищать те ресурсы, с которыми работал компонент
- останавливать запущенные таймеры
- останавливать запросы к серверу

Что вызывается:

```
componentWillUnmount()
```

Error

В компоненте произошла ошибка, которая не была поймана раньше.

Работает с ошибками жизненного цикла и метода rendering.

Это не замена try-catch.

Что делать:

Отключить ветку, в которой произошла ошибка.

Что вызывается:

```
componentDidCatch(error, info)
```

error – типичная ошибка JS

info – детали ошибки, характерные для React

Некоторые методы жизненного цикла компонента в порядке срабатывания:

```
componentDidMount()
componentDidUpdate()
componentWillUnmount()
componentDidCatch()
```

Граница ошибок

У проекта может быть несколько уровней компонентов, и если ошибка поймана, то вся цепочка вложенных компонентов может быть отключена. Если поймать ошибку на самом верхнем уровне Root, то выключится вся страница. Если на любом из нижестоящих – то только нижестоящий, а остальные продолжат работу.

Принцип создания таких отсекаемых компонентов:

```
import ErrorIndicator from 'error-indicator';
export default class My_component extends Component {

  state = {
    hasError: false;
  }
  componentDidCatch() {
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) {
      return <ErrorIndicator />;
    }

    return (
      ...
    );
  }
}
```

Это называется границей ошибок, error bounding. Поскольку они ловят ошибки ниже себя по иерархии и ограничивают область действия этих ошибок.

```
<Root>          <-- componentDidCatch()
  <Header>
    <Menu>
  </header>

  <LeftWidget>    <-- componentDidCatch()
    <Title>
    <Content>
  </LeftWidget>

  <RightWidget>   <-- componentDidCatch()
    <Title>
    <Content>
  </RightWidget>
</Root>
```

Это похоже на обычный try-catch. По аналогии, если в componentDidCatch() будет выброшено исключение, оно будет передано в следующий componentDidCatch().

Два аспекта:

- componentDidCatch() работает только с ошибками в методах жизненного цикла и в рендеринге.

А ошибки, например, в eventListener в каком-нибудь onClick не будут тут обрабатываться, или в асинхронных колбеках, даже если они были инициированы в методах жизненного цикла.

- это не замена стандартным проверкам. Его роль – ловить действительно непредвиденные ошибки и в идеале он вообще не должен работать. Они позволяют одному компоненту закрашиться, но при этом всё остальное приложение будет жить.

Классы

Главное отличие классов от функций – наличие свойства state, в котором можно хранить состояние.

Состояние – это разного рода переменные, которые влияют на внешний вид элемента или его потомков.

Если компоненту нужно работать со своим внутренним состоянием, (например, с счётчиком кликов или менять цвет элементов в зависимости от чего-то), то нужно использовать класс, а не функцию-конструктор.

Импорты

```
import React from 'react';
export default class TodoListItem extends React.Component {

  import React, {Component} from 'react';
  export default class TodoListItem extends Component {
```

render()

Этот метод возвращает элемент. Он делает то же самое, что функция-конструктор и прописывается так же, кроме аргументов. Доступ к аргументам .render получает так:

```
render() {
  const {label, important = false} = this.props;
  return (...);
```

State, внутреннее состояние

state

В реакте внутреннее состояние компонентов хранится в специальном поле state. Это основная причина для использования классов.

Он обязательно должен быть объектом, в котором можно сохранить любую информацию.

Если данные нужно использовать в нескольких компонентах, их надо хранить в родительском компоненте.

старомодный синтаксис

```
constructor() {
  super();
  this.state = { ... };
}
```

новый синтаксис

эквивалентно инициализации в конструкторе (поля классов)
state = { ... };

setState()

В реакте есть важное правило: state нельзя изменять после инициализации напрямую. Изменения в state вносятся через setState().

В функцию setState передаются только те свойства объекта, которые надо изменить. setState() говорит реакту: состояние этого компонента изменилось, перерендири его, заново вызови функцию render().

В функции render() прописана вся логика, часть которой опирается на state.

Например, в state можно указать, выполнен ли элемент списка задач. Render прочитает state и, в зависимости от данных в state, выберет класс done или удалит элемент.

setState может работать асинхронно.

Чтобы учитывать асинхронность, надо возвращать анонимную функцию, которая принимает в качестве первого параметра текущий state. Это означает, что текущий код надо выполнить только тогда, когда текущий state будет в финальном состоянии и его можно будет использовать для того, чтобы вычислить новый state (к вопросу об асинхронности). Этот подход используется, когда новое значение state опирается на старое значение.

Например, если надо поменять true или false на обратное значение, увеличить счётчик на единицу.

Если новое состояние никак не зависит от старого состояния, то не надо использовать функцию, можно просто передавать объект с новым состоянием.

Первым параметром в функцию setState автоматом передаётся state, его можно деструктуризовать сразу.

В примере ниже в функцию передаётся объект с изменениями, которые надо внести в state:

```
onMarkImportant = () => {
  this.setState( (state) => {
    return { important: !state.important };
  });
};

onMarkImportant = () => {
  this.setState( {important} ) => {
    return { important: !important };
  );
};

упрощённо
onMarkImportant = () => {
  this.setState( { important: true } );
}
```

Обработка событий

Главная задача – правильно передать this и не наебаться.

Обработчики-атрибуты

Всё как в обычных html: onClick = { функция }

Обработчик в экземпляре класса

```
constructor() {
  super();
  this.onLabelClick = () => {
    console.log(`Done: ${this.props.label}`)
  }
}

render() {
  const {label, important = false} = this.props;
  ...
}
```

```
<span  
  onClick={ this.onLabelClick } >  без вызова() функции!  
</span>
```

Ещё вариk

Забыл, как называется новый синтаксис. Вписывается без конструктора, просто так в теле

```
export default class TodoListIem extends Component {  
  onLabelClick = () => {  
    console.log(`Done: ${this.props.label}`)  
  }  
}
```

React фишули

Пометить li как важный или выполненный в todo list

```
const { important, done } = this.props;  
  
let classNames = 'todo-list-item';  
  
if (done) classNames += ' done';  
if (important) classNames += ' important';  
  
return <span className={classNames} />
```

React API

Children

props.children

Второй способ передать в компоненты свойства – это записать их в «тело» тега между открывающим и закрывающим тегом.

Доступ к таким свойствам осуществляется через: this.props.children

Для работы с children есть специальный API React.Children.

Документация [тут](#).

Каждого children можно скрывать, оборачивать, делать что угодно. Потому что компонент не обязательно должен использовать child в том виде, в котором они переданы.

Изменять children нельзя. Для изменений, их надо скопировать и менять копии. Для этого существует React.cloneElement().

this.props.children

Обращение ко всему, что было передано в теле элемента

```

React.Children.map(children, function(el, index))
React.Children.forEach(children, function[(thisArg)])
React.Children.count(children)
React.Children.only(children)
React.Children.toArray(children)

const itemList = (
  <ItemList
    onItemSelected={this.onPersonSelected}
    getData={this.swapiService.getAllPeople}>
    {((i) => (
      `${i.name}, ${i.birthYear}`
    )));
  </ItemList>
);

```

Context api

Специальное хранилище данных, которое позволяет не прорасывать эти данные через пропсы от вышестоящих в иерархии компонентов к нижестоящим, которые эти данные применяют.

Контекст нужен для того, чтобы решить проблему «глобальных» данных.

Вместо того, чтобы передавать props через все слои приложения, данные можно передавать через контекст. С помощью контекста можно сделать так, чтобы компоненты не создавали объекты сервиса, а получали этот один объект.

Создаются 2 компонента:

Provider – в котором в свойстве value указываются данные, которые надо протащить.

Consumer – используется для получения данных. Должен возвращать функцию.

Создание файла-компоненты для контекста

`React.createContext();`

Значение по умолчанию, если consumer не сможет найти никакой контекст.

Возвращает пару: Provider и Consumer.

```

const {
  Provider: SwapiServiceProvider,
  Consumer: SwapiServiceConsumer
} = React.createContext();

```

Кратко

```

// в модуле
const { Provider, Consumer } = React.createContext();

// в родительском файле
<Provider value={someValue}>
  // провайдер обворачивает часть приложения
</Provider>

// в файле-потомке любой вложенности
<Consumer>
  { (someValue) => <MyComponentData={someValue} /> }
</Consumer>

```

Пример

Provider importуется в вышестоящий компонент (app.js) и обергает нижестоящие компоненты, получает value={}, который надо в них протащить.

```
<SwapiServiceProvider value = {this.swapiService}>
  <el>
    <el>
      <details>
        <el>
</SwapiServiceProvider>
```

Consumer используется в компоненте, который использует этот value.

Его тоже надо импортировать.

Он возвращает функцию, которая первым параметром получит value.

Можно сразу деструктуризировать (в примере для наглядности без деструктуризации).

```
const PersonDetails = ({itemId}) => {
  return (
    <SwapiServiceConsumer>
      { эта скобка обязательно должна стоять здесь, а не выше
        (swapiService) => {
          return (
            <ItemDetails
              itemId={itemId}
              getData={swapiService.getPerson}
              getImageUrl={swapiService.getPersonImage}>
              </ItemDetails>
            );
          }
        }
      </SwapiServiceConsumer>
    );
};
```

Hooks

Справочник API хуков [здесь](#).

Хук – это специальная функция, которая даёт возможность функциональным компонентам использовать почти все возможности компонентов-классов:

- использование state;
- методы жизненного цикла;
- передача контекста;
- какие-то другие возможности.

Правила хуков:

- Можно вызывать только из react-компонентов и собственных хуков.
- Нельзя вызывать внутри циклов, условных операторов или вложенных функций.
- Не работают в компонентах-классах.
- Не покрывают все методы жизненного цикла: при помощи хуков нельзя создать componentDidCatch.

Документация [здесь](#).

React содержит несколько встроенных хуков, таких как useState. Вы также можете создавать собственные хуки.

Основные хуки

```
import React, { useState, useContext, createContext } from 'react';
```

useState

используется для обновления состояния

```
const [state, setState] = useState(initialState);
```

старое значение нельзя мутить,

но можно использовать его для возврата из функции нового значения

```
setState((oldState) => oldState + 1);
```

объекты нужно деструктуризировать

```
setState((oldState) => {...oldState, name: value})
```

useContext

```
const MyContext = createContext();
```

```
const value = useContext(MyContext);
```

Используется для получения значения из контекста.

Контекст – это пропс value ближайшего <MyContext.Provider>.

useEffect

сработает только при изменении value

```
useEffect( () => console.log('Вызвана'), [value]);
```

Дополнительные хуки

```
useReducer
```

```
useCallback
```

```
useMemo
```

```
useRef
```

```
useImperativeHandle
```

```
useLayoutEffect
```

```
useDebugValue
```

useState

Используется для обновления состояния.

```
import React, { useState, useContext, createContext } from 'react';
```

useState

```
const [state, setState] = useState(initialState);
```

state

Переменная хранит какое-то текущее значение

```
setState()
```

Это функция, в которую надо передать новое значение для изменения state.

Старое значение нельзя мутить.

Как и в случае использования «классического» state, нельзя мутировать старое значение. Если надо получить новое значение на основе старого (вычесть из старого или прибавить к нему), то надо использовать функцию и, после обработки логики, вернуть из неё новое значение.

Первый аргумент функции – старый state.

```
setState((oldState) => oldState + 1);
```

```
Если в state хранится объект с данными, объекты можно деструктуризовать и дописать в конце  
новое значение, которое надо обновить:  
setState((oldState) => {...oldState, name: value})
```

Также, чтобы обновить поле в «классическом» state, нужно было вернуть из setState только одно обновлённое свойство: setState({ поле: значение })

Хук работает иначе.

Если в state хранится объект с данными и надо обновить только одно его поле, то объект можно деструктуризовать и дописать в конце новое значение, которое надо обновить:

```
setState((oldState) => {...oldState, name: value})
```

useContext

Единственное отличие хука от обычного контекста – это способ получения данных:

1. Вызвать функцию createContext() с сохранить вызов в переменную. Эта функция возвращает объект.
2. В любом месте создать элемент-провайдер <Переменная.Provider value='123'>.

В элемент-провайдер обернуть элемент, который будет потребителем контекста.

Всё, как обычно.

3. Единственное отличие – потребитель получает контекст так: value = useContext(Переменная).

```
import React, { useContext, createContext } from 'react';
import ReactDOM from 'react-dom';

const MyContext = createContext();

const App = () => {
  return (
    <MyContext.Provider value="Hello World">
      <Child />
    </MyContext.Provider>
  );
};

const Child = () => {
  // какой именно контекст надо использовать
  const value = useContext(MyContext);
  return <p>{value}</p>;
};

ReactDOM.render(<App />, document.getElementById('root'));
```

useEffect

Документация [тут](#).

Заменяет компоненты жизненного цикла:

- componentDidMount
- componentDidUpdate
- componentWillUnmount

В зависимости от того, какие переданы аргументы в функцию, хук будет срабатывать на разных этапах жизненного цикла компонента, в котором он находится:

```
componentDidUpdate и componentDidMount
```

```

useEffect( () => console.log('Вызвана'));

componentDidMount
useEffect( () => console.log('Вызвана'), []);

сработает только при изменении value
useEffect( () => console.log('Вызвана'), [value]);

componentWillUnmount
useEffect(() => () => console.log('unmount'), []);

componentDidMount и componentWillUnmount
useEffect( () => {
  console.log('Создана');
  return () => console.log('Удалена');
}, []);

```

Принимает 2 аргумента:

1. функцию, которая будет срабатывать при определённых условиях;
2. необязательно – массив данных. Будет срабатывать только если данные из массива обновились. Пустой массив – useEffect сработает только при componentDidMount.

Очистка предыдущих эффектов.

Если в первом аргументе сделать return и вернуть какую-то функцию, то она будет срабатывать при удалении или обновлении компонента.

Чтобы функция срабатывала только при удалении компонента, надо передать пустой массив.

useCallback

Документация [здесь](#).

Возвращает мемоизированный колбэк (кеширует функцию).

Мемоизация - сохранение результатов выполнения функций для предотвращения повторных вычислений.

Хук вернёт мемоизированную версию колбэка, который изменяется только, если изменяются значения одной из зависимостей. Это полезно для предотвращения ненужных рендеров.

Если данные в массиве не изменяются, то useCallback вернёт ту же ссылку на ту же функцию.

```

const foo = useCallback(fn, [deps])

аналог:
const foo = useMemo(() => fn, [deps])

```

useMemo

Документация [здесь](#).

Возвращает мемоизированное значение (кеширует значение).

Первый аргумент – функция, которая создаёт и возвращает какое-то значение.

Второй – те данные, от которых это значение зависит. Если передать пустой массив, то это означает, что значение не зависит ни от каких данных и будет вычисляться только 1 раз.

Если в коде есть какое-то значение, которое надо передавать в useEffect, чтобы сравнить старое состояние и новое, то этот хук понадобится.

```
const memorized = useMemo( () => a, [a]);
```

Хуки и загрузка данных

```
const PlanetInfo = ({id}) => {
  const [name, setName] = useState(null);

  useEffect( () => {
    let cancelled = false;

    fetch(`https://swapi.dev/api/planets/${id}/`)
      .then(res => res.json())
      .then(data => !cancelled && setName(data.name));
    return () => cancelled = true;
  }, [id]);

  return (
    <div>{id} - {name}</div>
  );
};
```

Создание хуков

Хук – функция, имя которой начинается с use. Внутри используются стандартные хуки.

```
const useCustomHook = (id) => {
  const [name, setName] = useState(null);

  useEffect( () => {
    let cancelled = false;
    fetch(`https://swapi.dev/api/planets/${id}/`)
      .then(res => res.json())
      .then(data => !cancelled && setName(data.name));
    return () => cancelled = true;
  }, [id]);

  return name;
};

const PlanetInfo = ({id}) => {
  const name = useCustomHook(id);
  return <div>{id} - {name}</div>;
};
```

Паттерны React

Большинству приложений необходимы вспомогательные компоненты:

- ErrorBoundary
- контекст или несколько контекстов
- НОС для работы с контекстом (withXyzService)

Эти и другие вспомогательные компоненты лучше создать до начала работы над основным функционалом приложения.

Из подкурса react-redux, каркас react-redux приложения:

```
ReactDOM.render(  
  доступ к Redux Store  
  <Provider store={store}>  
  
  обработка ошибок в компонентах ниже  
  <ErrorBoundary>  
  
    передаёт сервис через Context API  
    <BookstoreServiceProvider value={bookstoreService}>  
  
      из пакета react-router  
      <Router>  
        <App />  
  
      </Router>  
    </BookstoreServiceProvider>  
  </ErrorBoundary>  
</Provider>,
```

props.children

Второй способ передать в компоненты свойства – это записать их в «тело» тега между открывающим и закрывающим тегом.

Доступ к таким свойствам: this.props.children

```
this.props.children  
Обращение ко всему, что было передано в теле элемента
```

```
const itemList = (  
  <ItemList  
    onItemSelected={this.onPersonSelected}  
    getData={this.swapiService.getAllPeople}  
  >  
  {((i) => (  
    `${i.name}, ${i.birthYear}`  
  ))};  
  
  </ItemList>  
)
```

ErrorBoundary

Крутая штука – отлавливатель ошибок на любых компонентах. См. ErrorBoundary в проекте SWAPI.

```
class ErrorBoundary extends Component {  
  state = {  
    hasError: false  
  };  
  
  componentDidCatch() {  
    this.setState({  
      hasError: true  
    });  
  }  
  
  render() {  
    if (this.state.hasError) {
```

```
        return <ErrorIndicator />
    }
    return this.props.children;
}
};
```

Любой элемент, на котором надо отловить ошибки, оборачивается в это добро:

```
<ErrorBoundary>
  <PersonDetails/>
</ErrorBoundary>
```

HOCS – компоненты высшего порядка

Это функции-обёртки.

Этот компонент берёт на себя обязанности, о которых не нужно заботиться внутреннему компоненту. Например, получать данные по сети, отображать спиннер или компонент с ошибкой.

Можно сказать, что обёртка занимается менеджментом данных. В неё можно передавать любой компонент, для которого она выполняет рутинные операции, а компонент будет только отображать данные. В неё выносится вся логика, которая была в оригинальном компоненте: state, componentDidMount, спиннер.

Раньше оригинальный компонент получал данные из state. Поскольку state теперь находится в компоненте-обёртке, теперь он будет получать данные из props.

Сопутствующие данные и функции (типа что именно запрашивать по сети), необходимые для оборачиваемого компонента, можно передать в hoc аргументами (пропсами).

```
const hoc = (MyComponent, [options]) => {
  return class extends Component {
    state {...}
    componentDidMount() {...}
    getData() {...}

    render() {
      const { data } = this.state;
      if (!data) return <Spinner />
      return <MyComponent {...this.props} />
    }
  }
}

const MyWrappedComponent = hoc(InnerComponent);
```

Композиция функций

Композиция – применение одной функции к результату другой.

Результат работы одной функции передаётся в другую функцию.

Компоненты высшего порядка – это обычные функции, которые возвращают компоненты, используя метод композиции. Так можно применять несколько эффектов HOCS.

```
const comp = (x) => f( g(x) );
```

HOCS-context

Обязанность получать данные из контекста можно вынести в компонент высшего порядка.

```
const withValueFromContext = (Wrapped) => {
```

```
return (
  <Consumer>
    { (value) => <Wrapped value={value} /> }
  </Consumer>
);
};
```

Зачем нужны паттерны?

Задача паттернов – избегать копирования кода и правильно переиспользовать части поведения компонентов.

Однаковые компоненты могут иметь разное содержание: список персонажей, кораблей, планет. Компоненты будут отличаться данными с сервера, но процесс получения и обработки данных (паттерн создания) у них одинаковый:

- запрос данных с сервера при создании;
- отрисовка этих данных;
- обработка ошибок с error indicator.

Чтобы избежать копипасты и не плодить одинаковые компоненты, можно сделать один универсальный компонент, который будет просто получать разные данные и одинаково работать с ними.

Использование функций

Функции могут инкапсулировать получение данных по сети и передавать их через пропсы. Эта часть про то, как можно скрыть получение данных.

itemList может стать универсальным компонентом, который работает с разными данными, не только со списком персонажей. Важно! Универсальным станет только itemList, т.е. только списки слева, но не PersonDetails.

Если представить, что itemList используется для запроса списка планет или кораблей, то вот их сходства и различия:

swapiService
У всех присутствует
RenderItems
не изменяется.

componentDidMount

В зависимости от того, какую сущность надо отобразить, выполняется разный запрос (метод) swapiService. Единственное, что меняется у компонентов – функция для запроса данных:

- getAllPeople()
- getAllStarships()
- getAllPlanets()

Обновляют состояние setState они одинаково. Логичный шаг – вынести получение данных наружу из компонента.

Что делать:

1. SwapiService удаляется из компонента itemList. Функцию запроса данных можно вынести наружу. Вместо того, чтобы создавать экземпляр класса внутри компонента, этот компонент получит функцию getData из свойств.

getData тоже будет возвращать промис. Компонент будет получать данные из этой функции, устанавливать их в качестве своего state.

Все peopleList заменяются на общее itemList, потому что компонент будет заниматься не только персонажами, а всем, чем угодно.

Было:

```
item-list.js

state = {
  peopleList: null
};

swapiService = new SwapiService();

componentDidMount() {
  this.swapiService.getAllPeople()
    .then((peopleList) => {
      this.setState({
        peopleList
      });
    });
}
```

Стало:

```
state = {
  itemList: null
};

componentDidMount() {
  const { getData } = this.props;

  getData()
    .then((itemList) => {
      this.setState({
        itemList
      });
    });
}
```

2. сделать методы SwapiService стрелочными функциями, чтобы this не терялся:

swapi-service.js

```
getResource = async (url) => {

getAllPeople = async () => {
getPerson = async(id) => {

getAllPlanets = async () => {
getPlanet = async (id) => {

getAllStarships = async () => {
getStarship = async (id) => {

_extractId = (item) => {

_transformPlanet = (planet) => {
_transformStarship = (starship) => {
_transformPerson = (person) => {
```

3. Изменить способ вызова универсального компонента.

Важно! Универсальным стал только ItemList, т.е. списки слева, но не PersonDetails.

Такие же изменения надо внести в компонент people-page, потому что там тот же самый код.

Было:

app.js

```
<div className="row mb2">
  <div className="col-md-6">
    <ItemList onItemSelected={this.onPersonSelected} />
  </div>
  <div className="col-md-6">
    <PersonDetails personId={this.state.selectedPerson} />
  </div>
</div>
```

Стало:

```
<div className="row mb2">
  <div className="col-md-6">
    <ItemList
      onItemSelected={this.onPersonSelected}
      getData={this.swapiService.getAllPlanets}
    />
  </div>
  <div className="col-md-6">
    <PersonDetails personId={this.state.selectedPerson} />
  </div>
</div>

<div className="row mb2">
  <div className="col-md-6">
    <ItemList
      onItemSelected={this.onPersonSelected}
      getData={this.swapiService.getAllStarShips}
    />
  </div>
  <div className="col-md-6">
    <PersonDetails personId={this.state.selectedPerson} />
  </div>
</div>
```

Render-функции

Паттерн, в котором в компонент в качестве пропса передаётся функция, которая занимается рендерингом всего компонента или его части. Такая функция возвращает строку, react-элемент или разметку jsx.

Это чем-то похоже на функцию .filter, которая решает, что оставить (в данном случае, что отобразить).

Допустим, надо сделать так, чтобы в списке персонажей рядом с именем отображались также пол и возраст, а в списке планет – диаметр, для кораблей – модель. Т.е. должна быть какая-то дополнительная информация.

До сих пор все элементы списков отображались одинаково. Внутри компонента item-list есть функция renderItems. Она печатает только свойство name и это свойство есть у всех сущностей.

item-list.js

```
renderItems(arr) {
  return arr.map(({id, name}) => {
    return (
      <li
        className="list-group-item"
        key={id}
        onClick={() => this.props.onItemSelected(id)}
      >
        {name}
      </li>
    );
  });
}
```

```
});  
}
```

Теперь требования изменились и в зависимости от типа сущностей надо отображать разную информацию. Один из способов решить эту задачу – использовать render-функцию и передавать её через пропсы.

Функция renderItem() (внутри компонента item-list) получает на вход item и возвращает то, что надо будет отрисовать.

Такие же изменения надо внести в компонент people-page, потому что там тот же самый код.

app.js

```
<div className="row mb2">  
  <div className="col-md-6">  
    <ItemList  
      onItemSelected={this.onPersonSelected}  
      getData={this.swapiService.getAllStarships}  
      renderItem={(item) => item.name}>  
    />  
  </div>  
  <div className="col-md-6">  
    <PersonDetails personId={this.state.selectedPerson} />  
  </div>  
</div>
```

Теперь надо перейти в компонент item-list и использовать эту рендер-функцию: заменить прошлый блок {name} на результат вызова рендер-функции. Name больше не используется, его не надо деструктуризировать, а вместо person в массиве содержится любой элемент, поэтому он заменяется на item.

item-list.js

```
renderItems(arr) {  
  // было return arr.map((person) => {  
  return arr.map((item) => {  
    const { id } = item;  
    const label = this.props.renderItem(item);  
  
    return (  
      <li  
        className="list-group-item"  
        key={id}  
        onClick={() => this.props.onItemSelected(id)}  
      >  
        {label}  
      </li>  
    );  
  });  
}
```

Теперь для каждого компонента ItemList можно определять, что он будет выводить с помощью рендер-функции. Так можно сделать в app.js и в people-page.js:

```
<ItemList  
  onItemSelected={this.onPersonSelected}  
  getData={this.swapiService.getAllPeople}  
  renderItem={(item) => `${item.name} (${item.gender}, ${item.birthYear})`}>
```

сразу деструктуризировать:

```
renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}/>
```

Пока что возвращается только текст для отображения в li. Можно пойти дальше и вернуть JSX-разметку. В примере ниже будет создаваться кнопка:

```
renderItem={(item) => (<span> {item.name} <button>!</button> </span>)}
```

Свойства-элементы (контейнеры)

В качестве свойств в компонент можно передавать другие react-элементы.

Сейчас на странице отображаются 2 элемента рядом: список персонажей и их детализация. Может понадобиться в дальнейшем создавать группы компонентов в таком же формате: список кораблей и их детализация и т.д. Чем больше копипасты – тем сложнее поддерживать код, потому что придётся делать несколько одинаковых правок сразу в нескольких местах. Для этого разметку (не имеющиеся сейчас компоненты, а именно разметку, в которую они обёрнуты), можно вынести в отдельный компонент. Это именно создание компонента из разметки: кода html и стилей без сложного функционала и логики.

Можно сделать компонент-контейнер, который будет принимать через пропсы другие компоненты и правильно их размещать. Этот контейнер должен уметь работать с любыми двумя компонентами, которые нужно разнести по двум сторонам, а не предзаданными. Соответственно, у него в пропсах будут 2 свойства: left и right.

После этого, если понадобится переиспользовать разметку, не нужно будет копировать html, а достаточно переимпользовать этот компонент.

Кроме того, можно прикрутить логику, чтобы такие элементы могли выбирать, что рендерить в зависимости от условия (загрузка, ошибка и т.д.).

Было:

```
people-page.js
```

```
render() {
  if (this.state.hasError) {
    return <ErrorIndicator />;
  }

  return (
    <div className="row mb2">
      <div className="col-md-6">
        <ItemList
          onItemSelected={this.onPersonSelected}
          getData={this.swapiService.getAllPeople}
          renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}/>
      </div>
      <div className="col-md-6">
        <PersonDetails personId={this.state.selectedPerson} />
      </div>
    </div>
  );
}
```

В файле people-page.js для удобства в отдельные константы будут вынесены itemList и personDetails, потому что сейчас эти компоненты разрастаются, в их пропсы передаётся много кода.

Вся окружающая их разметка будет вынесена в отдельный компонент Row.

Стало:

```
row.js
const Row = ({ left, right }) => {
  return (
    <div className="row mb2">
```

```

        <div className="col-md-6">
          {left}
        </div>
        <div className="col-md-6">
          {right}
        </div>
      </div>
    );
};

people-page.js
render() {
  if (this.state.hasError) {
    return <ErrorIndicator />;
  }

  const itemList = (
    <ItemList
      onItemSelected={this.onPersonSelected}
      getData={this.swapiService.getAllPeople}
      renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}
    />);

  const personDetails = (
    <PersonDetails personId={this.state.selectedPerson} />
  );

  return (
    <Row left={itemList} right={personDetails} />
  );
}

```

Для приложения можно создать также другие элементы-контейнеры, чтобы переиспользовать блоки html и css. Например, вместо random-planet можно сделать элемент-контейнер, который будет заниматься не только планетами.

Children + ErrorBoundary

Документация по API Children [тут](#).

Есть 2 способа передавать свойства компонентам (равнозначны):

- передавать пары ключ-значения, похожие на атрибуты в html;
- записать что-то в тело компонента.

```

<Component>
  Hello, {[1, 2, 4]}
</Component>

class Component extends... () {
  const data = this.props.children;
}

const Component = (props) => {
  const data = this.props.children;
}

```

Сейчас в файле people-page.js компонент ItemList получает рендер функцию так:

```
people-page.js
```

```
<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
  renderItem={({name, gender, birthYear}) => `${name} (${gender}, ${birthYear})`}
/>
```

Поскольку рендер-функция отвечает за то, что будет выведено на экран, её можно передать в теле компонента. Деструктуризации больше нет, чтобы всё было более читаемо.

```
people-page.js
```

```
const itemList = (
  <ItemList
    onItemSelected={this.onPersonSelected}
    getData={this.swapiService.getAllPeople}
  >
    {(i) => (
      `${i.name}, (${i.birthYear})`
    )}
  </ItemList>
);
```

Теперь надо сделать так, чтобы сам компонент `ItemList` мог получить доступ к содержимому тела. Доступ к такой информации можно получить через `this.props.children`.

`.children` работает только в том случае, если из компонента `App` удалить разметку, которая «не компонент» и создаёт другие пары: корабли и планеты. См. ответ [здесь](#).

Было:

```
renderItems(arr) {
  return arr.map((item) => {
    const { id } = item;
    const label = this.props.renderItem(item);
```

Стало:

```
renderItems(arr) {
  return arr.map((item) => {
    const { id } = item;
    const label = this.props.children(item);
```

ErrorBoundary

С помощью `children` можно передавать дерево компонентов. Предлагается выделить в отдельный компонент `ErrorBoundary`. Он будет заниматься высекающей ошибкой. Компонент может получить один или несколько элементов в качестве `children` и он отрендерит их точно в таком виде, в котором получил.

Из компонента `people-page` переезжают в отдельный компонент `ErrorBoundary`:

- метод жизненного цикла `componentDidCatch`
- `hasError` из стейта.

В компоненте `ErrorBoundary` поступаем так же, как обычно при обработке ошибок. Этот компонент будет решать, что ему возвращать: компонент-ошибку или `props.children`.

В такой компонент можно обернуть любой другой компонент и `ErrorBoundary` автоматом будет отлавливать все ошибки, которые происходят в его `children`.

В примере ниже в ErrorBoundary можно обернуть компонент Row целиком и/или PersonDetails и удалить из этих компонентов их собственные обработчики ошибок. Компонент-обёртка ErrorBoundary будет отлавливать всё.

error-boundry.js

```
class ErrorBoundary extends Component {
  state = {
    hasError: false
  };

  componentDidCatch() {
    this.setState({
      hasError: true
    });
  }

  render() {
    if (this.state.hasError) {
      return <ErrorIndicator />
    }
    return this.props.children;
  }
};
```

people-page.js

```
render() {

  const itemList = (
    <ItemList
      onItemSelected={this.onPersonSelected}
      getData={this.swapiService.getAllPeople}
    >
      {((i) => (
        `${i.name}, ${i.birthYear}`)
      )}
    </ItemList>
  );
}

const personDetails = (
  <ErrorBoundary>
    <PersonDetails personId={this.state.selectedPerson} />
  </ErrorBoundary>
);

return (
  <ErrorBoundary>
    <Row left={itemList} right={personDetails} />
  </ErrorBoundary>
);
}
```

Практика - рефакторинг компонента

Файл person-details.js и все person-переменные переименовываются в item-details.js и item соответственно.

Поскольку item-details должен работать с любыми данными, наружу в пропсы выносятся некоторые функции. В State добавляется поле image, чтобы хранить «универсальное» изображение, которое туда записывается после загрузки данных:

app.js

```
const { getPerson, getStarship } = this.swapiService;

const personDetails = ( <ItemDetails
```

```

itemId={11}
getData={getPerson}
getImageUrl={()/> };

const starshipDetails = ( <ItemDetails
itemId={5}
getData={getStarship}
getImageUrl={()/> };

item-details.js
state = {
  item: null,
  loading: false,
  image: null
}

getData(itemId)
  .then((item) => {
    this.setState({ item, loading: false, image: getImageUrl(item) });
  });

```

В Swapi-Service добавляются функции для получения изображений персонажей, кораблей и планет. Эти функции будут передаваться в item-details как пропсы для получения того или иного изображения:

```

swapi-service.js
_imageBase = 'https://starwars-visualguide.com/assets/img';

getPersonImage = ({id}) => {
  return `${this._imageBase}/characters/${id}.jpg`;
};

getStarshipImage = ({id}) => {
  return `${this._imageBase}/starships/${id}.jpg`;
};

getPlanetImage = ({id}) => {
  return `${this._imageBase}/planets/${id}.jpg`;
};

```

Работа с props.children

Сейчас компонент item-details хорошо работает только с персонажами, потому что у кораблей нет полей «возраст», «цвет глаз» и т.д., а Gener, Birth Year и Eye Color захардкожены. Эти значения надо вынести наружу и сделать так, чтобы компонент можно было конфигурировать.

Решить задачу можно несколькими способами.

Например, передавать в пропсы объект с конфигурацией типа fields = {[{field: 'gender', label:' Gender'}]}. Такой код – не в духе React (Юра говорит), потому что не соблюдается принцип компонентности.

Надо знать наименование полей объекта и сам объект с данными. Это можно реализовать с функцией Record, которая будет создавать li-элементы. Её параметры: item – сам элемент, field – поле, которое надо достать из объекта и label – запись на ui.

Было (захардкожено и заточено под конкретные данные):

```

<ul className="list-group list-group-flush">

<li className="list-group-item">
  <span className="term">Gender</span>
  <span>{gender}</span>
</li>

```

```

<li className="list-group-item">
  <span className="term">Birth Year</span>
  <span>{birthYear}</span>
</li>

</ul>

```

Стало (пока добавить в этот файл функцию record):

```

item-details.js

const Record = ({item, field, label}) => {
  return (
    <li className="list-group-item">
      <span className="term">{label}</span>
      <span>{ item[field] }</span>
    </li>
  );
};

export { Record };

app.js
import ItemDetails, { Record } from '../item-details';

const personDetails = ( <ItemDetails
  itemId={11}
  getData={getPerson}
  getImageUrl={getPersonImage}
>
  <Record field='gender' label='Gender' />
  <Record field='eyeColor' label='Eye Color' />
</ItemDetails> );

```

Это не всё «стало», продолжение ниже.

API Children

Чтобы Record заработал, ему надо знать item, из которого он будет доставать все эти данные. Т.е. сейчас можно достать детей (функции Record) и вывести на экран их содержимое через { this.props.children }, но нужно как-то обратиться к родительскому объекту, чтобы получить доступ к его полям. Так, например, item и item[field] пока не доступны.

Для работы с детьми есть специальный API Children, документация по нему [тут](#).

Дело в том, что children может быть строкой, элементом, числом, чем угодно. Функция React.Children.map сделает так, чтобы не надо было задумываться, какого именно типа child сейчас попался, он пройдётся по каждому child и обработает все специальные случаи типа null или undefined. Можно заменять и обрабатывать child-элементы как угодно, можно возвращать вместо них null и тем самым скрывать, можно оборачивать... не обязательно возвращать их в том виде, как они поступают.

Кратко: React.Children.map() позволяет проитерироваться по this.props.children и сделать что-нибудь с каждым child и вернуть что угодно. Есть одно правило: react-элементы нельзя изменять после того, как они были созданы.

В данном случае, в каждый child надо передать item.

```
item.details, class ItemDetails на том месте, где было захардкожено
```

```

<div className="card-body">
  <h4>{name}</h4>
  <ul className="list-group list-group-flush">
    {
      React.Children.map(this.props.children, (child, idx) => {

```

```

        return <li>{idx}</li>;
    })
}
</ul>
<ErrorButton />
</div>

```

Это не весь рефактор, сейчас пока он не работает правильно, потому что нет доступа к item, см. следующий раздел.

Клонирование элементов

`React.Children.map()` позволяет проитерироваться по `this.props.children` и сделать что-нибудь с каждым child и вернуть что угодно. Есть одно правило: react-элементы нельзя изменять после того, как они были созданы, с ними надо работать так, будто они неизменяемые. Поэтому нельзя написать внутри тар-функции `child.item = item`.

Вместо этого надо создать новый элемент, в котором есть новое свойство `item`. Для этого есть ещё один метод: `React.cloneElement()`. Документация [тут](#).

Копирует и возвращает новый реакт-элемент.

`Config` должен содержать все новые `props`, `key` или `ref`. Возвращаемый элемент будет иметь все старые + новые `props`. Новые `children` заменят существующих `children`.

```
React.cloneElement(element, [config], [...children])
```

Почти эквивалентно этому

```
<element.type {...element.props} {...props}>{children}</element.type>
```

Вот итоговая реализация:

Был создан дополнительный компонент **Record** (запись).

Ему необходимы 3 поля: «`field`», «`label`» и «`item`». Первые 2 передаются через `app`, а `item` нельзя сразу передать, потому что эти данные получаются внутри компонента `ItemDetails` и снаружи их получить никак нельзя. Внешний код ничего не знает о том, как создаётся объект `item`.

Для того, чтобы всё-таки передать объект `item` в child-элементы `ItemDetails`, была осуществлена итерация по каждому child с помощью `React.Children.map()` и внутри каждого child был преобразован с помощью `React.cloneElement()`, с помощью которого элементу был передан готовый к этому времени `item`.

```
app.js
import ItemDetails, { Record } from '../item-details';
const personDetails = (
  <ItemDetails
    itemId={11}
    getData={getPerson}
    getImageUrl={getPersonImage}
  >
    <Record field='name' label='Name' />
    <Record field='gender' label='Gender' />
    <Record field='eyeColor' label='Eye Color' />
  </ItemDetails> );

```

```
item-details.js
const Record = ({item, field, label}) => {
  return (
    <li className="list-group-item">
      <span className="term">{label}</span>
```

```

        <span>{ item[field] }</span>
    </li>
);
};

export default class ItemDetails extends Component {
...state = {
  item: null,
  loading: false,
  image: null
}

updateItem() {
  ...getData(itemId)
  .then((item) => {
    this.setState({ item, loading: false, image: getImageUrl(item) });
  });
}

render() {
  const { name, item, loading, image } = this.state;
  ...return (
    ...<ul className="list-group list-group-flush">
    {
      React.Children.map(this.props.children, (child) => {
        return React.cloneElement(child, {item});
      })
    }
  </ul>
}
}

```

Компоненты высшего порядка (HOC)

Работа с itemList (списко персонажей, кораблей и т.д.).

Как у любого другого сетевого компонента, его работа состоит из нескольких фаз:

1. В componentDidMount() делаем запрос, получаем данные, обновляем state.
2. В render() проверяем, если есть данные, то их отображаем. Если данных нет – отображаем спиннер.

Чтобы создать новый сетевой компонент, не обязательно копировать весь этот код (который отправляет запрос, проверяет наличие данных, обновляет state, отображает спиннер, ошибку и т.д.). На самом деле, меняются только 2 аспекта: как делается запрос и как отображается результат на экране.

Чтобы сделать этот код лучше, существует паттерн HOC.

Цель – вынести сетевой код, отображение данных и ошибку в отдельную конструкцию и переиспользовать с другими компонентами.

В JS функция может возвращать другую функцию и/или класс.

Вместо экспорта класса, можно экспортировать функцию (её вызов), которая возвращает класс.

```

// возврат функции
const f = (a) => {
  return (b) => {
    console.log(a + b);
  }
};
f(1)(2);

// возврат класса
const f = () => {
  return ItemList;
}

```

```
};

export default f();
```

В JS можно создавать безымянные функции и безымянные классы. Таким образом будет возвращён новый класс, у которого нет имени, но есть содержимое. В примере ниже возвращается анонимный класс, который наследует Component и у него есть функция render():

```
const f = () => {
  return class extends Component {
    render() {
      return <p>Hi</p>
    }
  };
};

export default f();
```

Поскольку внутренний анонимный класс – это компонент, значит в него можно передать componentDidMount() или любую другую функцию жизненного цикла:

```
const f = () => {
  return class extends Component {

    componentDidMount() {
      console.log(this.props);
    }

    render() {
      return <p>Hi</p>
    }
  };
};

export default f();
```

Поскольку возврат функции (а функция возвращает анонимный класс) экспортируется под именем ItemList (переименование происходит в index.js, тут я её называю anonim), то она получает все пропсы (в т.ч. children), которые в неё передаются в App (выведутся в консоль), а именно:

```
<AnonimItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>

  { ({name}) => <span>{name}</span> }

</ AnonimItemList >
```

Следующий шаг – сделать полноценный компонент-класс <ItemList>, в этом же файле создать функцию высшего порядка, которая будет брать на себя все расчёты и запросы, а возвратит компонент-класс <ItemList> и «напихает» ему уже полученных пропсов.

В примере ниже создана обёртка, которая не делает ничего. Она получает пропсы, как в примере выше, и возвращает класс <ItemList/> и передаёт ему свои пропсы. Всё будет работать, как и раньше:

```
class ItemList extends Component {
  ...объявление класса
```

```

}

const f = () => {
  return class extends Component {
    render() {
      return <ItemList {...this.props} />;
    }
  };
};

export default f();

app.js
const list = <ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>
  { ({name}) => <span>{name}</span> }
</ItemList>

```

НОС

Финальный шаг – вынести в обёртку всю логику из оригинального компонента:

- работа с сетью
- спиннер
- ошибка
- что отображать по клику

State, componentDidMount, логика спиннера и данные перезжают из компонента в обёртку. Переменная itemList переименовывается на data.

item-list.js

```

// компонент, который надо вернуть
class ItemList extends Component {
  renderItem(arr) {
    return arr.map((item) => {
      const { id } = item;
      const label = this.props.children(item);
      return (
        <li
          className="list-group-item"
          key={id}
          onClick={() => this.props.onItemSelected(id)}
        >
          {label}
        </li>
      );
    });
  }

  render() {
    const { data } = this.props;
    const items = this.renderItem(data);

    return (
      <ul className="item-list list-group">
        {items}
      </ul>
    );
  }
}

// возврат класса

```

```

const f = () => {
  return class extends Component {
    state = { data: null };

    componentDidMount() {
      const { getData } = this.props;

      getData()
        .then((data) => {
          this.setState({
            data
          });
        });
    }

    render() {
      const { data } = this.state;
      if (!data) return <Spinner />;

      return <ItemList {...this.props} data={data} />;
    }
  };
};

export default f();

```

```

app.js
import ItemList from '../item-list';

<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>
  { ({name}) => <span>{name}</span> }
</ItemList>

```

Единственная зависимость, которая осталась – компонент знает, что работает с `ItemList` и заточен под него. В функцию `f` можно передать аргумент, который будет являться совершенно любым компонентом вместо `ItemList`.

`F` переименована в `withData` чтобы было понятно, что она делает.

```

const withData = (View) => {
  return class extends Component {

    state = { data: null };

    componentDidMount() {
      const { getData } = this.props;

      getData()
        .then((data) => {
          this.setState({
            data
          });
        });
    }

    render() {

      const { data } = this.state;
      if (!data) return <Spinner />;
    }
  };
};

export default withData;

```

```

        return <View {...this.props} data={data} />;
    );
};

export defaultWithData(ItemList);

```

Таким образом, ItemList разделён на 2 части. Одна отвечает только за отрисовку, а вторая – за логику работы с сетью. Функция нужна для того, чтобы можно было выбирать (передавать в её параметр), какой именно компонент будет заниматься отображением данных.

Следующий шаг: поскольку itemList не содержит state, из него можно сделать компонент-функцию. Обёртка больше не получает GetData в пропсах, теперь это делается явно при экспорте. Поскольку логика и отрисовка больше не связаны друг с другом, можно раскидать их по разным файлам.

Итоговая реализация (ErrorIndicator пока не прикручен):

item-list.js

```

import React from 'react';
import SwapiService from '../../services/swapi-service';
import { withData } from '../hoc-helpers/';
import './item-list.css';

const ItemList = (props) => {

  const { data, onItemSelected, children: renderLabel } = props;

  const items = data.map((item) => {
    const {id} = item;
    const label = renderLabel(item);
    return (
      <li
        className="list-group-item"
        key={id}
        onClick={() => onItemSelected(id)}
      >
        {label}
      </li>
    );
  });

  return (
    <ul className="item-list list-group">
      {items}
    </ul>
  );
};

const { getAllPeople } = new SwapiService();

export defaultWithData(ItemList, getAllPeople);

```

with-data.js

```

import React, { Component } from 'react';
import Spinner from '../spinner';
import ErrorIndicator from '../error-indicator';

const withData = (View, getData) => {

```

```

return class extends Component {

  state = { data: null };

  componentDidMount() {
    getData()
      .then((data) => {
        this.setState({ data });
      });
  }

  render() {
    const { data } = this.state;

    if (!data) return <Spinner />

    return <View {...this.props} data={data} />;
  };
};

export defaultWithData;

```

app.js

```

<ItemList
  onItemSelected={this.onPersonSelected}
  getData={this.swapiService.getAllPeople}
>
  { ({name}) => <span>{name}</span> }
</ItemList>

```

Рефакторинг, вынос больших компонентов

Если компонент гибкий с т.з. настройки, то он начинет разрастаться в объёме и его сложно читать в общем потоке кода: очень много конфигурационных параметров. Хорошее решение – вынести такие компоненты с раздутой конфигурацией в отдельные файлы и затем импортировать их.

В папке sw-components хранятся новые компоненты: item-lists.js и details.js

Последние 2 строчки в ItemList удалить, экспортовать только сам ItemList

```

item-list.js
Было
const { getAllPeople } = new SwapiService();
export default withData(ItemList, getAllPeople);

Стало
export default ItemList;

```

item-lists.js

```

import React from 'react';
import ItemList from '../item-list';
import { withData } from '../hoc-helpers';
import SwapiService from '../../services/swapi-service';

const swapiService = new SwapiService();
const { getAllPeople, getAllStarships, getAllPlanets } = swapiService;

```

```

const PersonList = withData(ItemList, getAllPeople);
const PlanetList = withData(ItemList, getAllPlanets);
const StarshipList = withData(ItemList, getAllStarships);

export { PersonList, PlanetList, StarshipList };

```

details.js (не используется HOC)

```

import React from 'react';
import ItemDetails, { Record } from '../item-details';
import SwapiService from '../../services/swapi-service';

const swapiService = new SwapiService();
const {
  getPerson, getStarship, getPlanet,
  getPersonImage, getStarshipImage, getPlanetImage
} = swapiService;

const PersonDetails = ({ itemId }) => {
  return (
    <ItemDetails itemId={itemId} getData={getPerson} getImageUrl={getPersonImage}>
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
    </ItemDetails>
  );
};

const PlanetDetails = ({ itemId }) => {
  return (
    <ItemDetails itemId={itemId} getData={getPlanet} getImageUrl={getPlanetImage}>
      <Record field='population' label='Population' />
      <Record field='diameter' label='Diameter' />
    </ItemDetails>
  );
};

const StarshipDetails = ({ itemId }) => {
  return (
    <ItemDetails itemId={itemId} getData={getStarship} getImageUrl={getStarshipImage}>
      <Record field='model' label='Model' />
      <Record field='length' label='Length' />
    </ItemDetails>
  );
};

export { PersonDetails, PlanetDetails, StarshipDetails };

```

Использование в App.js

```

return (
  <div className='app'>
    <Header />

    <PersonDetails itemId={11} />
    <PlanetDetails itemId={5} />
    <StarshipDetails itemId={9} />

    <PersonList>
      {({name}) => <span>{name}</span> }
    </PersonList>

    <StarshipList>
      {({name}) => <span>{name}</span> }
    </StarshipList>

    <PlanetList>

```

```
{ ({name}) => <span>{name}</span> }</PlanetList>
</div>
```

Композиция компонентов высшего порядка

Композиция – применение одной функции к результату другой: $f(g(x))$.

Композиция НОС – это использование одних НОС внутри других НОС.

Сейчас в компоненты-листы всё ещё надо явно передавать `children`, в котором хранится рендер-функция. Чтобы от этого избавиться, можно использовать ещё одну функцию высшего порядка.

Функция `withChildFunction` умеет брать любой компонент и вставлять ему что-то в качестве `children`.

`Wrapped` – это компонент, который она будет оберачивать.

`fn` – функция, которую надо передать в качестве `props.children` во `Wrapped`-компонент.

Рендер-функции тоже могут быть разными.

«`renderName`» и «`renderModelAndName`» немного отличаются и выводят на экран разный текст.

```
import React from 'react';
import ItemList from '../item-list';
import { withData } from '../hoc-helpers';
import SwapiService from '../../services/swapi-service';

const swapiService = new SwapiService();
const { getAllPeople, getAllStarships, getAllPlanets } = swapiService;

// обернёт компонент и вставит ему детей
const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

// рендер-функции для персонажей и кораблей пойдут в children
const renderName = ({ name }) => <span>{name}</span>;
const renderModelAndName = ({ model, name }) => <span>{name} ({model})</span>

// композиция функций высшего порядка
const PersonList = withData(
  withChildFunction(ItemList, renderName),
  getAllPeople);

const PlanetList = withData(
  withChildFunction(ItemList, renderName),
  getAllPlanets);

const StarshipList = withData(
  withChildFunction(ItemList, renderModelAndName),
  getAllStarships);

export { PersonList, PlanetList, StarshipList };
```

Использование в app.js

```
<PersonList />
```

```
<StarshipList />
<PlanetList />
```

Теперь детали конфигурации скрыты. Использовать такие компоненты в разы легче.

Контекст API

Документация [тут](#).

Context API решает проблему проброса свойств (property drill) от компонентов верхнего уровня до компонентов нижнего.

Контекст позволяет создать специальное хранилище данных и эти данные будут доступны для всех дочерних компонентов без необходимости явно передавать эти свойства.

Гугли dependency injection.

На практике вызывается функция, которая возвращает объект, в котором хранятся создаются 2 компонента: Provider и Consumer.

Кратко:

```
import React from 'react';
```

1. создать контекст через функцию (можно сразу деструктуризовать)

```
const { Provider, Consumer } = React.createContext()
```

2. Обернуть в Provider компоненты-потребители

```
<Provider value={this.swapiService}>
  <div className='app'>
    <PersonList />
    <StarshipList />
    <PlanetList />
  </div>
</Provider>
```

3. Через Consumer вытащить переданное значение

```
const PersonDetails = () => {
  return (
    <Consumer>
      {
        (swapiService) => {
          return (
            <ItemDetails getData={swapiService.getPerson} />
          );
        }
      }
    </Consumer>
  );
};
```

createContext

```
import React from 'react';

const myContext = React.createContext(defaultValue)

const { Provider, Consumer } = React.createContext()
```

```
const { Provider : SwapiProvider, Consumer : SwapiConsumer } = React.createContext()
```

Функция возвращает объект контекста.

Этот объект можно сразу деструктуризовать на константы-компоненты Provider и Consumer.

Provider – компонент, который хранит передаваемое значение

Consumer – компонент, который будет потреблять это значение.

Аргумент default будет использован только в том случае, если у компонента нет соответствующего Provider в иерархии над ним. Если Consumer не сможет найти никакого Provider, он будет использовать значение default.

Provider

Передаёт значение из своего пропса компонентам-потребителям.

```
<MyContext.Provider value={someValue}>
```

Consumer

Считывает значение из Provider. Внутри содержит функцию, которая принимает на вход свойство-пропс из Provider, а возвращает любые компоненты, которые используют это переданное из provider свойство.

```
<ChildElement>
  <Consumer>
    {
      (lang) => {
        return (
          <Chat lang={lang} />
        )
      }
    }
  </Consumer>
</ChildElement>
```

Использование Context API

С помощью контекста элементы смогут получать сервис, а не создавать каждый для себя его инстансы.

Рефакториться будут details-элементы.

Создаётся новая директория:

```
components
  swapi-service-context
    index.js
    swapi-service-context.js
```

Весь код – это создать Provider и Consumer и экспортить их.

При создании они сразу переименовываются в swapi-поставщик и swapi-потребитель.

```
swapi-service-context.js
```

```
import React from 'react';

const {
  Provider : SwapiServiceProvider,
  Consumer: SwapiServiceConsumer
} = React.createContext();
```

```
export { SwapiServiceProvider, SwapiServiceConsumer };
```

Теперь можно пойти в компонент самого высшего уровня и использовать контекст, чтобы у всех компонентов был доступ к тому значению, которое передаём:

app.js

```
import { SwapiServiceProvider } from '../swapi-service-context/swapi-service-context';

return (
  <ErrorBoundary>
    <SwapiServiceProvider value={this.swapiService}>

      <div className='app'>
        <Header />
        <PersonDetails itemId={11} />
        <PlanetDetails itemId={5} />
        <StarshipDetails itemId={9} />
        <PersonList />
        <StarshipList />
        <PlanetList />
      </div>

    </SwapiServiceProvider>
  </ErrorBoundary>
);
});
```

Теперь в details можно получить это значение.

Импорт потребителя, обернуть в него элементы, которые ждут передаваемое значение.

Важный нюанс – потребитель передаёт значение как аргумент функции. Следовательно, он должен вернуть все обхвачиваемые компоненты.

Аргумент swapiService – это именно то значение, которое было передано в Provider по иерархии выше.

.details.js.

```
const PersonDetails = ({ itemId }) => {
  return (
    <SwapiServiceConsumer>
      {
        (swapiService) => {
          return (
            <ItemDetails
              itemId={itemId}
              getData={swapiService.getPerson}
              getImageUrl={swapiService.getPersonImage}>
            <Record field='gender' label='Gender' />
            <Record field='eyeColor' label='Eye Color' />
          </ItemDetails>
        );
      }
    </SwapiServiceConsumer>
  );
};
```

Можно сразу же деструктуризовать:

```
<SwapiServiceConsumer>
{
```

```

// (swapiService) => {
({getPerson, getPersonImage}) => {
  return (
    <ItemDetails
      itemId={itemId}
      getData={getPerson}
      getImageUrl={getPersonImage}>
    >
    <Record field='gender' label='Gender' />
    <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
);
}
}
</SwapiServiceConsumer>

```

То же самое проделать с кораблями и планетами и swapiService можно выпиливать из details.js.

Теперь для того, чтобы изменить данные, которые запрашиваются по сети, на локальные тестовые данные, достаточно сделать так:

app.js

```

import DummySwapiService from '../../services/dummy-swapi-service';
swapiService = new DummySwapiService();

```

Таким же макаром можно подменять данные, если сервис будет выдавать ошибку или долго грузиться. Сейчас этот код выглядит громоздко, но его можно улучшить с НОС.

Использование НОС для работы с контекстом

Details.js в каталоге sw-components надо разбить на 3 отдельных файла: PersonDetails, PlanetDetails, StarshipDetails.

Задача получения контекста должна быть вынесена в компонент высшего порядка.

В каталоге hoc-components создаётся компонент WithSwapiService. Вместо всего кода, который повторялся, можно использовать эту НОС-функцию.

with-swapi-service.js

```

import React from 'react';
import { SwapiServiceConsumer } from '../swapi-service-context/';

const withSwapiService = (Wrapped) => {

  return (props) => {
    return (
      <SwapiServiceConsumer>
        {
          (swapiService) => {
            return (
              <Wrapped {...props} swapiService={swapiService} />
            )
          }
        }
      </SwapiServiceConsumer>
    );
  };
};

export default withSwapiService;

```

В файле person-details.js можно удалить всё, что работает с контекстом, а экспорт обернуть в withSwapiService. Также, надо вытащить переданный НОС-ом в пропсы swapiService.

person-details.js

```
import React from 'react';
import ItemDetails, { Record } from '../item-details';
import { withSwapiService } from '../hoc-helpers';

const PersonDetails = ({ itemId, swapiService }) => {
  const { getPerson, getPersonImage } = swapiService;

  return (
    <ItemDetails
      itemId={itemId}
      getData={getPerson}
      getImageUrl={getPersonImage}>
    <Record field='gender' label='Gender' />
    <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
);
};

export default withSwapiService(PersonDetails);
```

Таким образом, НОС взял на себя функцию «выплёвывания» wrapped-компонента и передачи ему в пропсы swapiService.

Трансформация props в компонентах НОС

Вместо того, чтобы передавать весь swapiService в PersonDetails, можно туда передать исключительно те методы, которые нужны этому компоненту: getPerson и getPersonImage.

Ещё лучше – если передать getPerson под именем getData, а getPersonImage – как getImageUrl, потому что именно с такими именами работают details-компоненты.

Правила этой передачи проще всего описать функцией map.

Эта функция должна использоваться в withSwapiService в качестве второго параметра для вычисления нужных данных.

Person-details.js

```
import React from 'react';
import ItemDetails, { Record } from '../item-details';
import { withSwapiService } from '../hoc-helpers';

// было
const PersonDetails = ({ itemId, swapiService }) => {
  const { getPerson, getPersonImage } = swapiService;

  return (
    <ItemDetails
      itemId={itemId}
      getData={getPerson}
      getImageUrl={getPersonImage}>
    <Record field='gender' label='Gender' />
    <Record field='eyeColor' label='Eye Color' />
  </ItemDetails>
);
```

```

};

const PersonDetails = ({ itemId, getData, getImageUrl }) => {

  return (
    <ItemDetails
      itemId={itemId}
      getData={getData}
      getImageUrl={getImageUrl}
    >
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
    </ItemDetails>
  );
};

const mapMethodsToProps = (swapiService) => {
  return {
    getData: swapiService.getPerson,
    getImageUrl: swapiService.getPersonImage
  }
};

export default withSwapiService(PersonDetails, mapMethodsToProps);

```

Поскольку передаваемые пропсы теперь полностью соответствуют наименованию требуемых свойств, можно сократить их так:

```

Person-details.js

import React from 'react';
import ItemDetails, { Record } from '../item-details';
import { withSwapiService } from '../hoc-helpers';

const PersonDetails = (props) => {

  return (
    <ItemDetails ...props>
      <Record field='gender' label='Gender' />
      <Record field='eyeColor' label='Eye Color' />
    </ItemDetails>
  );
};

const mapMethodsToProps = (swapiService) => {
  return {
    getData: swapiService.getPerson,
    getImageUrl: swapiService.getPersonImage
  }
};

export default withSwapiService(PersonDetails, mapMethodsToProps);

```

Переделка with-swapi-service.js под функцию map:

```

with-swapi-service.js

import React from 'react';
import { SwapiServiceConsumer } from '../swapi-service-context/';

```

```

const withSwapiService = (Wrapped, mapMethodsToProps) => {

  return (props) => {
    return (
      <SwapiServiceConsumer>
        {
          (swapiService) => {
            const serviceProps = mapMethodsToProps(swapiService);

            return (
              <Wrapped {...props} {...serviceProps} />
            );
          }
        }
      </SwapiServiceConsumer>
    );
  };
};

export default withSwapiService;

```

Теперь надо обновить list-компоненты. Сейчас они используют with-data.

WithData вторым аргументом получал getData как внешний аргумент. Но withSwapiService уже умеет передавать нужные функции для запроса данных, поэтому getData больше не надо передавать в явном виде.
Вместо этого можно взять getData из props.

Было:

with-data.js

```

import React, { Component } from 'react';
import Spinner from '../spinner';

constWithData = (View, getData) => {
  return class extends Component {
    state = { data: null };

    componentDidMount() {
      getData()
        .then((data) => {
          this.setState({
            data
          });
        });
    }

    render() {
      const { data } = this.state;

      if (!data) return <Spinner />

      return <View {...this.props} data={data} />;
    }
  };
};

export default withStyles;

```

Стало:

with-data.js

```

import React, { Component } from 'react';
import Spinner from '../spinner';

constWithData = (View) => {
  return class extends Component {
    state = { data: null };

    componentDidMount() {
      this.props.getData()
        .then((data) => {
          this.setState({
            data
          });
        });
    }
  };
}

render() {
  const { data } = this.state;

  if (!data) return <Spinner />

  return <View {...this.props} data={data} />;
};

};

export defaultWithData;

```

Обновление списков

Было

```

item-lists.js

import React from 'react';
import ItemList from './item-list';
import SwapiService from '../../services/';

// удаляется полностью
// const swapiService = new SwapiService();
// const { getAllPeople, getAllStarships, getAllPlanets } = swapiService;

const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

const renderName = ({ name }) => <span>{name}</span>;
const renderModelAndName = ({ model, name }) => <span>{name} ({model})</span>

const PersonList = withData(
  withChildFunction(ItemList, renderName),
  // getAllPeople); в явном виде не надо

const PlanetList = withData(
  withChildFunction(ItemList, renderName),
  // getAllPlanets); в явном виде не надо

```

```

const StarshipList = withData(
  withChildFunction(ItemList, renderModelAndName),
  // getAllStarships); в явном виде не надо

export { PersonList, PlanetList, StarshipList };

```

Стало:

```

item-lists.js

import React from 'react';
import ItemList from '../item-list';
import { withData, withSwapiService } from '../hoc-helpers';

// обернёт компонент и вставит ему детей
const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};

// рендер-функции для персонажей и кораблей пойдут в children
const renderName = ({ name }) => <span>{name}</span>;
const renderModelAndName = ({ model, name }) => <span>{name} ({model})</span>

const mapPersonMethodsToProps = (swapiService) => {
  return { getData: swapiService.getAllPeople }
};

const mapPlanetMethodsToProps = (swapiService) => {
  return { getData: swapiService.getAllPlanets }
};

const mapStarshipMethodsToProps = (swapiService) => {
  return { getData: swapiService.getAllStarships }
};

// композиция функций высшего порядка
const PersonList = withSwapiService(
  withData(
    withChildFunction(ItemList, renderName)),
  mapPersonMethodsToProps);

const PlanetList = withSwapiService(
  withData(
    withChildFunction(ItemList, renderName)),
  mapPlanetMethodsToProps);

const StarshipList = withSwapiService(
  withData(
    withChildFunction(ItemList, renderModelAndName)),
  mapStarshipMethodsToProps);

export { PersonList, PlanetList, StarshipList };

```

Обновление контекста

Контекст не обязательно должен быть статичным, его можно обновлять «на лету». Он работает так же, как любые другие компоненты: если value обновилось, то компоненты ниже по иерархии получат обновлённое значение.

Это может пригодиться, если приложение поддерживает смену языков, темы визуального оформления и т.д.

Важно, что компоненты должны уметь правильно реагировать на изменение контекста. Обычно компоненты реагируют на изменения элементов, которые рендерятся на странице (например, новое значение строки). Надо прописать соответствующие условия в их componentDidUpdate, чтобы компоненты сравнивали предыдущие и новые функции получения (getData) данных из props. Ниже про это будет написано.

Сейчас через контекст передаётся swapiService. Если залезть в App.js, то в нём можно переписать swapi на dummySwapi. Такое переключение можно реализовать через кнопку на странице, которая будет обновлять контекст.

1. Объявить onServiceChange в app.js и передать его через пропсы в header.
2. В header.js добавить кнопку, повесить на неё onServiceChange.

Как реализовать смену значения в сервисе?

Сейчас сервис – это просто поле класса и не находится в state. Если его обновить, то React не узнает, что приложение надо перерисовать. В этой связи, swapiService надо перенести в state и обновить код, который его использует.

onServiceChange

При переключении надо знать предыдущее значение state чтобы знать, на что переключиться. Поэтому в setState передаётся функция.

```
Header.js
const Header = ({ onServiceChange }) => {
  return (
    <div className="header d-flex">
      ...
      <button onClick={onServiceChange} > Change Service </button>
    </div>
  );
}

App.js
state = {
  showRandomPlanet: true,
  swapiService: new SwapiService()
};

onServiceChange = () => {
  this.setState(({ swapiService }) => {
    const Service = swapiService instanceof SwapiService ? DummySwapiService : SwapiService;
    return { swapiService: new Service() };
  });
}

<SwapiServiceProvider value={this.state.swapiService}>
  <Header onServiceChange={this.onServiceChange} />
</SwapiServiceProvider>
```

Если всё оставить в таком виде, то service действительно будет меняться, но компоненты не будут обновляться. Причина – компоненты обновляются только тогда, когда меняется их id. Поскольку нужно сохранить старый id, но вынудить компонент обновиться, надо дописать им в update это:

Item-details.js

```

componentDidUpdate(prevProps) {
  if (this.props.itemId !== prevProps.itemId ||
    this.props.getData !== prevProps.getData ||
    this.props.getImageUrl !== prevProps.getImageUrl) {
    this.updateItem();
  }
}

```

Код сверху будет обновлять компонент, если новая функция получения данных getData или новая функция получения изображения getImageUrl не соответствуют прежним.

Компонент высшего порядка в With-data.js сейчас работает только с методом componentDidMount. Надо дописать обновления. Весь код из didMount выносится в функцию update, которая вызывается в двух методах жизненного цикла:

```

With-data.js

update() {
  this.props.getData()
    .then((data) => {
      this.setState({ data });
    });
}

componentDidMount() {
  this.update();
}

componentDidUpdate(prevProps) {
  if (this.props.getData !== prevProps.getData) {
    this.update();
  }
}

```

Рефакторинг НОС-компонентов.

Упаковка <Row />, -list и -details компонентов в обобщённые -page компоненты.

Лишние импорты и поля класса в app.js удаляются.

Директория People-page удаляется, т.к. нигде не используется.

Создаётся новая директория pages.

Ранее в App использовались Row, StarshipDetails, StarshipsList и т.д., но теперь их заменяют соответствующие компоненты pages:

```

People-page.js

import React, { Component } from 'react';
import { PersonDetails, PersonList } from '../sw-components';
import Row from '../row';

export default class PeoplePage extends Component {

  state = {
    selectedItem: null
  };

  onItemSelected = (selectedItem) => {

```

```
    this.setState({ selectedItem });
};

render() {
  const { selectedItem } = this.state;

  return (
    <Row
      left={<PersonList onItemSelected={this.onItemSelected} />}
      right={<PersonDetails itemId={selectedItem} />}
    />
  );
};

};


```

App.js

Было

```
<Row
  left={<StarshipList />}
  right={<StarshipDetails itemId={9} />}
/>
```

Стало

```
<PeoplePage />
<StarshipsPage />
<PlanetsPage />
```

+ правки в with-data.js

Функция compose()

В этом уроке автор что-то намудрил. Если следовать его указаниям, то приложение ломается. Чтобы всё работало, надо переписать много компонентов, которые он оставил за кадром. Решения использовать функцию compose() особой погоды не делает, поэтому просто запишу принцип работы, а своё приложение переписывать не буду.

Сейчас файл с НОС-компонентами item-lists.js выглядит запутанно и непонятно. Чтобы избавиться от цепочки вложенных функций, можно создать одну функцию compose(), которая будет принимать массив функций, которые должны поочерёдно передавать результат друг другу, и также компонент, который надо пропустить через эти функции.

Функцию compose можно было бы организовать так:

1. передать несколько других функций, композицию которых надо получить;
2. передать компонент, который через эти функции должен пройти.

```
compose(
  withSwapiService(mapMethodsToProps),
  withData,
  withChildFunction(renderModelAndName)
)(itemList);
```

В общих чертах выглядеть будет так

```
const compose = (...functions) => (component) => {...}
```

Эти функции будут в итоге работать одинаково

```
compose(a, b, c)(value) == a(b(c(value)));
```

Этот код читать легче (нет): видно, во что компонент оборачивается.

Можно пройтись по массиву функций справа налево, вызвать каждую, передать её результат в следующую функцию. Но это не лучшее решение (Дима так говорит в своём курсе).

В js есть функция reduceRight (именно о ней сразу и подумал).

Для этой функции создаётся отдельный файл в HOC-helpers

```
compose.js

const compose = (...functions) => (component) => {
  return functions.reduceRight(
    (prevResult, foo) => foo(prevResult), component);
};

export default compose;
```

Функция withChildFunction тоже переезжает из item-lists.js в директорию HOC-helpers и немного изменяется:

with-child-function.js

было

```
const withChildFunction = (Wrapped, fn) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    );
  }
};
```

стало:

```
const withChildFunction = (fn) => (Wrapped) => {
  return (props) => {
    return (
      <Wrapped {...props}>
        {fn}
      </Wrapped>
    )
  };
}

export default withChildFunction;
```

defaultProps

Работает так же, как значения по умолчанию параметров функции, только устанавливает значения по умолчанию для свойств (пропсов) компонентов.

defaultProps можно записать как свойство созданного компонента. Эта запись происходит вне тела определения. Так надо записывать defaultProps для компонентов-функций:

```
// как свойство компонента
ComponentName.defaultProps = {
  onItemSelected: () => {}
};
```

Для компонентов-классов defaultProps можно записывать как статическое свойство всего класса:

```
class MyComponent extends Component {  
  static defaultProps = {  
    props: value  
  }  
}
```

Теперь в случае, если в пропсы компонентов не передано значение или это значение – `undefined`, будет использовано дефолтное значение. В этом свойстве-объекте можно перечислить сколько угодно свойств, которые компонент потребляет из пропсов. Важно, что значение `null` будет обрабатываться нормально и `defaultProps` использоваться не будет.

Например, `itemList` рассчитывает получить функцию-обработчик `onItemSelected` из `people-page.js`. Может возникнуть ситуация, когда надо будет просто отобразить список чего-то и никак не реагировать на клики. Один из вариантов (не лучший) – прописать пустую функцию в деструктуризации. Если ничего не передать в `props`, то в обработчик присвоится пустая функция:

```
const { onItemSelected = () => {} } = props;
```

Намного проще будет установить значение по умолчанию. Теперь если функция `onItemSelected` не будет передана вышестоящим компонентом, то будет использоваться значение по умолчанию.

```
const ItemList = (props) => {  
  // определение компонента  
};  
  
ItemList.defaultProps = {  
  onItemSelected: () => {}  
};
```

Ещё пример – компонент `random-planet.js`

Интервал обновления сейчас прописывается в самом компоненте – в функции-таймере. Можно сделать так, чтобы время обновления передавалось через пропсы, заодно установить `defaultProps`:

```
random-planet.js  
  
static defaultProps = {  
  updateInterval: 10000  
}  
  
componentDidMount() {  
  const { updateInterval } = this.props;  
  this.updatePlanet();  
  this.interval = setInterval(() => this.updatePlanet(), updateInterval);  
}
```

propTypes

Поскольку JS – это язык с динамической типизацией, переменные могут менять тип хранимых данных во время жизненного цикла, а отлов ошибок, связанных с типами, откладывается до момента запуска приложения.

TS добавляет строгую типизацию, но это «громоздкая» надстройка, не очень удобно использовать. В React есть способ указать типы для свойств компонентов. Свойства начнут проверяться перед тем, как компонент их получит и начнёт работать. Проверка `propTypes` срабатывает после `defaultProps`.

```
random-planet.js
```

```
static propTypes = {  
  propName: (props, propName, componentName) => {  
    const value = props[propName];  
  
    if (typeof value === 'number' && !isNaN(value)) {  
      return null;  
    }  
  
    return new TypeError(`#${componentName}: ${propName} must be number`);  
  }  
}  
}
```

props это весь объект свойств (пропсов) компонента.
propName имя того свойства, для которого сейчас проводится валидация.
componentName название компонента, для которого проводится валидация

Если все проверки прошли успешно, надо вернуть null.
Если нет – возвращаем (не выбрасываем) ошибку.

prop-types библиотека

Чтобы не писать код проверки вручную, как в примере выше, есть несколько библиотек, которые реализуют правило валидации.

Документация [здесь](#).

Библиотека с большими возможностями есть у Airbnb, называется тоже prop-types и лежит на GitHub. В ней множество разных валидаций, более сложная логика.

Установка:

```
npm install prop-types
```

Использование:

```
import PropTypes from 'prop-types';  
import PropTypes from 'prop-types'; // с функциональными компонентами
```

```
static propTypes = {  
  propName: PropTypes.number  
}
```

```
// если свойство необходимо передать явно  
static propTypes = {  
  propName: PropTypes.number.isRequired  
}
```

Поскольку defaultProps срабатывает раньше, то .isRequired нужно указывать по больше части для разработчика (самодокументирование и всё такое).

Кроме простых пропсов с примитивными типами, можно описывать функции или массивы.
Например, в компоненте ItemList:

```
item-list.js
```

```
ItemList.propTypes = {
```

```
onItemSelected: PropTypes.func,
data: PropTypes.arrayOf(PropTypes.object.isRequired), // какого типа массив
children: PropTypes.func.isRequired
};
```

row.js

```
Row.propTypes = {
  left: PropTypes.element, // только react-элемент
  right: PropTypes.node // более универсальный, всё что можно отрендерить
};
```

Можно описать объект с определённой структурой.

Если есть компонент, который должен получать в качестве пропса объект, у которого есть поля user и role:

```
MyComponent.propTypes = {

  user: PropTypes.shape({
    name: PropTypes.string,
    role: PropTypes.oneOf(['user', 'admin']) // одно из значений
  })
}
```

Router

Роутинг – это переключение между «виртуальными страницами» UI приложения.

Гит роутера [здесь](#).

В обычных приложениях переход между страницами влечёт их полную перезагрузку. В SPA, при переходе по внутренним ссылкам, переход на новые страницы не происходит: страница остаётся прежней, но компоненты этой страницы отображаются выборочно в зависимости от URL.

Такой подход называется Роутинг. Роутинг в контексте SPA – это функционал, который позволяет «переключать» страницы. Роутер – компонент, который, зная id страницы, отображает соответствующий компонент на экране.

Таким образом, в контексте роутинга понятие страниц и переходов – это «виртуальное» понятие. На самом деле, происходит скрытие одних компонентов и показ других в рамках одной и той же страницы браузера.

Библиотека React Router делает 2 вещи:

- читает url и выбирает, какие компоненты нужно отобразить пользователю;
- обновляет url, когда пользователь переходит на новую страницу.

Установка

```
npm install react-router-dom
версия для браузера
```

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
```

В тег BrowserRouter оборачивается всё, что возвращает компонент App.

В тег Routes оборачиваются Route со страницами

```
const App = () => {
  return (

```

```

<BrowserRouter>
  <div className='app'>
    <Header/>

    <Routes>
      <Route path="/" element={<Main />} />
      <Route path="/first" element={<PageOne />} />
      <Route path='/second' element={<PageOne />} />
    </Routes>
  </div>
</BrowserRouter>
);
}

```

Link

Компонент Link используется вместо тега `<a>` для перехода по ссылкам (для переключения страниц). Если оставить тег `<a>`, то браузер будет перезагружать страницы как обычно.

Поскольку в SPA страницы в действительности не перезагружаются, компонент Link использует history API браузера, чтобы обновить адрес в строке, но при этом не обновлять страницу и не перезагружать приложение.

Внутри этого компонента находится тот же `href`, только он дополнительно обрабатывается так, чтобы страница браузера не перезагружалась.

```

import { Link } from 'react-router-dom';

<header className='header'>

  <Link to='/'>Главная</Link>
  <Link to='/first'>Первая</Link>
  <Link to='/second'>Вторая</Link>

</header>

```

`</>`

Redux

Общее

Redux – это независимая библиотека для управления глобальным state всего приложения.

Это паттерн и библиотека.

Решает проблему управления состоянием в приложении путём хранения state в одном «глобальном» объекте.

Управление реализуется через «actions», которые сообщают, какое действие по изменению state надо выполнить и, если нужно, предоставляют необходимые для изменения данные.

Redux рекомендуется к использованию, когда:

- приложение использует «большой» state
- приложение часто обновляет state
- логика обновления state сложная
- в приложении много кода и над ним работают много людей

Документация [здесь](#).

Redux Fundamentals [здесь](#).

Документация по React-Redux [здесь](#).

Пример и описание минимального приложения [здесь](#).

Годно про Redux на русском: rajdee.gitbooks.io

Главные элементы:

Store – объект, который хранит в себе глобальный state и координирует обновления (отвечает за логику изменения state).

Reducer – функция, которая занимается обновлением глобального state в ответ на Actions (события в redux называются действия).

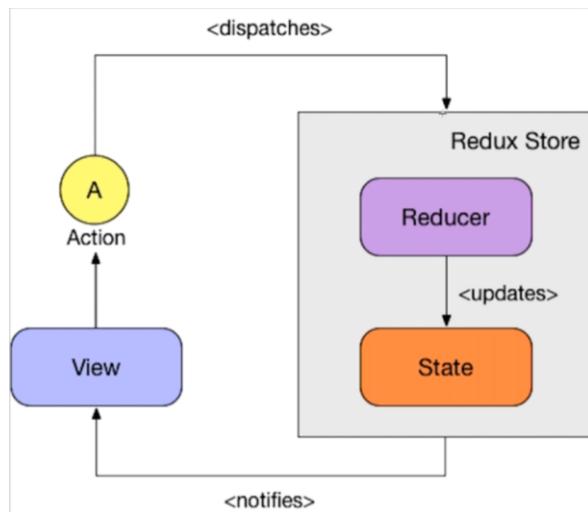
Action – обычный JS-объект, который описывает, что именно надо сделать. Reducer его получает, изменяет state и ui перерендеривается.

Dispatch – операция по изменению state.

Компоненты могут только читать значения из глобального State, но не писать в него (непосредственно).

Для того, чтобы изменить глобальный State, мы создаём объекты Action и передаём их в Store. Эта операция называется dispatch.

Store использует функцию, которая называется Reducer, чтобы обновить глобальный State. Reducer реагирует на actions и обновляет State.



Установка библиотек

Redux – оригинальная небольшая JS-библиотека

React-Redux – библиотека для удобной интеграции с React.

Редакс обычно используется с плагинами, которые указаны [тут](#).

Расширение для браузеров для отладки: Redux DevTools Extension [здесь](#).

Redux Toolkit – набор пакетов, который рекомендуют создатели [здесь](#).

Установка библиотек

```
npx create-react-app redux-sandbox  
npm install redux react-redux
```

Подключение к чистому HTML

```
<script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
```

Reducer

Функция, которая умеет реагировать на actions и обновлять state.

Словарь [здесь](#).

Структура редьюсеров [здесь](#).

```
const reducer = (state = initialState, action) => newState
```

```
const store = createStore(reducer);
```

state

Текущее состояние.

Он прописывается только один раз при создании функции reducer (указывается его первичное значение), а дальше redux использует его автоматом.

action

Это обычный объект, который должен содержать свойство .type.

Значение свойства type – это читаемая строка, которая описывает действие, которое надо совершить со state.

Логика работы:

Функция reducer читает action.type, выбирает функцию для его обработки, совершает необходимые действия и возвращает новый state. Старый state нельзя мутировать.

Если type неизвестен, то reducer должен вернуть state без изменений.

1. Проверить, работает ли reducer с переданным action.

2. Если да – сделать копию state, внести изменения и вернуть эту изменённую копию.

3. Если нет – вернуть исходный state без изменений.

Правила:

Reducer – это всегда чистая функция:

1. Reducer работает только с данными из аргументов state и action. Извне данные не берутся.

2. Существующий state не мутируется, возвращается его изменённая копия.

3. Не должно быть асинхронности, рандомных значений, таймеров, записи значений в localStore, запросов на сервер и т.д.

Пример из курса:

```
const initialState = 0

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'RND':
      return state + action.payload;

    case 'INC':
      return state + 1;

    case 'DEC':
      return state - 1;

    default:
      return state;
  }
};
```

Пример из руководства Redux:

```
const initialState = { value: 0 }

function reducer(state = initialState, action) {

  if (action.type === 'counter/incremented') {
    return { ...state, value: state.value + 1 }
  }

  return state
}
```

action

Документация [здесь](#).

Действия – это простой JS объект с обязательным полем type, которые отправляются в reducer в целях изменения state. Свойства .type могут быть строками или символами, но строки предпочтительнее, потому что их можно сериализовать.

Функция reducer читает action.type, выбирает функцию для его обработки, совершают необходимые действия и возвращает новый state.

Любые данные типа событий интерфейса, сетевые запросы и др. должны в конечном итоге отправляться как действия (dispatched as actions). Через action можно передавать в reducer случайные числа и всё такое.

Кроме свойства type, action также может содержать любые другие дополнительные свойства (payload).

action creator

Функция, которая возвращает объект action.

простой объект

```
const action = {type: 'INC'}
```

как функция action creator

```
const action = () => ({type: 'INC'})
```

с дополнительными параметрами

```
const rnd = (payload) => ({type: 'RND', payload});
```

отправить action

```
store.dispatch(action)
```

Как правило, их складывают в отдельный файл, потому что со временем их набирается оч много.

```
actions.js
export const inc = () => ({type: 'INC'});
export const dec = () => ({type: 'DEC'});
export const rnd = (payload) => ({type: 'RND', payload});
```

Store

После создания функции reducer, создаётся store.

Store API [здесь](#).

The Redux Store - центральный элемент каждого приложения, который хранит глобальный state. Это обычный js-объект с некоторыми специфичными функциями.

Правила:

- нельзя менять state внутри store напрямую.
- единственный способ поменять state – создать объект action, в котором записано, что именно в приложении произошло, а затем отправить (dispatch) этот action в store.
- когда action отправлено, Store запускает функцию reducer. Эта функция создаёт новый state с использованием action.
- Store уведомляет подписчиков, что state обновился и UI может быть обновлён новыми данными.

Store создаётся через библиотеку Redux.

Предварительно должна быть готова функция reducer, потому что она передаётся в store при создании.

Создание

```
import { createStore } from 'redux';
const store = Redux.createStore(reducer)
```

Методы Store

store.getState()

возвращает текущий state

store.dispatch(action)

Внести изменения в store

(store вызовет reducer с указанным action)

Возвращает обновлённый state. Подписчики также уведомляются об обновлении.

store.subscribe(listener)

Подписывает функцию-коллбэк на «прослушку» state.

listener – это коллбэк, который будет вызываться каждый раз, когда обновляется state.

store.unsubscribe

Не понял, как работает.

To unsubscribe the change listener, invoke the function returned by subscribe.

store.replaceReducer(nextReducer)

Replaces the reducer currently used by the store to calculate the state.

createStore принимает в качестве аргумента функцию reducer.

С помощью этой функции в store создаётся initial state. Именно поэтому функция должна иметь аргумент по умолчанию.

Store будет использовать переданный reducer для дальнейших обновлений.

.dispatch:

При вызове dispatch, store вызовет reducer с указанным action.

Можно написать код, который будет диспатчить только если определённые условия true.

Или асинхронный код, который будет диспатчить после задержки.

Можно передавать .dispatch внутри функций-коллбэков addEventListener.

```
store.dispatch({type: 'INC'});
```

UI

Redux может работать с любым ui, это самостоятельная, независимая библиотека.

Так, например, redux спокойно работает (и обновляет) нативный HTML. Для этого его надо подключить в header вот так:

```
<script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
```

В примере используются функции getState и subscribe.

```
const valueEl = document.getElementById('value')

// Whenever the store state changes, update the UI by
// reading the latest store state and showing new data
function render() {
  const state = store.getState()
  valueEl.innerHTML = state.value.toString()
}

// Update the UI with the initial data
render()
```

```
// And subscribe to redraw whenever the data changes in the future
store.subscribe(render)
```

Как только state будет меняться, redux обновит интерфейс:

- Создаётся функция render.
- Внутри она считывает значение из state и вставляет его в html-элемент.
- Функция render подписывается на изменения в store и вызывается каждый раз, когда эти изменения происходят.

Redux пошагово

Импорт

```
import { createStore } from 'redux';
```

Создать reducer

```
const initialState = { value: 0 }

function reducer(state = initialState, action) {
  if (action.type === 'counter/incremented') {
    return { ...state, value: state.value + 1 }
  }

  return state
}
```

Создать store, передать в него reducer,

Смотреть текущее состояние:

```
const store = createStore(reducer);
store.getState()

store.getState()
store.dispatch(action)
store.subscribe(listener)
store.unsubscribe
store.replaceReducer(nextReducer)
```

Создать action и диспатчить их в store:

```
const action = {type: 'INC'};

const rnd = (payload) => ({type: 'RND', payload});

store.dispatch(action);
```

Подписать store на обновления.

store.subscribe позволяет получать нотификации, когда store изменился (при диспатче).

Аргументом передаётся функция, которая будет запускаться, когда store изменился.

```
// при изменении store будет выводиться печать его состояния
store.subscribe( () => {
```

```
    console.log(store.getState());
});
```

Раскидать actions и reducer по отдельным файлам.

Обернуть actionCreator в bindActionCreators

index.js

```
import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import * as actions from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );

document.getElementById('inc').addEventListener('click', inc);
document.getElementById('dec').addEventListener('click', dec);

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  rnd(payload);
});

const update = () => {
  const value = store.getState();
  document.getElementById('counter').innerHTML = value;
}

store.subscribe(update);
```

Паттерны

Страница в документации с паттернами [тут](#).

React-Redux паттерны [тут](#).

bindActionCreators

Документация [здесь](#).

Используется для того, чтобы избавиться от постоянного store.dispatch(action) в коде.

Далее в примерах будут использоваться два action creators:

```
export const inc = () => ({type: 'INC'});
export const rnd = (payload) => ({type: 'RND', payload});
```

Чтобы сократить код и не дублировать действия, можно деструктурировать dispatch из store:

было
store.dispatch(action)

стало
const store = createStore(reducer);
const {dispatch} = store;

dispatch(action)

Следующее улучшение. Каждый раз, когда обрабатывается событие, сначала создаётся action, а потом – передача его в dispatch. Этот процесс повторяется каждый раз, когда надо что-то отправить. Можно создать одну функцию, которая будет выполнять оба этих действия:

```
было
document
  .getElementById('inc')
  .addEventListener('click', () => {
    dispatch(inc())
  });

стало
const store = createStore(reducer);
const {dispatch} = store;
const incDispatch = () => dispatch(inc());

document
  .getElementById('inc')
  .addEventListener('click', incDispatch);
```

Создавать вручную такие action не удобно. Можно написать свою функцию, которая будет оберачивать каждый actionCreator в dispatch.

```
было
const incDispatch = () => dispatch(inc());
const rndDispatch = (payload) => dispatch(rnd(payload));

стало
const bindActionCreator = (creator, dispatch) => (...args) => {
  dispatch(creator(...args));
}

...args нужен на тот случай, если передаются payload. Функция будет одинаково хорошо работать как с ас без аргументов, так и с аргументами.
Например, в функцию со случайнм числом, которой нужны payload, аргументы будут передаваться автоматом:

const incDispatch = bindActionCreator(inc, dispatch);
const rndDispatch = bindActionCreator(rnd, dispatch);
```

bindActionCreators

Такое связывание action creator с функцией dispatch является стандартным паттерном в redux. Для этого не надо писать свою функцию. Специальный метод bindActionCreators используется для того, чтобы обернуть каждый actionCreator в dispatch. В качестве результата он вернёт объект, свойством которого являются обёрнутые функции, а ключи – названия этих обёрток.

Важно! Не забыть импортировать:

```
import { createStore, bindActionCreators } from 'redux';
```

Можно использовать так же, как самописную функцию из примера выше, всё будет работать:

```
const incDispatch = bindActionCreators(inc, dispatch);
const rndDispatch = bindActionCreators(rnd, dispatch);
```

bindActionCreators позволяет обернуть сразу несколько action-creator.

Для этого, вместо первого аргумента, нужно передать объект. Ключи этого объекта – название новых функций-обёрток, которые хотим получить, а значения – актуальные action creators.

Функция вернёт объект с ключами, которые имеют такие же названия, как новые функции-обёртки.

```
было
const incDispatch = bindActionCreators(inc, dispatch);
const rndDispatch = bindActionCreators(rnd, dispatch);

стало
import { createStore, bindActionCreators } from 'redux';
import { inc, rnd } from './actions';

const actionCreatorsObject = bindActionCreators({
  incDispatch: inc,
  rndDispatch: rnd,
}, dispatch);

actionCreatorsObject.inc()
actionCreatorsObject.rnd()
```

Можно сразу деструктуризовать этот объект:

```
const {incDispatch, decDispatch, rndDispatch} = bindActionCreators({
  incDispatch: inc,
  decDispatch: dec,
  rndDispatch: rnd,
}, dispatch);
```

Можно ещё сильнее сократить этот код:

```
import * as actions from './actions';
импортировать всё из файла

const {inc, dec, rnd} = bindActionCreators(actions, dispatch);
```

Разнести actions и reducer по отдельный файлам, а затем применить bondActionCreators – хорошая практика, потому что теперь компоненты, которые отвечают за визуальную часть, зависят только от функций {inc, dec, rnd} и ничего не знают о том, как работает их логика.

Более того, приложение можно переписать без использования redux: просто передать другие функции.

Маленькие компоненты с небольшим количеством зависимостей – залог чистого кода, который легко поддерживать.

Юрий Бура

Reducer

Функция, которая умеет реагировать на actions и обновлять state.

Словарь [здесь](#).

Структура редьюсеров [здесь](#).

Создание

```
const reducer = (state, action) => newState
```

state

текущее состояние

action

обычный JS-объект, в котором свойство type описывает действие, которое нужно совершить.

Кроме обязательного свойства `type`, объект `action` также может содержать любые другие дополнительные свойства. Обычно поле с доп. параметрами называется `payload`.

state

этот аргумент – текущее состояние, тут всё понятно.

Если `state` – `undefined`, то нужно вернуть первоначальный (initial) state.

action

Это обычный объект, который обязательно должен содержать свойство `type`.

С помощью свойства `type` функция `reducer` будет понимать, что именно нужно сделать. Это как «имя» для действия, которое надо совершить. Например, увеличить счётчик (`action.type === 'INC'`).

Если тип `action` неизвестен – нужно вернуть `state` без изменений.

Кроме свойства `type`, объект `action` может содержать почти любую другую вспомогательную информацию.

Правила:

Если `state` – `undefined`, то нужно вернуть первоначальный (initial) state.

Если тип `action` неизвестен – нужно вернуть `state` без изменений.

Reducer должна быть чистой функцией:

1. Возвращаемое значение зависит только от аргументов. Аргументы – это текущий `state` и `action`. Результат работы функции `reducer` – новый `state`. Он должен зависеть исключительно от старого `state` и `action` и не от каких других параметров, которые берутся извне.
2. У `reducer` не может быть побочных эффектов: таймеров, записи значений в `localStorage`, запросов на сервер и т.д. Это помогает в крупных приложениях.

Пример:

```
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'RND':
      return state + action.payload;

    case 'INC':
      return state + 1;

    case 'DEC':
      return state - 1;

    default:
      return state;
  }
};
```

Redux Store

Это центральный объект в системе Redux. Его задача – координировать работу других компонентов.

Центральное место в нём занимает `reducer`.

Что касается `state`, то его можно получить из функции `reducer`, как в примере выше.

Документация по `Store API` [здесь](#).

`Store` содержит внутри себя `state tree`. Чтобы изменить `state`, нужно отправить `action`.

Это обычный объект с несколькими методами, не класс.

Создание `store` – это передача `reduce`-функции в функцию `createStore()`.

Создать `Store`

```
import { createStore } from 'redux';
const store = createStore(reducer);
```

Методы Store

.**getState()**

возвращает текущий state

.**dispatch(action)**

Обработать новый action и внести изменения в store.

action – это всегда объект типа {type: 'ACTION'}

.**subscribe(listener)**

Получать нотификации об изменениях store.

Change listener. It will be called any time an action is dispatched.

.**replaceReducer(nextReducer)**

Пример

```
// создание
const store = createStore(reducer);

// будет срабатывать при изменении store
store.subscribe(() => {
  console.log(store.getState());
});

// изменение store
store.dispatch({type: 'INC'});
store.dispatch({type: 'INC'});
```

UI для Redux

В качестве UI может использоваться любая библиотека или фреймворк.

Например:

HTML

```
<h2 id="counter">0</h2>
<button id="dec">DEC</button>
<button id="inc">INC</button>
```

JS

```
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    case 'RND': return state + action.payload;

    default: return state;
  };
};

const store = createStore(reducer);
```

```
document
```

```
  .getElementById('inc')
  .addEventListener('click', () => {
    store.dispatch({type: 'INC'});
  });


```

```
document
```

```
  .getElementById('dec')
  .addEventListener('click', () => {
    store.dispatch({type: 'DEC'});
  });


```

```

document
  .getElementById('rnd')
  .addEventListener('click', () => {
    const payload = Math.floor(Math.random() * 10);
    store.dispatch({ type: 'RND', payload });
});

const update = () => {
  document
    .getElementById('counter')
    .innerHTML = store.getState();
}

store.subscribe(update);

```

Передача payload

Если нужно передать случайные данные в reducer, то это делается не в reducer, а где угодно снаружи.

В примере из курса функция создаёт значение до action-объекта, а потом добавляет в него готовое значение.

```

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  store.dispatch({ type: 'RND', payload });
});

```

Action creator

Объекты action – это интерфейс общения между компонентами приложения и объектом store. Они используются для изменения состояния. В большом приложении экшены проще создавать через функции.

Функции, создающие action-объекты, называются action creator.

Самый элементарный способ их создать:

```

const inc = () => ( {type: 'INC'} );
const dec = () => ( {type: 'DEC'} );
const rnd = (payload) => ( {type: 'RND', payload} );

document.getElementById('inc').addEventListener('click', () => {
  store.dispatch( inc() );
});

document.getElementById('dec').addEventListener('click', () => {
  store.dispatch( dec() );
});

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  store.dispatch( rnd(payload) );
});

```

Структура проекта

AC и ред выносятся в отдельные файлы.

Они не зависят ни от редактора, ни от других библиотек.

```

actions.js
export const inc = () => ( {type: 'INC'} );
export const dec = () => ( {type: 'DEC'} );
export const rnd = (payload) => ( {type: 'RND', payload} );

reducer.js
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    case 'RND': return state + action.payload;

    default: return state;
  };
};

export default reducer;

index.js
import { createStore } from 'redux';
import reducer from './reducer';
import {inc, dec, rnd} from './actions';

```

bindActionCreators

чтобы постоянно не использовать store.dispatch, можно вынести эту функцию:

```

было:
store.dispatch( inc() );

стало:
const store = createStore(reducer);
const { dispatch } = store;

document.getElementById('inc').addEventListener('click', () => {
  dispatch( inc() );
});

```

В каждой функции повторяется один и тот же процесс: dispatch(actionCreator()).

Его можно сократить, сделав композиция функций вручную:

```

было:
dispatch( inc() );

стало:
import {inc, dec, rnd} from './actions';
const store = createStore(reducer);
const { dispatch } = store;

const incDispatch = () => dispatch(inc());
const decDispatch = () => dispatch(dec());
const rndDispatch = (payload) => dispatch(rnd(payload));

document.getElementById('inc').addEventListener('click', incDispatch);
document.getElementById('dec').addEventListener('click', decDispatch);

document.getElementById('rnd').addEventListener('click', () => {

```

```
const payload = Math.floor(Math.random() * 10);
rndDispatch(payload);
});
```

Если приложение будет большое, то все функции не обернёшь вручную в dispatch.
Можно написать функцию для этого. Она принимает 2 параметра: ac и dispatch, и возвращает функцию, которая их оборачивает.

Некоторым ac нужен payload и заранее неизвестно, сколько аргументов хочет получить ac. Поэтому во второй возвращаемой функции надо использовать rest и spread операторы.

было:

```
const incDispatch = () => dispatch(inc());
const decDispatch = () => dispatch(dec());
const rndDispatch = (payload) => dispatch(rnd(payload));
```

стало:

```
import {inc, dec, rnd} from './actions';
const store = createStore(reducer);
const { dispatch } = store;

const bindActionCreator = (actionCreator, dispatch) => (...args) => {
  dispatch(actionCreator(...args));
}

const incDispatch = bindActionCreator(inc, dispatch);
const decDispatch = bindActionCreator(dec, dispatch);
const rndDispatch = bindActionCreator(rnd, dispatch);
```

Использование точно такое же, как и раньше (см. выше).

bindActionCreators из Redux

Связывание ac и диспатч – это типичный паттерн для redux, поэтому в redux есть своя версия такой функции. Её можно импортировать и использовать так же, как и свою рукописную:

```
import { createStore, bindActionCreators } from 'redux';

const incDispatch = bindActionCreators(inc, dispatch);
const decDispatch = bindActionCreators(dec, dispatch);
const rndDispatch = bindActionCreators(rnd, dispatch);
```

bindAC позволяет оборачивать за один раз много функций. В него можно передать объект первым параметром, а вторым – dispatch.

Ключи – название функций, которые будут использоваться.

Значения – название ac, которые надо обернуть. Всё как при деструктуризации.

Если передать в эту функцию объект из ac, то они автоматически будут обёрнуты в dispatch.

В качестве результата bindAC вернёт объект с такой же структурой, как описана выше:

```
import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import {inc, dec, rnd} from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const bindObject = bindActionCreators(
```

```
{ incDispatch: inc, decDispatch: dec, rndDispatch: rnd },
dispatch
);

// использование
bindObject.incDispatch;
bindObject.decDispatch;
bindObject.rnd(payload);
```

Можно сразу деструктуризовать возвращаемый объект и использовать готовые функции по отдельности:

```
const {incDispatch, decDispatch, rndDispatch} = bindActionCreators(
{ incDispatch: inc, decDispatch: dec, rndDispatch: rnd },
dispatch
);

// использование
incDispatch;
decDispatch;
rndDispatch(payload);
```

Этот код можно ещё улучшить, если сделать импорт вот так:

```
import * as actions from './actions';

console.log( actions ) // Object { inc: Getter, dec: Getter, rnd: Getter }

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );
```

Итоговый код (чтобы не искать в другом месте):

```
index.js

import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import * as actions from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );

document.getElementById('inc').addEventListener('click', inc);
document.getElementById('dec').addEventListener('click', dec);

document.getElementById('rnd').addEventListener('click', () => {
  const payload = Math.floor(Math.random() * 10);
  rnd(payload);
});

const update = () => {
  const value = store.getState();
  document.getElementById('counter').innerHTML = value;
}

store.subscribe(update);
```

Использование React и Redux

Вместо базового DOM API будет использоваться React для отрисовки.

В этом уроке не используется react-redux, только ручное совмещение.

Использование чистого redux (не рекомендуется)

Сейчас в index.js есть 2 части:

- которая работает с redux
- которая работает с dom-деревом (всё, что начинается с document...)

Создаётся react-компонент для отрисовки, куда будут передаваться все нужные функции для отрисовки и вычисления значений. Поскольку компонент не обладает state (вместо него используется store), компонент будет обновляться через подписку store.subscribe(update).

Первый запуск функции update для отрисовки делается вручную. Повторные обновления делает store благодаря подписке.

React-компоненты должны как можно меньше использовать внутри себя библиотеку redux. В идеале - вообще ничего о нём не знать, только получать значения извне.

В примере ниже компонент не знает, как именно он будет обновляться. Он только отображает какое-то значение counter и при клике умеет вызывать функции-диспаччи.

Создаётся файл counter.js с компонентом:

```
counter.js
import React from 'react';

const Counter = ({ counter, inc, dec, rnd }) => {
  return (
    <div id="root" className="jumbotron">
      <h2 id="counter">{counter}</h2>

      <button
        onClick={dec}
        id="dec" className="btn btn-primary btn-lg">DEC</button>

      <button
        onClick={inc}
        id="inc" className="btn btn-primary btn-lg">INC</button>

      <button
        onClick={rnd}
        id="rnd" className="btn btn-primary btn-lg">RND</button>
    </div>
  );
};

export default Counter;
```

Файл index.js

При клике на кнопки, компонент будет вызывать функции-диспаччи, которые поменяют store.

Функция update подписана на изменение store и через ReactDOM.render каждый раз при обновлении store добавляет на страницу обновлённый компонент. На самом деле, react будет использовать свой reconciliation алгоритм и обновлять только изменённые элементы.

Первый раз update надо вызвать вручную.

```
index.js
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import Counter from './counter';
```

```

import { createStore, bindActionCreators } from 'redux';
import reducer from './reducer';
import * as actions from './actions';

const store = createStore(reducer);
const { dispatch } = store;

const {inc, dec, rnd} = bindActionCreators( actions, dispatch );

const update = () => {

  const value = store.getState();
  const payload = Math.floor(Math.random()*10);

  ReactDOM.render(
    <Counter
      counter={value}
      inc={inc}
      dec={dec}
      rnd={ () => (rnd(payload)) }
    />,
    document.getElementById('root')
  );
}

store.subscribe(update);
update();

```

Другие файлы (чтобы не искать):

```

reducer.js
const reducer = (state = 0, action) => {
  switch (action.type) {
    case 'INC': return state + 1;
    case 'DEC': return state - 1;
    case 'RND': return state + action.payload;
    default: return state;
  };
};

export default reducer;

actions.js
export const inc = () => ({type: 'INC'});
export const dec = () => ({type: 'DEC'});
export const rnd = (payload) => ({type: 'RND', payload});

```

Генерацию случайных чисел и значения value можно прописать прям в пропсах. Я вынес для удобства.

React-redux

В этом уроке используется react-redux.

Документация [здесь](#).

Кратко

```

index.js
import { createStore } from 'redux';

```

```

import { Provider } from 'react-redux';
import reducer from './reducer';

const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);

component.js
import { connect } from 'react-redux';
const MyComponent = () => {...};

export default connect(mapStateToProps, mapDispatchToProps)(Component)

```

Подробно

Для начала, надо как обычно переместить компоненты в components и всё организовать как обычно:

```

actions.js
reducer.js
index.js

components
  app.js
  counter.js

```

<Provider store={}>

Центральный элемент, вокруг которого всё происходит в Redux – это store. Он должен быть доступен для всех элементов одновременно.

Чтобы сделать его доступным, его нужно разместить в index.js и использовать контекст api. Библиотека react-redux предоставляет готовый инструмент, который основан на контексте и упрощает интеграцию, самому ничего писать не надо. Все детали интеграции, которые раньше приходилось писать, уже реализованы в нём, достаточно просто один раз отрендерить приложение.

Что делает:

1. передаёт store по иерархии

Компонент Provider основан на контексте и может передавать store по всей иерархии приложения.

2. автоматом обновляет приложение при изменении store.

Как только любой компонент из иерархии ниже вызовет dispatch, компонент Provider узнает об этом и обновит приложение.

Внутри себя он уже реализует подписку на изменение store и делает так, чтобы всё приложение обновлялось, как только store изменяется, поэтому вручную следить за обновлениями больше не надо.

index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/app';

import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducer from './reducer';

```

```

const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);

// больше не надо
const update = () => { ... }
store.subscribe(update);
update();

```

Connect()

Компоненты должны использовать диспатчи и информацию из store. Сейчас все компоненты по иерархии через контекст имеют доступ к store, но чтобы использовать его, надо знать, какие поля и диспатчи передавать в пропсы. Чтобы их передавать в компонент, используется функция connect.

Connect – это НОС, т.е. функция, которая делает какие-то подготовительные действия, а потом обворачивает компонент и предоставляет ему нужную инфо и возвращает новый, готовый компонент.

Документация [здесь](#).

В качестве параметров она принимает параметры конфигурации:

`mapStateToProps` – это функция, которая получает текущий state из store и её задача – вернуть объект с теми свойствами, которые нужны react-компоненту.

mapDispatchToProps

Может быть функцией или объектом.

Эта функция получит на вход функцию dispatch автоматом. Возвращает объект, в качестве ключей – имена создаваемых функций-диспачей, передаваемых в пропсы, а значения – action или actionCreator, которые будут обёрнуты в dispatch.

```

import { connect } from 'react-redux';

function connect(mapStateToProps, mapDispatchToProps, mergeProps, options)(Component)
  export default connect(mapStateToProps, mapDispatchToProps)(Counter);

const mapStateToProps = (state) => {
  return {
    name: state.name,
    age: state.age
  };
}

const mapDispatchToProps = (dispatch) => {
  return {
    inc: () => dispatch({type: 'INC'})
  }
}

// с actionCreators
const mapDispatchToProps = (dispatch) => {
  return {

```

```

inc: () => dispatch(inc()),
dec: () => dispatch(dec()),
rnd: () => {
  const randomValue = Math.floor(Math.random() * 10);
  dispatch(rnd(randomValue));
}
};

// Можно сократить ещё сильнее
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as actions from '../actions';

const mapDispatchToProps = (dispatch) => {

  const { inc, dec, rnd } = bindActionCreators(actions, dispatch);

  return {
    inc,
    dec,
    rnd: () => {
      const randomValue = Math.floor(Math.random()*10);
      rnd(randomValue);
    }
  }
};

```

rnd – это функция, которая сначала считает случайное число.

На практике выглядит так:

```

import React from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux'

import * as actions from '../actions';

const Counter = ({ counter, inc, dec, rnd }) => {
  return (
    <div className="jumbotron">
      <h2 id="counter">{counter}</h2>

      <button
        onClick={dec}
        id="dec" className="btn btn-primary btn-lg">DEC</button>

      <button
        onClick={inc}
        id="inc" className="btn btn-primary btn-lg">INC</button>

      <button
        onClick={rnd}
        id="rnd" className="btn btn-primary btn-lg">RND</button>
    </div>
  );
};

const mapStore = (store) => ( {counter: store} );

```

```
const mapDispatch = (dispatch) => {
  const {inc, dec, rnd} = bindActionCreators(actions, dispatch);

  return {
    inc,
    dec,
    rnd: () => {
      const randomValue = Math.floor(Math.random() * 10);
      rnd(randomValue);
    }
  };
};

export default connect(mapStore, mapDispatch)(Counter);
```

</>

React + Redux

Установка

Установка:

```
npx create-react-app re-store
npm install prop-types react-router-dom redux react-redux
```

Универсальные компоненты

Большинству приложений необходимы на старте:

ErrorBoundary	отлавливатель ошибок
Контекст	для передачи инфо к нижестоящим по иерархии компонентам
HOC для контекста	withContextService для удобной работы с consumer

Index.js:

```
ReactDOM.render(
  <Provider store={store}>
    <ErrorBoundary>
      <BookstoreServiceProvider value={bookstoreService}>
        <BrowserRouter>
          <App />
        </BrowserRouter>
      </BookstoreServiceProvider>
    </ErrorBoundary>
  </Provider>
);
```

Папки и файлы на старте:

```
store.js
actions
```

```
index.js

reducers
  index.js

components
  app
  error-boundry          отлавливатель ошибок
  error-indicator        элемент с визуалом ошибки
  spinner                отображается во время загрузки по сети
  bookstore-service-context
    hoc
    with-bookstore-service компонент-обёртка упрощает использование сервиса
  pages

services
  bookstore-service.js   сетевые запросы, трансформация полученных данных

utils
```

Порядок создания:

```
bookstore-service
app
error-indicator
spinner
error-boundry
bookstore-service-context
```

```
hoc => bookstore-service
pages (просто каталог)
```

Error-boundry максимально простой:

```
import React, { Component } from 'react';
import ErrorIndicator from '../error-indicator/';

export default class ErrorBoundary extends Component {

  state = { hasError: false };

  componentDidCatch() {
    this.setState({ hasError: true });
  }

  render() {
    if (this.state.hasError) return <ErrorIndicator/>

    return this.props.children;
  }
}
```

bookstore-service-context

```
import React from 'react';

const {
  Provider: BookstoreServiceProvider,
  Consumer: BookstoreServiceConsumer
```

```
} = React.createContext();

export {
  BookstoreServiceProvider,
  BookstoreServiceConsumer
};
```

WithBookstoreService

Это хос-компонент-обёртка. Нужен для того, чтобы было удобнее использовать сервис: он обернёт в Consumer компонент, чтобы он мог использовать данные из Provider.

Не забыть, что в consumer в качестве дочернего элемента передаётся функция, которая получает в качестве аргумента value из провайдера.

```
import React from 'react';
import { BookstoreServiceConsumer } from '../bookstore-service-context/bookstore-service-
context';

const withBookstoreService = (ошибка?) => (Wrapped) => {

  return (props) => {
    return (
      <BookstoreServiceConsumer>
        {
          (bookstoreService) => {
            return (<Wrapped {...props} bookstoreService={bookstoreService} />);
          }
        }
      </BookstoreServiceConsumer>
    );
  };
};

export default withBookstoreService;
```

Необходимые компоненты готовы.

Redux компоненты

Порядок создания:

reducer
action
store

reducer.js

```
const initialState = {
  books: [],
};

const reducer = (state = initialState, action) => {
  switch (action.type) {

    case 'BOOKS LOADED':
      return { ...state, books: action.payload };

    default:
      return state;
  }
};
```

```
export default reducer;
```

booksLoaded – это функция action-creator, которой для работы нужно передать список книг. Она его получает, обращается в reducer, а reducer смотрит на type и записывает полученные payload куда надо.

```
const booksLoaded = (newBooks) => ({ type: 'BOOKS_LOADED', payload: newBooks });

export { booksLoaded };
```

store.js

```
import { createStore } from 'redux';
import reducer from './reducers';

const store = createStore(reducer);

export default store;
```

Каркас приложения

Сейчас будет сборка компонентов, чтобы приложение что-то отображало на странице и использовало redux. Пока без асинхронности.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

import store from './store';
import { Provider } from 'react-redux';
import { BrowserRouter } from 'react-router-dom';

import BookstoreService from './services/bookstore-service';
import { BookstoreServiceProvider } from './components/bookstore-service-context';

import ErrorBoundary from './components/error-boundry/error-boundry';
import App from './components/app';

const bookstoreService = new BookstoreService();

ReactDOM.render(
  <Provider store={store}>
    <ErrorBoundary>
      <BookstoreServiceProvider value={bookstoreService}>
        <BrowserRouter>
          <App />
        </BrowserRouter>
      </BookstoreServiceProvider>
    </ErrorBoundary>
  </Provider>,
  document.getElementById('root')
);
```

Если всё работает и на экране отображается App, то самое время дописать компонент app: компоненту надо передать контекст для использования сетевого запроса, используется hoc. После обёртывания, экземпляр сетевого запроса будет доступен через пропсы компонента.

Код ниже – это пример, как будет работать редакс. На самом деле, с редаксом будут работать другие компоненты, а не app, но сейчас можно проверить его работу.

```
import React from 'react';
import { withBookstoreService } from '../hoc';
import './app.css';

const App = ({ bookstoreService }) => {
  console.log(bookstoreService.getBooks())
  return <div>App</div>;
};

export default withBookstoreService()(App);
```

В service можно докинуть dummy-данные:

```
const books = [
  {id: 1, title: 'Alice in Wonderland', author: 'Lewis Carroll'},
  {id: 2, title: 'Call of Cthulhu', author: 'Howard Lovecraft'},
];

export default class BookstoreService {

  getBooks() {
    return books;
  }
};
```

Роутинг

Привести App в исходное состояние (без redux), потому что с редаксом будут работать другие компоненты.

Создаётся папка pages, в неё добавляются 2 файла: home-page.js и cart-page.js.

Пока они ничего не делают, просто отображают своё название.

```
app.js
import React from 'react';
import { Routes, Route } from 'react-router-dom';
import './app.css';
import { HomePage, CartPage } from '../pages';

const App = () => {
  return (
    <div>
      <Routes>
        <Route path="/" element={HomePage} />
        <Route path="/cart" element={CartPage} />
      </Routes>
    </div>
  );
};

export default App;
```

```
home-page.js
import React, { Component } from 'react';
```

```
export default class HomePage extends Component {
  render() {
    return <div>Home</div>
  }
};
```

```
cart-page.js
import React from 'react';
const CartPage = () => {
  return <div>Cart</div>;
};
export default CartPage;
```

Внутри homepage будет находиться список книг.
Эти 2 компонента надо дописать и передать в него.

```
book-list-item.js
import React, { Fragment } from 'react';
import './book-list-item.css';
const BookListItem = ({ book }) => {
  const { title, author } = book;

  return (
    <Fragment>
      <span>{title}</span>
      <span>{author}</span>
    </Fragment>
  );
}
export default BookListItem;
```

BookList – более сложный компонент. Он будет делать сетевой запрос в componentDidMount, поэтому это должен быть класс-компонент.

```
book-list.js
```

Redux компоненты

Сначала reducer, потом action и store.

Reducer будет записывать в store полученный список книг.

```
reducers => index.js
```

```

const initialState = { books: [] };

const reducer = (state = initialState, action) => {
  switch (action.type) {

    case 'BOOKS_LOADED':
      return { books: action.payload };

    default: return state;
  }
};

export default reducer;

actions => index.js
const booksLoaded = (newBooks) => ({ type: 'BOOKS_LOADED', payload: newBooks });
export { booksLoaded };

store.js
import { createStore } from 'redux';
import reducer from './reducers';

const store = createStore(reducer);

export default store;

```

Provider store

На самом верхнем уровне – провайдер из react-redux, потому что это центр приложения.

ErrorBoundary

Как можно выше, чтобы отлавливал любые ошибки.

BookstoreServiceProvider

Это контекст, который будет передавать экземпляр сетевого сервиса ниже по иерархии.

Router

Чтобы все компоненты имели доступ к функциональности роутинга.

index.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { BrowserRouter as Router } from 'react-router-dom';

import App from './components/app';
import ErrorBoundary from './components/error-boundary';
import BookstoreService from './services/bookstore-service';
import { BookstoreServiceProvider } from './components/bookstore-service-context'

import store from './store';

const bookstoreService = new BookstoreService();

ReactDOM.render(
  <Provider store={store}>
    <ErrorBoundary>
      <BookstoreServiceProvider value={bookstoreService}>
        <Router>

```

```
<App />
</Router>
</BookstoreServiceProvider>
</ErrorBoundary>
</Provider>,
document.getElementById('root')
);
```

Работа с асинхронными данными

Service – это эмуляция сетевого запроса. Чтобы сделать запрос асинхронным, надо прописать service так:

```
export default class BookstoreService {

  data = [ массив объектов с книгами ];

  getBooks() {
    return new Promise((resolve) => {
      return setTimeout(() => resolve(this.data), 500);
    });
  }
}
```

Дописать

Обработка ошибок

Статус ошибки хранится в store.

Ошибка – это null по умолчанию, а также если начался запрос к серверу или книги успешно получены.

В reducer добавляется новое действие. Через payload будут переданы детали ошибки.

Reducer

```
const initialState = {
  books: [],
  loading: true,
  error: null
};

const reducer = (state = initialState, action) => {
  switch (action.type) {

    case 'BOOKS_REQUESTED':
      return { books: [], loading: true, error: null };

    case 'BOOKS_LOADED':
      return { books: action.payload, loading: false, error: null };

    case 'BOOKS_ERROR':
      return { books: [], loading: false, error: action.payload };

    default:
      return state;
  }
};

export default reducer;
```

actions

```
const booksError = (error) => ({ type: 'BOOKS_ERROR', payload: error });
```

book-list

Протаскивание в пропсы – как всегда через mapDispatch и mapState.

Протаскивание через mapState надо, чтобы отобразить текст ошибки. Перед этим она запишется в state, где раньше был null.

Не забыть обновить render, если есть ошибка. Обновление тривиальное.

```
const mapStateToProps = (state) => {
  return {books: state.books, loading: state.loading, error: state.error};
};

const mapDispatchToProps = {booksLoaded, booksRequested, booksError};

componentDidMount() {
  const { bookstoreService, booksLoaded, booksRequested, booksError } = this.props;
  booksRequested();
  bookstoreService.getBooks()
    .then((data) => booksLoaded(data))
    .catch((err) => booksError(err));
}

render() {
  const { books, loading, error } = this.props;

  if (loading) return <Spinner />

  if (error) {
    return <ErrorIndicator/>
  }
}
```

аргумент ownProps

Сейчас компонент book-list в методе componentDidMount получает много свойств и выполняет много действий, которые его прямой обязанностью не являются. По факту ему нужно только то, что содержится в функции render.

Всю логику сетевого запроса и обработки ошибки можно вынести в отдельную функцию.

```
books-list.js

componentDidMount() {

// const { bookstoreService, booksLoaded, booksRequested, booksError } = this.props;
// booksRequested();
// bookstoreService.getBooks()
//   .then((data) => booksLoaded(data))
//   .catch((err) => booksError(err));

  this.props.fetchBooks();
}
```

mapDispatch может быть либо объектом, либо функцией.

В случае функции, он принимает первым параметром dispatch и возвращает объект {названиеСвойства: ac}

Прикол в том, что в качестве значения в этот объект не обязательно оборачивать action-creator, а можно передать что угодно.

Books-list

Было:

```

const mapDispatch = {booksLoaded, booksRequested, booksError};

стало:
const mapDispatch = (dispatch) => {
  return {
    fetchBooks: () => {
      dispatch(booksRequested());

      bookstoreService.getBooks()
        .then((data) => dispatch(booksLoaded(data)) )
        .catch((err) => dispatch(booksError(err)) );
    }
  }
}

```

AC надо обернуть в dispatch, чтобы они заработали.
Сейчас есть одна проблема: bookstoreService недоступен.

У функций mapState и mapDispatch есть второй параметр: ownProperty. Если компонент во что-то обёрнут, то через второй параметр будут доступны собственные свойства создаваемого компонента.

Получается, что в ownProps будет bookstoreService.

```

const mapDispatch = (dispatch, ownProps) => {
  const { bookstoreService } = ownProps;

  return {
    fetchBooks: () => {
      dispatch(booksRequested());

      bookstoreService.getBooks()
        .then((data) => dispatch(booksLoaded(data)) )
        .catch((err) => dispatch(booksError(err)) );
    }
  }
}

export default compose(
  withBookstoreService(),
  connect(mapState, mapDispatch),
)(BookList);

```

Naming Convention

Несмотря на то, что функция fetchBooks не связана с работой store, она используется только там, где используются action-creators. Поэтому её можно положить с ними в один файл.

Аргументы передаются в первую функцию, чтобы компонент не заботился об этих пргументах, а просто вызывал функцию (с пустыми скобками).

Больше не имеет смысла экспортовать ac, потому что единственное место, где они используются – это функция fetchBooks.

Actions

```

const booksLoaded = (newBooks) => ({ type: 'BOOKS_LOADED', payload: newBooks });
const booksRequested = () => ({ type: 'BOOKS_REQUESTED' });
const booksError = (error) => ({ type: 'BOOKS_ERROR', payload: error });

const fetchBooks = (bookstoreService, dispatch) => () => {
  dispatch(booksRequested());
}

```

```

bookstoreService.getBooks()
  .then((data) => dispatch(booksLoaded(data)) )
  .catch((err) => dispatch(booksError(err)) );
}

export {
  fetchBooks
};

```

Book-list

```

const mapDispatch = (dispatch, ownProps) => {

  const { bookstoreService } = ownProps;
  return { fetchBooks: fetchBooks(bookstoreService, dispatch) };
};

```

Можно сразу деструктурировать:

```

const mapDispatch = (dispatch, { bookstoreService } ) => {
  return { fetchBooks: fetchBooks(bookstoreService, dispatch) };
};

```

Именование

Действия, которые отправляют запрос,

BOOKS_REQUESTED =>	FETCH_BOOKS_REQUEST	данные именно получаются, а не обновляются.
	UPDATE_BOOKS_REQUEST	обновление данных
BOOKS_LOADED =>	FETCH_BOOKS_SUCCESS	успешное получение данных
BOOKS_ERROR =>	FETCH_BOOKS_FAILURE	ошибка

Компоненты-контейнеры

В React считается хорошей практикой разделять компоненты, которые отвечают за поведение и компоненты, которые отвечают за отображение.

Сейчас book-list делает всё сразу: получает данные, обрабатывает состояния loading и error и формирует внешний вид компонента.

Book-list разбивается на 2 компонента:

- book-list-container отвечает за логику (спиннер, ошибка, запрос данных)
- book-list отвечает за отрисовку

Документация redux предписывает называть компоненты-обёртки container и размещать их в папке containers рядом с components. Но в этом проекте компоненты будут в одном файле.

```

const BookList = ({ books }) => {
  return (
    <ul className="book-list">
      {
        books.map((book) => {
          return (
            <li key={book.id}><BookListItem book={book} /></li>
          );
        })
      }
    </ul>
  );
}

```

```
        }
      </ul>
    );
}

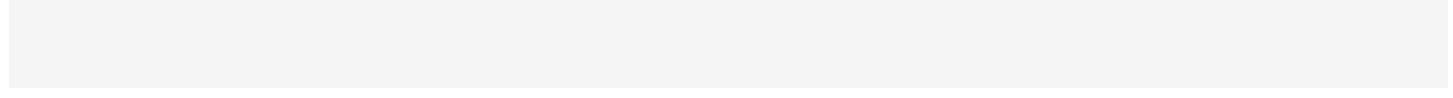
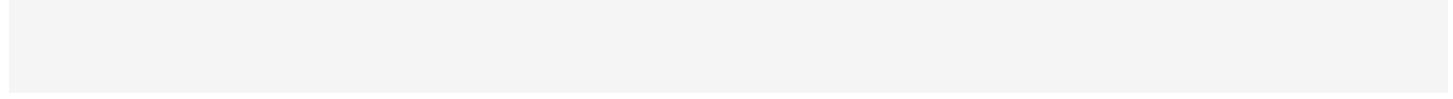
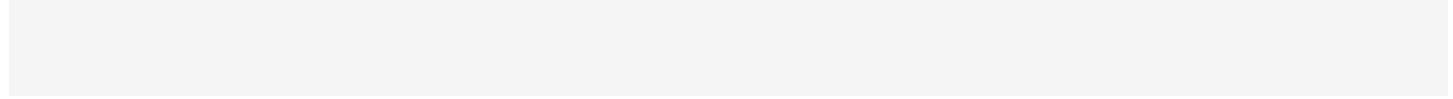
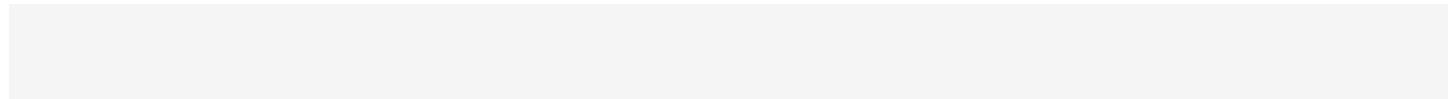
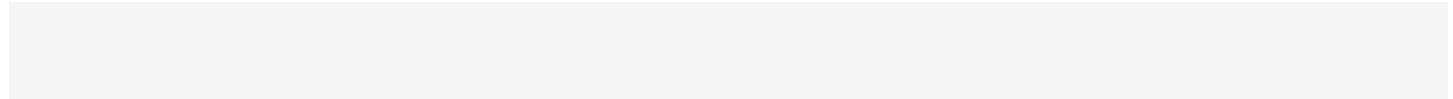
class BookListContainer extends Component {

  componentDidMount() {
    this.props.fetchBooks();
  }

  render() {
    const { books, loading, error } = this.props;

    if (loading) return <Spinner />
    if (error) return <ErrorIndicator/>

    return <BookList books={books}/>
  }
}
```



</>

Модули

Модуль обычно содержит класс или библиотеку с функциями. Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году. На данный момент она поддерживается большинством браузеров и Node.js.

Модуль – это просто файл. Один скрипт – это один модуль. Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью:

export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
import позволяет импортировать функциональность из других модулей.

Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута `<script type="module">`.

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.

```
<!doctype html>
<script type="module">
  import {sayHi} from './say.js';

  document.body.innerHTML = sayHi('John');
</script>
```

Udemy модули

Локальные

Для того, чтобы импортировать что-то из файла, используется указание относительного пути.

```
// именованный экспорт и импорт указанных функций или констант (просто перечислить)
export { add, subtract, multiply };
import { add, subtract } from './file1';

// переименование экспортируемых и импортируемых сущностей
export { add as newAddName, subtract as newSubName };
import { add as newAddName, subtract as newSubName } from './file1';

// импорт всего и сразу
import * as calc from './file1';
calc.add()

// экспорт по умолчанию без имён. Скобки не ставятся. Используется как переменная.
export default add;
import add from './file1';           // импорт по умолчанию можно переименовать
import newAddName from './file1';
add()

// можно комбинировать синтаксис импорта по умолчанию и именованные импорты
export { subtract, multiply, divide };
export default add;

import add, {subtract, multiply} from './file1';
import add, * as calc from './file1';
```

```
// можно прописывать экспорт прямо перед объявлением переменной
export default class Graph {...}
```

Если требуется просто запустить код, который находится в другом файле (например, для сайд-эффектов), можно сделать так:

```
file1.js
console.log('Side effect');

запуск из другого файла
import './file1';
```

Некоторые сборщики могут прописывать такой паттерн, чтобы информировать о том, что они от чего-то зависят. Например, от CSS-файла. После этого сборщик будет знать: чтобы запустить это файл, нужно в проект включить зависимость. Но это поведение специфично для разных сборщиков.

Глобальные

Если есть внешняя библиотека, установленная, например, через NPM, синтаксис импорта будет отличаться.

Для того, чтобы импортировать что-то из файла, использовалось указание относительного пути.

Для библиотеки, которая установлена, как зависимость, указывается только её наименование.

```
import 'joke' from 'one-line=joke'

// использование
joke.getRandomJoke().body;
```

</>

Особенности модулей

Отличие модулей от обычных скриптов:

Всегда «use strict»

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

```
<script type="module">
  a = 5;
  // ошибка
</script>
```

Своя область видимости переменных

Каждый модуль имеет свою собственную область видимости. Переменные и функции, объявленные в модуле, не видны в других скриптах. Поэтому нельзя импортировать модуль и просто так обращаться к его переменным.

Модули должны экспортировать функциональность, предназначенную для использования извне. А другие модули могут её импортировать.

Правильный вариант импорта из модуля:

```
user.js
export let user = "John";

hello.js
import {user} from './user.js';
document.body.innerHTML = user; // John

index.html
<!doctype html>
<script type="module" src="hello.js"></script>
```

В браузере также существует независимая область видимости для каждого отдельного скрипта `<script type="module">`:

```
<script type="module">
  // Переменная доступна только в этом модуле
  let user = "John";
</script>

<script type="module">
  alert(user);
  // Error: user is not defined
</script>
```

Если нам нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту `window`, тогда получить значение переменной можно обратившись к `window.user`. Но это должно быть исключением, требующим веской причины.

Код в модуле выполняется только один раз при импорте

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам.

Это очень важно для понимания работы модулей.

Во-первых, если при запуске модуля возникают побочные эффекты, например выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

```
// alert.js
alert("Модуль выполнен!");
// Импорт одного и того же модуля в разных файлах

// 1.js
import `./alert.js`;
// Модуль выполнен!

// 2.js
import `./alert.js`;
// (ничего не покажет)
```

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если мы хотим, чтобы что-то можно было использовать много раз, то экспортим это.

Более продвинутый пример – модуль экспортирует объект `admin`.

Если модуль импортируется в нескольких файлах, то объект `admin` станет общим для них всех. Если этот объект как-то изменится в одном модуле, то все остальные модули тоже увидят эти изменения.

В примере ниже оба файла 1 и 2 импортируют один и тот же объект. Изменения, сделанные в файле 1, будут видны в файле 2:

```
// admin.js
export let admin = {
  name: "John"
};

// 1.js
import {admin} from './admin.js';
admin.name = "Pete";

// 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete
```

Такое поведение позволяет конфигурировать модули при первом импорте. Мы можем установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль `admin.js` предоставляет определённую функциональность, но ожидает передачи учётных данных в объект `admin` извне.

В `init.js`, первом скрипте нашего приложения, мы установим `admin.name`. Тогда все это увидят, включая вызовы, сделанные из самого `admin.js`

[Другой](#) модуль тоже увидит `admin.name`:

```
// admin.js
export let admin = { };

export function sayHi() {
  alert(`Ready to serve, ${admin.name}!`);
}

// init.js
import {admin} from './admin.js';
admin.name = "Pete";

// other.js
import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete
sayHi();           // Ready to serve, Pete!
```

import.meta

Объект `import.meta` содержит информацию о текущем модуле. Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

```
<script type="module">
  alert(import.meta.url); // ссылка на html страницу для встроенного скрипта
</script>
```

В модуле «this» не определён

В модуле на верхнем уровне `this` не определён (`undefined`).

Сравним с не-модульными скриптами, там `this` – глобальный объект:

```
<script>
  alert(this); // window
</script>

<script type="module">
```

```
    alert(this); // undefined
</script>
```

Особенности в браузерах

Есть и несколько других, именно браузерных особенностей скриптов с type="module" по сравнению с обычными скриптами.

#? Не понимаю эту хуйню, тут тема переплетается с DOM.

Модули являются отложенными (deferred)

Модули всегда выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом defer (описан в главе [Скрипты: async, defer](#)). Это верно и для внешних и встроенных скриптов-модулей.

Другими словами:

- загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
- модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
- сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними.

```
<script type="module">
  alert(typeof button);
  // object
  // скрипт может 'видеть' кнопку под ним так как модули являются отложенными, то скрипт начнёт
  // выполняться только после полной загрузки страницы
</script>
```

Сравните с обычным скриптом:

```
<script>
  alert(typeof button);
  // Ошибка, кнопка не определена
  // скрипт не видит элементы под ним обычные скрипты запускаются сразу, не дожидаясь полной
  // загрузки страницы
</script>

<button id="button">Кнопка</button>
```

Модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта мы видим первым: второй скрипт выполнится раньше, чем первый. Поэтому мы увидим сначала undefined, а потом object.

При использовании модулей нам стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполняются модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Нам следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

Атрибут async работает во встроенных скриптах

Для не-модульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут async работает на любых скриптах.

Например, в скрипте ниже есть async, поэтому он выполнится сразу после загрузки, не ожидая других скриптов. Скрипт выполнит импорт (загрузит ./analytics.js) и сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются.

Это очень полезно, когда модуль ни с чем не связан, например для счётчиков, рекламы, обработчиков событий.

```
<!-- загружаются зависимости (analytics.js) и скрипт запускается -->
<!-- модуль не ожидает загрузки документа или других тэгов <script> -->
<script async type="module">
  import {counter} from './analytics.js';

  counter.count();
</script>
```

Внешние скрипты

Внешние скрипты с атрибутом `type="module"` имеют два отличия:

1. Внешние скрипты с одинаковым атрибутом `src` запускаются только один раз:

```
<!-- скрипт my.js загрузится и будет выполнен только один раз -->
<script type="module" src="my.js"></script>
<script type="module" src="my.js"></script>
```

2. Внешний скрипт, который загружается с другого домена, требует указания заголовков [CORS](#). Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок `Access-Control-Allow-Origin` означающий, что загрузка скрипта разрешена.

Это обеспечивает лучшую безопасность по умолчанию.

```
<!-- another-site.com должен указать заголовок Access-Control-Allow-Origin -->
<!-- иначе, скрипт не выполнится -->
<script type="module" src="http://another-site.com/their.js"></script>
```

Не допускаются «голые» модули

В браузере `import` должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (`bare`). Они не разрешены в `import`.

Например, этот `import` неправильный:

```
import {sayHi} from 'sayHi';

// Ошибка, "голый" модуль
// путь должен быть, например './sayHi.js' или абсолютный
```

Другие окружения, например Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Совместимость, «nomodule»

Старые браузеры не понимают атрибут `type="module"`. Скрипты с неизвестным атрибутом `type` просто игнорируются. Мы можем сделать для них «резервный» скрипт при помощи атрибута `nomodule`:

```
<script type="module">
  alert("Работает в современных браузерах");
</script>

<script nomodule>
  alert("Современные браузеры понимают оба атрибута - и type=module, и nomodule, поэтому пропускают этот тег script")
  alert("Старые браузеры игнорируют скрипты с неизвестным атрибутом type=module, но выполняют этот.");
</script>
```

Инструменты сборки

В реальной жизни модули в браузерах редко используются в «сыром» виде. Обычно, мы объединяем модули вместе, используя специальный инструмент, например [Webpack](#) и после выкладываем код на рабочий сервер. Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «своё», например CSS/HTML-модули.

Сборщик делает следующее:

1. Берёт «основной» модуль, который мы собираемся поместить в `<script type="module">` в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее)
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный `import` функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.

В процессе могут происходить и другие трансформации и оптимизации кода:

- Недоступимый код удаляется.
- Неиспользуемые экспорты удаляются («tree-shaking»).
- Специфические операторы для разработки, такие как `console` и `debugger`, удаляются.
- Современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью, например, с помощью [Babel](#).
- Полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если мы используем инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют `import/export` на свои вызовы. Поэтому итоговую сборку можно подключать и без атрибута `type="module"`, как обычный скрипт:

```
<!-- Предположим, что мы собрали bundle.js, используя например утилиту Webpack -->
<script src="bundle.js"></script>
```

Итого

Подводя итог, основные понятия:

1. Модуль – это файл. Чтобы работал `import/export`, нужно для браузеров указывать атрибут `<script type="module">`. У модулей есть ряд особенностей:
 - Отложенное (deferred) выполнение по умолчанию.
 - Атрибут `async` работает во встроенных скриптах.
 - Для загрузки внешних модулей с другого источника, он должен ставить заголовки CORS.
 - Дублирующиеся внешние скрипты игнорируются.
2. У модулей есть своя область видимости, обмениваться функциональностью можно через `import/export`.
3. В модулях всегда включена директива `use strict`.
4. Код в модулях выполняется только один раз. Экспортируемая функциональность создаётся один раз и передаётся всем импортёрам.

Когда мы используем модули, каждый модуль реализует свою функциональность и экспортирует её. Затем мы используем `import`, чтобы напрямую импортировать её туда, куда необходимо. Браузер загружает и анализирует скрипты автоматически.

В реальной жизни часто используется сборщик [Webpack](#), чтобы объединить модули: для производительности и других «плюшек».

Экспорт и импорт

Документация по импорту [здесь](#).

Документация по экспорту [здесь](#).

Директивы экспорт и импорт имеют несколько вариантов вызова.

Экспорт при объявлении

Можно пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс. Например, все следующие экспорты допустимы:

```
// экспорт массива
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// экспорт константы
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// экспорт класса
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

Экспорт отдельно от объявления

Также можно написать export отдельно: сначала объявить, а затем экспортировать.

Можно располагать и export выше функций.

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // список экспортируемых переменных
```

Импорт отдельных элементов { }

список того, что нужно импортировать, перечисляется в фигурных скобках import {...}

вот так:

```
import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

Импорт всего сразу *

Можно импортировать всё сразу **в виде объекта**, используя import * as <obj>

вот так:

```
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

В учебнике после этого примера пишется о том, что так делать не круто и приводятся доказательства. См. учебник, если понадобится.

Импорт «как» as

Можно использовать «as», чтобы импортировать под другими именами.

Например, для краткости импортируем sayHi в локальную переменную hi, а sayBye импортируем как bye:

```
import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

Экспортировать «как»

Аналогичный синтаксис «as» существует и для export.

```
// say.js
...
export {sayHi as hi, sayBye as bye};

// main.js
import * as say from './say.js';

say.hi('John'); // Hello, John!
say.bye('John');
```

Теперь hi и bye – официальные имена для внешнего кода, их нужно использовать при импорте.

Экспорт по умолчанию

На практике модули встречаются в основном одного из двух типов:

1. Модуль, содержащий библиотеку или набор функций, как say.js выше.
2. Модуль, который объявляет что-то одно, например модуль user.js экспортирует только class User.

Удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле. Потребуется много файлов, но это не проблема: навигация по проекту становится проще, особенно, если у файлов хорошие имена, и они структурированы по папкам.

Модули предоставляют специальный синтаксис export default («экспорт по умолчанию») для второго подхода.

Ставим export default перед тем, что нужно экспортировать и потом импортируем без фигурных скобок. В файле может быть только один export default:

```
// user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

// main.js
import User from './user.js'; // не {User}, просто User
new User('John');
```

Запомним: фигурные скобки необходимы в случае именованных экспортов, для export default они не нужны.

Именованный экспорт	Экспорт по умолчанию
export class User {...}	export default class User {...}
import {User} from ...	import User from ...

Технически в одном модуле может быть как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают.

Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя. Это нормально, потому что может быть только один export default на файл, так что import без фигурных скобок всегда знает, что импортировать. Без default такой экспорт выдал бы ошибку

Примеры снизу – это корректные экспорты по умолчанию:

```
export default class { // у класса нет имени
  constructor() { ... }
}

export default function(user) { // у функции нет имени
  alert(`Hello, ${user}!`);
}

// экспортируем, не создавая переменную
export default ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Имя «default»

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется default. Например, чтобы экспортить функцию отдельно от её объявления:

```
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// то же самое, как если бы мы добавили "export default" перед функцией
export {sayHi as default};
```

Или, ещё ситуация, давайте представим следующее: модуль user.js экспортирует одну сущность «по умолчанию» и несколько именованных (редкий, но возможный случай):

```
// user.js
export default class User {
  constructor(name) {
    this.name = name;
  }
}

export function sayHi(user) {
  alert(`Hello, ${user}!`);
}
```

Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

```
// main.js
```

```
import {default as User, sayHi} from './user.js';
new User('John');
```

И если мы импортируем всё как объект import *, тогда его свойство default – как раз и будет экспортом по умолчанию:

```
// main.js
import * as user from './user.js';

let User = user.default; // экспорт по умолчанию
new User('John');
```

Довод против экспортов по умолчанию

Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, говорит нам, что именно экспортируется. Именованные экспорты вынуждают нас использовать правильное имя при импорте, в то время как для экспорта по умолчанию мы выбираем любое имя при импорте. Так что члены команды могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо.

Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных должны соответствовать именам файлов:

```
import User from './user.js';
import LoginForm from './loginForm.js';
import func from '/path/to/func.js';
```

Реэкспорт

Синтаксис «реэкспорта» export ... from ... позволяет импортировать что-то и тут же экспортировать, возможно под другим именем, вот так:

```
export {sayHi} from './say.js';
// реэкспортировать sayHi

export {default as User} from './user.js';
// реэкспортировать default
```

</>

GIT

Добавление репозитория с Github

```
git init
git remote add origin https://github.com/Join-Red-Army/mogo-landing.git
git add .
git commit -am 'added images for the page'
git clone 'https'
```

Откатиться к последнему коммиту

команда безвозвратно удаляет несохраненные текущие изменения из рабочей области и из индекса

```
git reset --hard HEAD
```

Удалить коммит из github

```
git reset --hard a3775a5485af      хэш-код коммита, к которому хотим вернуться  
git push --force                  отсылаем эту правку на гитхаб
```

Получить изменения с сервера

```
git pull
```

Узнать текущую ветку

```
git branch
```

Переключиться на конкретный коммит

```
git checkout [номер]
```

```
</>
```

SASS

Документация: [sass-lang](#)

Это метаязык на основе CSS, предназначенный для увеличения уровня абстракции CSS-кода и упрощения файлов каскадных таблиц стилей.

```
sass --watch app/sass:public/stylesheets
```

Копирование названий селекторов

`&:hover`

Не нашёл это в документации. & указывает на селектор выше по вложенности, поэтому вместо `btn:hover` можно прописать `&:hover`.

Переменные

В переменных хранятся любые css-значения. Создаются и вставляются через символ \$.

Вложенность

SASS позволяет вкладывать селекторы, как это делается в html.

Фрагментирование

Можно создавать файлы с фрагментами CSS, которые подключаются к другим sass.

Такие файлы создаются с начальным подчёркиванием: _partial.scss.

Это сообщить компилятору, что файл не надо генерировать в css.

Фрагменты используются с правилом @use.

Модули

Не обязательно хранить весь sass в одном файле. Можно ссылаться на [mixins](#) и [functions](#).

Модули будут компилироваться в итоговый css-файл.

Расширение файла не указывается.

Модуль подключается через @use 'module_name'

Mixins

Миксин нужен для создания сразу нескольких строк свойств CSS, которые можно все вместе куда-то вставлять.

В миксины также можно передавать значения извне, назначив переменную, которая будет использоваться внутри.

@mixin name

Создание директивы и присвоение ей имени.

@include name

Использование миксина

Extend/Inheritance

@extend позволяет передавать набор свойств из одного селектора в другой.

A placeholder class – специальный тип класса, который печатает только при расширении.

Ничего не понял, см. документацию.

Документация: [sass-lang](#)

Установка

npm install -g sass

Использование

sass input.scss output.css

компилировать sass файл в css

--watch

sass --watch input.scss output.css

следить за указанными файлами или папками.

sass --watch app/sass:public/stylesheets

следить за всеми файлами в папке app/sass и компилировать их в папку public/stylesheets

Переменные

\$ символ для создания и вставки переменной

Создание

\$font-stack: Helvetica, sans-serif;

\$primary-color: #333;

Использование

```
body {  
  font: 100% $font-stack;  
  color: $primary-color;  
}
```

Копирование названий селекторов

&:hover

```
.btn {  
    background: ...  
    color: ...  
  
    &:hover {  
        background-color: ...  
    }  
}
```

Вложенность

```
nav {  
    ul {  
        margin: 0;  
        list-style: none;  
    }  
    li { display: inline-block; }  
    a {  
        padding: 6px 12px;  
    }  
}
```

Модули

```
_base.scss  
body {  
    font: 100% $font-stack;  
    color: $primary-color;  
}
```

```
styles.scss  
@use 'base';  
Body будет вставлен, как если бы он был написан тут.  
Далее пишешь обычный код
```

Mixins

@ mixin name

Создание директивы и присвоение ей имени.

@ include name

Использование миксина

(\$theme)

В скобках указывается имя переменной и её значение по умолчанию.

При использовании миксина, этой переменной можно передать другое значение.

Объявление

```
@ mixin theme($theme: DarkGray) {  
    background: $theme;  
    box-shadow: 0 0 1px rgba($theme, .25);  
}
```

Использование

```
.info {  
    @include theme;  
.success {  
    @include theme($theme: red);  
}
```

</>

Gulp

Gulp — это таск-менеджер для автоматического выполнения часто используемых задач (например, минификации, тестирования, объединения файлов).

API в документации [здесь](#).

Установка

Установка

```
npm install --global gulp-cli  
  
npx mkdirp my-project  
cd my-project  
  
npm init  
npm install --save-dev gulp
```

Create a gulpfile

```
function defaultTask(cb) {  
  cb();  
}
```

подключение плагинов

```
const autoprefixer = require("gulp-autoprefixer");
```

консоль

```
gulp  
gulp <task> <othertask>
```

gulpfile.js

Файл gulp.js автоматом запускается при команде gulp.

Внутри этого файла могут быть API типа src(), dest(), series(), parallel().

Кроме API, можно импортировать обычные JS или Node модули.

Любые экспортированные функции будут зарегистрированы в системе задач gulp.

Транспиляция

Новые версии ноды поддерживают большинство функций TypeScript или Babel, кроме синтакса import-export. gulpfile transpilation документация [здесь](#).

Разделение файлов

Если тасков будет много, каждая таска может быть описана в отдельном файле, затем импортирована в основной. Это также позволит тестить каждую таску независимо.

Можно поместить файл gulpfile.js в папку, которая тоже называется gulpfile.js. Сам файл при этом должен быть назван index.js. Здесь же могут быть модули для тасков.

Если используется трансплаер, надо назвать папку и файл соответственно.

Создание тасков

Каждая таска — это асинхронная функция, которая принимает первый коллбэк-ошибку или возвращает промис, stream, event emitter, child process, or observable.

Синхронные таски не поддерживаются.

Экспорт

Таски могут быть публичными или приватными.

Публичные – экспортируются из gulpfile и могут быть запущены через команду gulp.

Приватные – используются только внутри gulpfile, обычно в апи series() и parallel(). В остальном используются так же, как и публичные, просто не могут быть запущенными из консоли пользователем.

Чтобы сделать таску публичной, её надо экспортировать из gulpfile:

```
const { series } = require('gulp');

clean – приватная функция, используется в series()
function clean(cb) {
  cb();
}

build – публичная функция, используется в series()
function build(cb) {
  cb();
}

exports.build = build;
exports.default = series(clean, build);
```

будут запущены и выполнены последовательно: default (clean, build).

Составление тасков

Gulp позволяет объединять индивидуальные таски в одну большую операцию с помощью методов series() и parallel(). Эти методы принимают любое кол-во тасков, а также могут быть вложены друг в друга.

series()

Позволяет выполнять таски строго последовательно. Сначала импортируешь, затем используешь:

```
// сначала импротрируешь
const { series } = require('gulp');

function transpile(cb) {
  cb();
}
function bundle(cb) {
  cb();
}

// затем экспортируешь и используешь
exports.build = series(transpile, bundle);
```

parallel()

Ну, соответственно

```
const { parallel } = require('gulp');

function javascript(cb) {
  cb();
}
function css(cb) {
  cb();
}

exports.build = parallel(javascript, css);
```

Можно комбинировать задачи и запускать под разными условиями:

```
const { series, parallel } = require('gulp');

if (process.env.NODE_ENV === 'production') {
  exports.build = series(transpile, minify);
} else {
  exports.build = series(transpile, livereload);
}

exports.build = series(
  clean,
  parallel(
    cssTranspile,
    series(jsTranspile, jsBundle)
  ),
  parallel(cssMinify, jsMinify),
  publish
);
```

Async Completion

Пропустил. Тут говорится про возврат асинхронных функций из гальпа.

Работа с файлами

Методы `src()` и `dest()` предоставляются для работы с файлами.

Обычно разные плагины размещаются между ними и с помощью метода `.pipe()` передаются в потоке.

Тем не менее, их можно размещать в разных местах потока, не только в начале и конце.

`src()` может работать в нескольких режимах:

- Buffering по умолчанию. Загружает файлы в память.
- Streaming работает с большими файлами типа огромных изображений или видеофайлов. Контент обрабатывается потоком в виде маленьких чанков и хранится не в памяти, а на жёстком диске.
- Empty не содержит контента и полезен когда обрабатывается мета информация.

`src()`

получает glob (адрес в формате строки) и считывает файлы для передачи в Node stream.
Этот поток должен быть возвращён из задачи.

Импорт: `const { src } = require('gulp');`

`dest()`

Получает адрес директории для вывода файлов (в формате строки).

Также создаёт Node stream.

Когда он получает файл, прошедший через pipeline, он записывает результат в указанную директорию.

Испорт: `const { dest } = require('gulp');`

`symlink()`

Работает как `dest`, но создает ссылки вместо файлов.

`.pipe()`

Для передачи потока из одного плагина в другой.

Размещается в пайплайне.

```
const { src, dest } = require('gulp');
```

```

exports.default = function() {
  return src('src/*.js')
    .pipe(dest('output/'));
}

const { src, dest } = require('gulp');
const babel = require('gulp-babel');

exports.default = function() {
  return src('src/*.js')
    .pipe(babel())
    .pipe(dest('output/'));
}

```

dest также может содержаться внутри .pipe

```

exports.default = function() {
  return src('src/*.js')
    .pipe(babel())
    .pipe(src('vendor/*.js'))
    .pipe(dest('output/'))
    .pipe(uglify())
    .pipe(rename({ extname: '.min.js' }))
    .pipe(dest('output/'));
}

```

Explaining Globs

Глоб – это строка (или literal или wildcard characters), которые используются для указания путей к файлам.

Метод src() ожидает glob или массив globs чтобы определить, какой файл пускать в пайплайн. Как минимум, один глоб должен вести к файлам для работы, чтобы src() не выдал ошибку.

[Micromatch Documentation](#)

[node-glob's Glob Primer](#)

[Begin's Globbing Documentation](#)

[Wikipedia's Glob Page](#)

Сепаратор: /

Экран: //

Сегмент: всё, что между сепараторами.

Нельзя использовать path.join

*

Означает любое кол-во (в т.ч. ни одного) символов в одном сегменте. Обычно используется для добавления всех файлов из одного каталога.

'*.js' – выбрать все файлы в каталоге с расширением .js

**

Означает все файлы, в т.ч. во вложенных каталогах.

'scripts/**/*.js' – выбрать все файлы с расширением .js в текущем и во вложенных подкаталогах.

!

Символ отрицания. Обязательно должен располагаться после «положительного» символа. Логика: положительный сначала ищет массив результатов, а ! удаляет лишние из него.

Чтобы исключить все файлы в каталоге, надо добавить /** после имени каталога.

```
[ 'scripts/**/*.js', '!scripts/vendor/**' ]
```

Если любые неотрицательные глобусы следуют за отрицательным, ничего не будет удалено из более позднего набора совпадений.

Отрицание можно использовать для ограничения поиска по двойным звёздам **
['**/*.js', '!node_modules/**']

Все js файлы в подпапках, кроме любых файлов в папке node_modules.

#

Перекрывающиеся файлы (коллизии). Когда несколько глобов укзывают на одни и те же файлы

Когда в одном src () используются перекрывающиеся глобусы, gulp делает все возможное, чтобы удалить дубликаты, но не пытается дедуплицировать отдельные вызовы src () .

Using Plugins

Гальповские плагины – это [Node Transform Streams](#), которые трансформируют файлы в пайплайн. Они могут менять наименование файла, мета данные или контент каждого файла, который проходит через них.

Плагины из NPM можно найти с приставкой "gulpplugin" и "gulpfriendly".

Их можно объединять, чтобы получить желаемый результат.

```
const { src, dest } = require('gulp');
const uglify = require('gulp-uglify');
const rename = require('gulp-rename');

exports.default = function() {
  return src('src/*.js')
    // The gulp-uglify plugin won't update the filename
    .pipe(uglify())
    // So use gulp-rename to change the extension
    .pipe(rename({ extname: '.min.js' }))
    .pipe(dest('output/'));
}
```

Conditional plugins

Плагины не учитывают тип передаваемых в них файлов. Поэтому надо использовать gulp-if чтобы преобразовывать подмножество файлов.

```
const { src, dest } = require('gulp');
const gulpif = require('gulp-if');
const uglify = require('gulp-uglify');

function isJavaScript(file) {
  // Check if file extension is '.js'
  return file.extname === '.js';
}

exports.default = function() {
  // Include JavaScript and CSS files in a single pipeline
  return src(['src/*.js', 'src/*.css'])
    // Only apply gulp-uglify plugin to JavaScript files
    .pipe(gulpif(isJavaScript, uglify()))
    .pipe(dest('output/'));
}
```

Watching Files

Дохуя всего.

Валак

Установка

```
npm install --global gulp-cli
```

перейти в директорию с проектом
npx mkdirp my-project
cd my-project

Необходимо создать пакет с json-файлом и зависимостями

```
npm init
package.json {
  "name": "gulp-web",
  "version": "1.0.0",
  "description": "test using gulp",
  "author": "Red Army"
}
```

установить gulp как зависимость в проект
npm install --save-dev gulp

Создать файл и в нём написать:

```
gulpfile.js
const {src, dest} = require("gulp");
const gulp = require("gulp");
src - где читать файлы-исходники
dest - куда экспортirовать обработанные файлы
```

Поставить дополнения

gulp-autoprefixer
npm install --save-dev gulp-autoprefixer

gulp-cssbeautify
npm install --save-dev gulp-cssbeautify
Красивое форматирование css-файла на выходе

gulp-strip-css-comments
npm install --save-dev gulp-strip-css-comments
для удаления комментариев из css-файла при создании минифицированной версии

gulp-rename

```
npm install --save-dev gulp-rename  
для изменения названия файлов (min и не min файлы)
```

gulp-sass
npm install sass gulp-sass --save-dev
компилятор sass в css

gulp-cssnano
npm install gulp-cssnano --save-dev
для сжатия css-файлов (минификатор)

gulp-rigger
npm install --save-dev gulp-rigger
для склеивания разных js-файлов (компонентов) в один

gulp-uglify
npm install --save-dev gulp-uglify
отвечает за сжатие (минификацию) js-файлов

gulp-plumber
npm install --save-dev gulp-plumber
не будут слетать гальповские таски, если есть ошибки

gulp-imagemin
npm install --save-dev gulp-imagemin
сжатие и оптимизация изображений

del
npm install --save-dev del
будет очищать папку с проектом для свежих файлов

panini
npm install --save-dev panini
работа с HTML, создание шаблонов, фрагментов кода (шаблонов), и подключать их везде.

browser-sync
npm install --save-dev browser-sync
локальный сервер для работы в прямом эфире

подключить эти плагины в gulp-файле
const {src, dest} = require("gulp");
const gulp = require("gulp");

const del = require("del");
const autoprefixer = require("gulp-autoprefixer");
const cssbeautify = require("gulp-cssbeautify");
const cssnano = require("gulp-cssnano");
const imagemin = require("gulp-imagemin");
const plumber = require("gulp-plumber");
const rename = require("gulp-rename");
const rigger = require("gulp-rigger");
const sass = require("gulp-sass");
const stripCssComments = require("gulp-strip-css-comments");
const uglify = require("gulp-uglify");
const panini = require("panini");
const sass = require("sass");
const browsersync = require("browser-sync").create();

Структура проекта

dist хранятся скомпилированные файлы

src все html файлы + рабочие файлы
assets все необходимые файлы

```
sass
  blocks
    style.scss
js
  components
    app.js
img
```

Прописать в гальпе пути

к файлам, где он должен брать исходники и перемещать в готовом виде. Пишется под импортами

```
var path = {

  build: { // это какие файлы куда копировать после обработки
    html: "dist/",
    js: "dist/assets/js/",
    css: "dist/assets/css/",
    images: "dist/assets/img"
  },

  src: { // пути для исходников
    html: "src/*.html",
    js: "src/assets/js/*.js",
    css: "src/assets/sass/style.scss",
    images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
  },

  watch: { // за какими файлами наблюдать
    html: "src/**/*.{html}",
    js: "src/assets/js/**/*.{js}",
    css: "src/assets/sass/**/*.{scss}",
    images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
  },

  // папка, в которой будут храниться скомпилированные файлы
  // она будет очищаться перед загрузкой в неё новых файлов
  clean: "./dist"
}
```

gulpfile.js от Валака

```
const {src, dest} = require("gulp");
const gulp = require("gulp");

const del = require("del");
const autoprefixer = require("gulp-autoprefixer");
const cssbeautify = require("gulp-cssbeautify");
const cssnano = require("gulp-cssnano");
const plumber = require("gulp-plumber");
const rename = require("gulp-rename");
const rigger = require("gulp-rigger");
var sass = require('gulp-sass')(require('sass'));
const stripCssComments = require("gulp-strip-css-comments");
const uglify = require("gulp-uglify");
const panini = require("panini");
const browserSync = require("browser-sync").create();
const removeComments = require("gulp-strip-css-comments");

// const sass = require("sass");
// import imagemin from "gulp-imagemin"; ошибка SyntaxError: Cannot use import statement outside a module

var path = {
  build: { // это какие файлы куда копировать после обработки
    html: "dist/",
    js: "dist/assets/js/",
    css: "dist/assets/css/",
    images: "dist/assets/img"
  },
}
```

```

src: { // пути для исходников
  html: "src/*.html",
  js: "src/assets/js/*.js",
  css: "src/assets/sass/style.scss",
  images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
},
watch: { // за какими файлами наблюдать
  html: "src/**/*.html",
  js: "src/assets/js/**/*.js",
  css: "src/assets/sass/**/*.scss",
  images: "src/assets/img/**/*.{jpg, png, svg, gif, ico}"
},
// папка, в которой будут храниться скомпилированные файлы
// она будет очищаться перед загрузкой в неё новых файлов
clean: "./dist"
};

// задачи
function browserSync(done) {
  browsersync.init({
    server: {
      baseDir: "./dist/"
    },
    port: 3000
  });
}

function browserSyncReload(done) {
  browsersync.reload();
}

function html() {
  return src(path.src.html, {base: "src/"})
    .pipe(plumber())
    .pipe(dest(path.build.html));
}

function css() {
  return src(path.src.css, {base: "src/assets/sass/"})
    .pipe(plumber())
    .pipe(sass())
    .pipe(autoprefixer({
      overrideBrowserslist: ['last 8 versions'],
      cascade: true
    }))
    .pipe(cssbeautify())
    .pipe(dest(path.build.css))
    .pipe(cssnano({
      zindex: false,
      discardComments: {
        removeAll: true
      }
    }))
    .pipe(removeComments())
    .pipe(rename({
      suffix: ".min",
      extname: ".css"
    }))
    .pipe(dest(path.build.css))
}

function js() {
  return src(path.src.js, {base: './src/assets/js/'})
    .pipe(plumber())
    .pipe(rigger())
    .pipe(gulp.dest(path.build.js))
    .pipe(uglify())
    .pipe(rename({
      suffix: ".min",
      extname: ".js"
    }))
    .pipe(dest(path.build.js))
    .pipe(browsersync.stream());
}

function images() {
  return src(path.src.images)
  // .pipe(imagemin())
}

```

```
.pipe(dest(path.build.images));
}

function clean() {
  return del(path.clean);
}

function watchFiles() {
  gulp.watch([path.watch.html], html);
  gulp.watch([path.watch.css], css);
  gulp.watch([path.watch.js], js);
  gulp.watch([path.watch.images], images);
}

const build = gulp.series(clean, gulp.parallel(html, css, js, images));
const watch = gulp.parallel(build, watchFiles, browserSync);

/* Exports Tasks */
exports.html = html;
exports.css = css;
exports.js = js;
exports.images = images;
exports.clean = clean;
exports.build = build;
exports.watch = watch;
exports.default = watch;
```

</>

Babel

Установка и запуск

В пустой папке инициализировать новый проект:

```
npm init
npm init -y инициализирует проект без дополнительных вопросов
```

Установка Babel

```
npm install --save-dev @babel/core @babel/cli
```

--save-dev

Означает, что те зависимости, которые сейчас будут установлены, устанавливаются исключительно для целей разработки, а не для работы приложения.

@babel/core

Называется name space, или пространство имён. Это позволяет структурировать те пакеты, которые разрабатывает одна организация. Babel, в данном случае, представлен двумя приложениями. В итоге это будут самые обычные пакеты.

Файл package.json будет обновлён. Появится блок «devDependencies», в него запишутся установленные пакеты.

Теперь в консоли можно использовать **npx**.

Npx – это утилита для запуска в качестве скрипта один из установленных пакетов, т.е. запустить пакет как обычное node.js приложение.

```
npx babel src --out-dir build
```

Первый параметр – откуда надо брать исходные файлы: `src`

Второй параметр – куда складывать трансформированные файлы: `build`

Результат – сообщение о том, что файл успешно преобразован. В папке будут те же файлы, которые лежат в папке src, но преобразованные. Без дополнительных модулей Babel не трансформирует код, поэтому файлы останутся без изменений.

Плагины

Babel – это модульный компилятор.

Для каждого преобразования, для каждой конструкции языка нужно установить отдельный плагин.

Для того, чтобы он начал преобразовать код, ему надо указать, какие именно аспекты языка надо преобразовывать.

Есть список плагинов на сайте Babel [здесь](#).

Babel-плагины – это самые обычные npm-пакеты и устанавливаются как всегда через npm.

После установки плагинов можно запустить трансплайнт, но обязательно с флагом --plugins, после которого перечислить те плагины, которые будут использоваться.

Важно! Перечисляются через запятую без пробелов.

```
Установить плагин как обычный пакет  
npm install --save-dev @babel/наименование
```

```
Запустить и перечислить используемые плагины  
npx babel src --out-dir build --plugins @babel/наименование, @babel/наименование
```

Пример:

```
@babel/plugin-transform-template-literals  заменяет конструкцию с бэктиками на обычные кавычки  
@babel/plugin-transform-classes  классы преобразуются в обычные функции  
@babel/plugin-transform-block-scoping  вместо const используется var
```

```
npm install --save-dev @babel/plugin-transform-template-literals @babel/plugin-transform-  
classes @babel/plugin-transform-block-scoping
```

```
npx babel src --out-dir build --plugins @babel/plugin-transform-template-literals,  
@babel/plugin-transform-classes, @babel/plugin-transform-block-scoping
```

Конфигурация .babelrc

В предыдущем виде, команда для запуска Babel становится громоздкой и её неудобно использовать, потому что надо перечислять все плагины в командной строке.

Конфигурационные файлы – простой механизм Babel, чтобы сразу указать конфигурацию для компилятора. Документация [здесь](#).

Babel поддерживает .json или .js файлы для конфигурации. Первые распространены больше.

Имя файла начинается с точки.

Внутри этого файла размещают обычный json объект с параметрами конфигурации.

Используется синтаксис json: двойные кавычки, нет висячих запятых, ключи берутся в кавычки и т.д.

```
.babelrc  
{  
  "plugins": [  
    "@babel/plugin-transform-template-literals",  
    "@babel/plugin-transform-classes",  
    "@babel/plugin-transform-block-scoping"  
  ],
```

```
"presets": ["@babel/preset-env" (или просто "@babel/env")]
}

{ "presets": [ [пресет, {targets: {браузер: версия}}] ] }
```

plugins массив плагинов для использования, чтобы не перечислять их в командной строке
presets массив пресетов

запуск
npx babel src --out-dir build

Babel Presets

Совершенно очевидно, что вручную устанавливать, хранить и поддерживать список необходимых плагинов неудобно. Чтобы упростить этот процесс, есть специальный механизм: presets.

Это просто заранее сконфигурированный список плагинов, который можно передать в Babel, чтобы не перечислять плагины по одному вручную.

@babel/preset-env

Этот пресет содержит плагины, для того чтобы поддерживать самый свежий стандарт языка script. Он всегда отражает текущую стандартную версию языка. Экспериментальные возможности, которые пока не стали стандартом, в этом пресете не учитываются, их надо устанавливать отдельно.

```
npm install --save-dev @babel/preset-env
```

См. также конфигурацию.

Сборки под браузеры

Не всегда надо трансформировать все аспекты языка, если приложение разрабатывается под какой-то определённый браузер. Намного лучше трансформировать только тот код, который целевые браузеры не смогут понять.

Чтобы определить, какие именно трансформации нужны (например, поддерживает ли браузер async функции), можно использовать [can I use](#) [здесь](#). Тем не менее, это очень трудозатратно.

Одна из возможностей пресета env – поддержка конкретных версий браузеров.

В .babelrc можно указать дополнительные опции: в массив пресетов передаётся ещё один массив. В нём первый элемент – сам пресет, а второй – объект с настройками для этого пресета.

```
{ "presets": [ [пресет, {targets: {браузер: версия}}] ] }

{
  "presets": [
    ["@babel/env", { "targets": {"edge": "18", "chrome": "74"} }]
  ]
}
```

Динамический выбор браузеров, browserslist

Кроме указания на конкретные браузеры, можно указать выражение, которое будет выбирать браузеры по каким-то признакам.

Для этого надо изменить синтаксис targets, теперь это будет массив со строкой, а не объект:

Последние 2 версии Хрома и Фокса:

```
"presets": [ ["@babel/env", { "targets": ["last 2 chrome versions"] }] ]
```

Комбинации:

```
{ "targets": ["last 2 chrome versions", "last 2 firefox versions", "last 2 ios versions"] }
```

Можно вывести в консоль во время сборки с помощью debug, под какие именно браузеры перегоняется код:

```
{
  "presets": [
    "@babel/env",
    {
      "targets": ["last 2 chrome versions", "last 2 firefox versions", "last 2 ios versions"],
      "debug": true
    }
  ],
  "plugins": ["@babel/plugin-proposal-class-properties"]
}
```

Чтобы проверить, под какие именно браузеры бабель будет писать код, можно вбить target на сайте (или приложение) browserlist [здесь](#).

Ещё:

```
"targets": ["> 0.3%"]
```

Все браузеры, у которых пользователей в мире больше, чем 0,3%

```
"targets": ["> 0.3%", "not ie > 0"]
```

Все браузеры, кроме ie

Файлы конфигурации browserslist

</>

TypeScript

Файлы создаются с расширением *.ts

Установка

TypeScript compiler

установка

```
npm install typescript
```

компиляция в js
npx tsc index.ts

создать конфигурационный файл tsconfig.json
npx tsc --init

Настройки комплаера

Документация по настройке комплаера [TWT](#).

Настройки хранятся в файле tsconfig.json

noEmitOnError	не создавать js файл, если есть ошибки в коде
outDir	указать output-папку
target	в какой стандарт ES перегонять output
noImplicitAny	нелья оставлять переменные без указания их типа
strictNullChecks	необходимо явно обрабатывать null и undefined

Типы

Явное указание типов

Примитивы

```
let value: string = 'hello'  
let value: number = 123  
let value: boolean = true  
let value: any = ...
```

Массивы

```
number[]      массив, состоящий только из цифр  
string[]  
boolean[]  
any[]
```

Параметры и возврат из функций

```
function foo(person: string, date: Date)
```

```
function getFavoriteNumber(): number {  
    ...  
    return 26;  
}
```

Объекты

interface declaration:

```
interface User {  
    name: string;  
    id: number;  
}  
  
const user: User = {  
    name: "Hayes",  
    id: 0,  
};
```

```
class UserAccount {  
    name: string;  
    id: number;
```

```
constructor(name: string, id: number) {  
    this.name = name;  
    this.id = id;  
}  
  
const user: User = new UserAccount("Murphy", 1);
```

</>

CSS для JavaScript

Единицы измерения: px, em, rem

px

em относительно текущего шрифта

rem относительно размера шрифта, указанного для элемента <html>

% относительно размера родителя

vw 1% ширины окна

vh 1% высоты окна

vmin наименьшее из (vw, vh), в IE9 обозначается vm

vmax наибольшее из (vw, vh)

Пиксель px – это базовая единица измерения.

Количество пикселей задаётся в настройках разрешения экрана. Все значения браузер в итоге пересчитает в пиксели. Пиксели могут быть дробными: 16.5px. При окончательном отображении дробные пиксели округляются и становятся целыми.

1em – текущий размер шрифта.

Можно брать любые пропорции от текущего шрифта: 2em, 0.5em и т.п. Размеры в em – относительные, они определяются по текущему контексту. Если и у родителя, и у потомка размер 1.5em, то у потомка он будет больше, потому что принимает за единицу размер родителя.

rem

Задаёт размер относительно размера шрифта, указанного для элемента <html>.

Элементы, размер которых задан в rem, не зависят друг от друга и от контекста – и этим похожи на px, а с другой стороны они все заданы относительно размера шрифта <html>.

Единица rem не поддерживается в IE8-.

Проценты %

Относительные единицы. Как правило, процент будет от значения свойства родителя с тем же названием, но не всегда:

- при установке свойства margin-left, процент берётся от ширины родительского блока, а не от его margin-left.

- при установке свойства line-height в %, процент берётся от текущего *размера шрифта*, а вовсе не от line-height родителя.
- для width/height обычно процент от ширины/высоты родителя, но при position:fixed, процент берётся от ширины/высоты окна.

Относительно экрана: vw, vh, vmin, vmax

Эти значения были созданы, в первую очередь, для поддержки мобильных устройств. Их основное преимущество – в том, что любые размеры, которые в них заданы, автоматически масштабируются при изменении размеров окна.

vw – 1% ширины окна

vh – 1% высоты окна

vmin – наименьшее из (vw, vh), в IE9 обозначается vm

vmax – наибольшее из (vw, vh)

Что понимать под размером шрифта

Он обычно чуть больше, чем расстояние от верха самой большой буквы до низа самой маленькой. В эту высоту помещается любая буква. Но при этом «хвосты» букв, таких как p, g могут заходить за это значение, то есть вылезать снизу. Поэтому обычно высоту строки делают чуть больше, чем размер шрифта.

display

Свойство display (CSS) определяет тип отображения самого элемента, к которому применяется, и его дочерних элементов.

Бокс - это прямоугольная область, являющаяся изображением элемента.

Значений много, вот основные:

```
display: none;  
  
display: block;  
display: inline;  
display: inline-block;  
  
display: flex;  
display: grid;  
display: table;
```

none

Самое простое значение. Элемент не показывается, вообще. Как будто его и нет.

block

Блок стремится расшириться на всю доступную ширину. Можно указать ширину и высоту явно.

Блочные элементы располагаются один над другим, вертикально (если нет особых свойств позиционирования, например float). Блоки прилегают друг к другу вплотную, если у них нет margin.

Это значение многие элементы имеют по умолчанию: <div>, заголовок <h1>, параграф <p>.

inline

Элементы располагаются на той же строке, последовательно.

Ширина и высота элемента определяются по содержимому. Поменять их нельзя.

Например, инлайновые элементы по умолчанию: , <a>.

inline-block

Означает элемент, который продолжает находиться в строке (inline), но при этом может иметь свойства блока.

Это значение используют, чтобы отобразить в одну строку блочные элементы, в том числе разных размеров.

Свойство vertical-align позволяет выровнять такие элементы внутри внешнего блока.

Как и инлайн-элемент:

- Располагается в строке.

- Размер устанавливается по содержимому.

Во всём остальном – это блок:

- Элемент всегда прямоугольный.

- Работают свойства width/height.

table-*

Современные браузеры (IE8+) позволяют описывать таблицу любыми элементами.

Для таблицы целиком table, для строки – table-row, для ячейки – table-cell и т.д.

С точки зрения современного CSS, обычные <table>, <tr>, <td> и т.д. – это просто элементы с предопределёнными значениями display.

Внутри ячеек table-cell свойство vertical-align выравнивает содержимое по вертикали. Это можно использовать для центрирования.

flex-box

Flexbox позволяет удобно управлять дочерними и родительскими элементами на странице, располагая их в необходимом порядке.

float

<https://learn.javascript.ru/float>

position

Позволяет сдвигать элемент со своего обычного места. Основные значения:

```
position: static  
position: relative  
position: absolute  
position: fixed
```

position: static

производится по умолчанию, в том случае, если свойство position не указано. Такая запись встречается редко и используется для переопределения других значений position.

position: relative

Относительное позиционирование сдвигает элемент относительно его текущего положения в потоке.

Необходимо указать элементу CSS-свойство position: relative и координаты left/right/top/bottom.

position: absolute

Абсолютное позиционирование делает две вещи:

1. Элемент исчезает с того места, где он должен быть и позиционируется заново. Остальные элементы занимают его место, будто этого элемента не было.
2. Координаты top/bottom/left/right для нового местоположения отсчитываются от ближайшего позиционированного родителя, т.е. родителя с позиционированием, отличным от static. Если такого родителя нет – то относительно документа.

Кроме того:

1. Ширина элемента с position: absolute устанавливается по содержимому.
2. Элемент получает display:block, который перекрывает почти все возможные display.
3. В абсолютно позиционированном элементе можно одновременно задавать противоположные границы left/right, top/bottom. Браузер растянет такой элемент до границ.

Важное отличие от relative: так как элемент удаляется со своего обычного места, то элементы под ним сдвигаются, занимая освободившееся пространство.

Иногда бывает нужно поменять элементу position на absolute, но так, чтобы элементы вокруг не сдвигались. Как правило, это делают, меняя соседей – добавляют margin/padding или вставляют в документ пустой элемент с такими же размерами.

position: fixed

Позиционирует объект точно так же, как absolute, но относительно window.

Когда страницу прокручивают, фиксированный элемент остаётся на своём месте и не прокручивается вместе со страницей.

Центрирование горизонтальное и вертикальное

Горизонтальное

text-align

Для центрирования инлайновых элементов – достаточно поставить родителю text-align: center

Для центрирования блока это уже не подойдёт, свойство просто не подействует.

margin: auto

Блок по горизонтали центрируется через margin: auto

Значение margin-left:auto;margin-right:auto заставляет браузер выделять под margin всё доступное сбоку пространство. А если и то и другое auto, то слева и справа будет одинаковый отступ, таким образом элемент окажется в середине.

Вертикальное

Вертикальное центрирование изначально не было предусмотрено в спецификации CSS и по сей день вызывает ряд проблем. Есть три основных решения.

position:absolute + margin

Центрируемый элемент позиционируем абсолютно и опускаем до середины по вертикали при помощи top:50% и смещения margin на половину ширины элемента.

Одна строка: line-height

Вертикально отцентрировать одну строку в элементе с известной высотой height можно, указав эту высоту в свойстве line-height. Это работает, но лишь до тех пор, пока строка одна, а если содержимое вдруг переносится на другую строку, то начинает выглядеть довольно уродливо.

Таблица с vertical-align

В таблицах свойство vertical-align указывает расположение содержимого ячейки. С vertical-align: middle содержимое находится по центру. Таким образом, можно обернуть нужный элемент в таблицу размера width:100%;height:100% с одной ячейкой, у которой указать vertical-align:middle, и он будет отцентрирован.

Но мы рассмотрим более красивый способ, который поддерживается во всех современных браузерах, и в IE8+. В них не обязательно делать таблицу, так как доступно значение display:table-cell. Для элемента с таким display используются те же алгоритмы вычисления ширины и центрирования, что и в TD. И, в том числе, работает vertical-align. Этот способ замечателен тем, что он не требует знания высоты элементов.

```
<div style="display: table-cell; vertical-align: middle; ... >
```

Можно и в процентах, завернув «псевдоячейку» в элемент с display:table, которому и поставим ширину:

```
<div style="display: table; width: 100%">
  <div style="display: table-cell; vertical-align: middle; height: 100px; border: 1px solid blue">
    <button>Кнопка<br>с любой высотой<br>и шириной</button>
  </div>
</div>
```

Центрирование в строке с vertical-align

Для инлайновых элементов (display:inline/inline-block), включая картинки, свойство vertical-align центрирует сам инлайн-элемент в окружающем его тексте.

```
vertical-align:
```

baseline	по умолчанию
middle	по середине
sub	вровень с <sub>
super	вровень с <sup>
text-top	верхняя граница вровень с текстом
text-bottom	нижняя граница вровень с текстом

[Центрирование с vertical-align без таблиц](#)

Какая-то ёбань, см. по ссылке.

С использованием модели flexbox

```
.outer {
  display: flex;
  justify-content: center; /* Центрирование по горизонтали */
  align-items: center;     /* Центрирование по вертикали */
}
```

Свойства font-size и line-height

font-size – размер шрифта, в частности, определяющий высоту букв.

line-height – высота строки.

Размер шрифта обычно равен расстоянию от самой верхней границы букв до самой нижней, исключая «нижние хвосты» букв, таких как р, г. При размере строки, равном font-size, строка не будет размером точно «под букву». В зависимости от шрифта, «хвосты» букв при этом могут вылезать.

Обычно размер строки делают чуть больше, чем шрифт. По умолчанию в браузерах используется специальное значение line-height: normal. Как правило, оно будет в диапазоне 1.1 - 1.25, но стандарт не гарантирует этого, он говорит лишь, что оно должно быть «разумным» (reasonable).

Множитель для line-height

Значение line-height можно указать при помощи px или em, но гораздо лучше – задать его числом.

Значение-число интерпретируется как множитель относительно размера шрифта. Например, значение с множителем line-height: 2 при font-size: 16px будет аналогично line-height: 32px (=16px*2).

Установить font-size и line-height можно **одновременно**.

Соответствующий синтаксис выглядит так:

```
минимум
font: 20px/1.5 Arial,sans-serif;

максимум
font: italic bold 20px/1.5 Arial,sans-serif;
```

Свойство white-space

Свойство white-space

управляет тем, как обрабатываются пробельные символы внутри элемента.

[white-space](#) на MDN

```
white-space: normal;    несколько пробелов объединяются в один, перенос слова только автоматом
white-space: nowrap;   переносы строки запрещены, всё будет в одну строку
```

```
white-space: pre;      текст ведёт себя, будто оформлен в тег <pre>. Перенос слов только явно.  
white-space: pre-wrap; как и pre, но строки автоматом переносятся, если не влезают  
white-space: pre-line; как и pre, но строки переносятся вручную и автоматом + пробелы объед.  
  
white-space: break-spaces;  
Сохранены переводы строк, ничего не вылезает, но пробелы интерпретированы в режиме обычного HTML. (не понял)
```

Свойство outline

Свойство outline

Задаёт дополнительную рамку вокруг элемента, за пределами его CSS-блока

Отличия от border:

1. рамка outline не участвует в блочной модели CSS: не занимает места и не меняет размер элемента.
2. можно задать только со всех сторон

свойство outline-offset задаёт отступ outline от внешней границы элемента

Часто используют для стилей :hover и других аналогичных, когда нужно выделить элемент, но чтобы ничего при этом не прыгало.

outline на MDN

```
color, style, width  
outline: green solid 3px;
```

outline-offset задаёт отступ outline от внешней границы элемента

Свойство box-sizing

Свойство box-sizing определяет как вычисляется общая ширина и высота элемента.

box-sizing на MDN

box-sizing: content-box

width и height задаются для контента, а ширина границ и внутренних отступов будет добавлена

box-sizing: border-box

width и height задают высоту ширину всего элемента (с отступами и границей)

Свойство margin

Определяет внешний отступ на всех четырёх сторонах элемента.

Особенности:

1. Вертикальные отступы поглощают друг друга, горизонтальные – нет.

Из двух вертикальных отступов выбирается и применяется наибольший.

2. Отрицательные значения margin-top и margin-left

Смещают элемент со своего обычного места. В отличие от position:relative, при сдвиге через margin соседние элементы занимают освободившееся пространство. Элемент продолжает полноценно участвовать в потоке. Исключительно полезное средство позиционирования

3. Отрицательные margin-right/bottom

Не сдвигают элемент, а «укорачивают» его. Хотя сам размер блока не уменьшается, но следующий элемент будет думать, что он меньше на указанное в margin-right/bottom значение.

[margin](#) на MDN

Свойство overflow

Управляет тем, как ведёт себя содержимое блочного элемента, если его размер превышает допустимую длину/ширину. Обычно блок увеличивается в размерах при добавлении в него элементов, заключая в себе всех потомков. Если размеры блока указаны явно и он переполняется, то можно использовать overflow.

[overflow](#) на MDN

overflow

overflow-x

overflow-y

visible

Содержимое может вылезать за границы блока.

hidden

Контент обрезается, без предоставления прокрутки.

scroll

Содержимое обрезается, элементы прокрутки отображены всегда.

auto

Предоставляется прокрутка, если содержимое переполняет блок.

Особенности height в %

Если высота внешнего блока вычисляется по содержимому, то высота в % не работает, и заменяется на height:auto. Кроме случая, когда у элемента стоит position:absolute.

то height % работает, если:

- высота внешнего блока задана точно
- внешний блок имеет position: absolute

Если у родительского элемента не установлено height, а указано min-height, то, чтобы дочерний блок занял 100% высоты, нужно родителю поставить свойство height: 1px;

Знаете ли вы селекторы?

Хорошая тема в учебнике [здесь](#).

на MDN

Фильтр по месту среди соседей

:first-child первый потомок своего родителя.

:last-child последний потомок своего родителя.

:only-child единственный потомок своего родителя, соседних элементов нет.

:nth-child(a) потомок номер a своего родителя, например :nth-child(2) – второй потомок.

:nth-child(an+b) – указание номера потомка формулой, где a,b – константы, а под n подразумевается любое целое число.

Фильтр по месту среди соседей с тем же тегом

аналогичные псевдоклассы, которые учитывают не всех соседей, а только с тем же тегом:

:first-of-type

```
:last-of-type  
:only-of-type  
:nth-of-type  
:nth-last-of-type
```

CSS-спрайты

CSS-спрайт – способ объединить много изображений в одно, чтобы:

- Сократить количество обращений к серверу.
- Если у изображений сходная палитра, то объединённое изображение будет весить меньше, чем несколько картинок.

Сдвиг фона background-position позволяет выбирать, какую именно часть спрайта видно.

Для автоматизированной сборки спрайтов используются специальные сервисы.

</>

Вопросы на собеседование

Область видимости: scope, глобальная-функциональная-блочная.

Лексическое окружение: объект LexicalEnvironment из 2-х частей.

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this и прочую служебную информацию.

Call, apply, bind меняет контекст.

this

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия. This – ключевое слово для доступа к информации внутри объекта.

Задание – это использование метода одного объекта в рамках другого объекта.

Конструктор объектов:

```
function User(name) {  
    // создание свойства  
    this.name = name;  
  
    // создание метода  
    this.read = function() {  
        console.log("Меня зовут: " + this.name);  
    };  
}  
  
// создание  
let user = new User('Valakas');
```

Как скрыть элемент со страницы?

```
visibility: hidden;  
скрыт, место остаётся занятым.  
visibility: collapse;
```

скрыт, место занимается другими элементами.

`display: none;`

Элемент скрыт, как если бы его вообще не было в HTML.

HTMLElement.hidden

Атрибут и DOM-свойство, технически работает как `style="display:none"`, но применение проще.

`opacity: 0;`

Элемент просто становится прозрачным

`color: transparent`

костыль

JavaScript

Типы данных

Типизация – это классификация информации.

Типы данных это конкретные виды информации, качественно отличающиеся друг от друга.

Есть восемь основных типов данных в JavaScript.

1. Number
2. BigInt
3. String
4. Boolean (логический тип)
5. Значение «null»
6. Значение «undefined»
7. Object
8. Symbol

Структуры данных

Про структуры есть на английском [тут](#).

Структура данных – это конкретный способ хранения и организации данных.

Для добавления, поиска, изменения и удаления данных, структура предоставляет некоторый набор функций, составляющих её интерфейс.

Некоторые структуры данных:

Массив

Стек

Очередь

Связанные списки

Графы

Деревья

Префиксные деревья

Хэш таблицы

Куча

Матрица

Массив – это структура данных, содержащая упорядоченный набор элементов и предоставляющая возможность произвольного доступа к своим элементам.

Доступ реализуется через индексы.

Стек – абстрактный тип данных, представляющий список элементов, организованный по принципу LIFO (последним пришёл – первым вышел).

Очередь – список элементов, организованный по принципу FIFO (First in First Out).

Связанный список – массив, в котором каждый элемент состоит из двух элементов – собственно данных и ссылки на следующий узел.

Графы – это набор узлов, которые соединены друг с другом в виде сети ребрами.

Деревья – это иерархическая структура, состоящая из узлов и ребер. По сути, деревья – это связанные графы без циклов.

Хэш таблицы

Хэширование – это процесс, используемый для уникальной идентификации объектов и хранения каждого объекта в заранее рассчитанном уникальном индексе (ключе).

Объект хранится в виде пары «ключ-значение», а коллекция таких элементов называется «словарем». Каждый объект можно найти с помощью этого ключа.

По сути это массив, в котором ключ представлен в виде хеш-функции.

NaN

«не число» получается, когда математическая функция сработала неверно. Это значение не равно никакому другому, включая самого себя, поэтому стандартные методы сравнения не срабатывают.

Поэтому для проверки, является ли значение NaN, используют функцию isNaN().

Оператор !!

Используется для приведения значения к логическому типу. Используя его с любым типом данных, получаем true или false.

Ещё один способ приведения к логическому значению, о котором могут спросить – функция Boolean().

Оператор %

Для получения остатка от деления. Таким образом можно проверять кратность.

Доп. вопрос: почему $1 \% 7 = 1$ или $3 \% 7 = 3$? Дело в том, что если делимое число меньше делителя, то целое число получается 0, а в остатке как раз делимое число.

Операторы И и ИЛИ

&& - находит и возвращает первое ложное значение или последний операнд, когда все истинные.

|| - находит первое истинное значение.

Оба используют короткое замыкание (ленивые вычисления?) во избежание затрат.

Строгое и нестрогое равенство

Строгое – сравнивает и значения, и типы данных, нестрогое – сравнивает только значения, приводя их к одному типу.

Сравнение объектов

Механизмы сравнения примитивов и объектов отличаются: примитивы сравниваются по значению, объекты – по ссылке (адресу в памяти). Иными словами, чтобы объект был равным другому, надо чтобы это был один и тот же адрес в памяти, т.е. один объект.

Как превратить любой тип данный в булевый?

1. использовать функцию Boolean(значение);
2. использовать двойное отрицание !!

Ложные значения в JS

Есть несколько значений, которые называют ложными

```
0  
""    пустая строка  
null  
undefined  
NaN
```

Разница между null и undefined?

Оба обозначают отсутствие данных. Только null задаётся явно, а в undefined автоматом присваивается интерпретатором во время выполнения скрипта.

Undefined – значение по умолчанию для:

- переменной, которой не было присвоено другое значение;
- функции, которая явно ничего не возвращает;
- несуществующего свойства объекта.

Разница между Function declaration и expression?

declaration – всплывает, создаётся в основном потоке документа. Создаётся интерпретатором для выполнения кода, поэтому её можно вызвать до объявления и это не повлечёт ошибку.

expression – не всплывает, записывается в переменную.

Как проверить, является ли значение массивом?

Используется метод Array.isArray()

Как определить наличие свойства в объекте

```
obj.hasOwnProperty('property');  
'property' in obj;  
obj['property'];
```

Объектная обёртка (Wrapper Objects)

В JS основные типы данных – примитивы и объекты. У примитивов есть свои методы, но реализуются они благодаря объектной обёртке: в момент исполнения кода примитив временно преобразуется в объект. Кроме null или undefined.

Как создать объект

```
// Объектный литерал  
const person1 = {  
  property: value  
}  
  
// Метод Object.create  
const person = Object.create(person1);  
  
// Function Constructor  
function Person(name) {  
  this.name = name  
}  
const person2 = new Person('Valakas');
```

{ key: value } литеральной (иницирующей) нотации

```
new Object({ key: value })  
Object.create( Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj) );  
Object.assign(target, source)  
Object.fromEntries([ key, value ])
```

Копирование объектов

Копировать объект можно следующими способами:

```
{ ...spread }  
Object.assign(target, source)  
Object.fromEntries([ key, value ])  
  
let clone = Object.create(  
  Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj)  
)
```

New

Используется с функцией-конструктором, в результате создаётся новый экземпляр класса (новый объект). Сначала создаётся пустой объект, потом привязывает к новому объекту значение `this`, затем связывает прототип для создаваемого экземпляра, и возвращает значение `this`.

Boxing и unboxing в JS

Термины упаковка и распаковка. Это связано с понятием «объект-обёртка».

Boxing:

Если у примитивного значения вызвать метод, то неявно вызывается объект-обёртка `new String` (или `num`), вызывает метод. После исполнения объект уничтожается, а нам возвращается новое значение.

Unboxing:

Распаковка – преобразование ссылочных типов данных (объектов) в примитивы. Например, это `valueOf()` и `toString()`

host-объекты и нативные объекты

host-объекты – это объекты, которые предоставляются средой выполнения, т.е. браузером или нодой. В браузере это: `Window`, `Document`, `History`, `XMLHttpRequest` и другие.

Нативные объекты – объекты, которые являются частью языка JS. `String`, `Object`, `Function` и т.д.

Можно сказать, что есть ещё и пользовательские объекты.

Типы таймеров в JS

`setTimeout(foo, ms)`

Позволяет вызывать переданную функцию один раз по истечении указанного времени.

Выключается через `clearTimeout()`

`setInterval(foo, ms)`

Позволяет вызывать переданную функцию постоянно через определённый промежуток времени. Выключается через `clearInterval()`

Оба таймера возвращают идентификатор, который может быть присвоен в переменную. Эту переменную можно передать в одну из функций, чтобы отменить таймер: `clearTimeout(id)`, `clearInterval(id)`.

Псевдомассив Arguments

Это коллекция аргументов, которые передаются в функцию. Псевдомассивом называют потому, что это подобный массиву объект: есть свойство `length` и индексы, но методы массива не доступны. С его помощью можно получить доступ к любому из аргументов.

В стрелочных функциях `arguments` не доступен.

Arguments можно превратить в обычный массив:

```
function foo(a, b) {  
    let arr1 = Array.prototype.slice.call(arguments);  
    let arr2 = [...arguments];  
    let arr3 = Array.from(arguments);  
}
```

Всплытие, hoisting

Это механизм подъёма функции или переменной в глобальную или функциональную область видимости. Это значит, что к переменным, объявленным через `var`, а также к функциям, объявленным через декларацию, можно обращаться до присвоения значения.

Область видимости (scope)

Область видимости (scope) – это область, в рамках которой доступны переменные или функции. В JS есть 3 типа областей видимости:

Глобальная

Переменные и функции, объявленные в этой области, становятся глобальными, появляются в пространстве имён и доступны из любого места в коде.

Функциональная (локальная)

Переменные, объявленные внутри функции, доступны только внутри этой функции и всем вложенными в неё функциями. За её пределами за обращение к таким переменным получится ошибка.

Блочная

Такая область видимости находится внутри фигурных скобок так называемого блока. Например, внутри `if`, `for` и т.д. Это не относится к переменным `var`. Появилась в ES6.

Кроме того, области видимости – это набор правил, по которым происходит поиск переменных. Сначала переменные ищутся в локальной области видимости, затем во внешней, пока не доходит до глобальной.

Лексическое окружение

Лексическое окружение, `LexicalEnvironment` – это специальный скрытый объект, который хранит в себе сведения о идентификаторах и переменных. Такой объект есть у каждой выполняемой функции, блока кода и скрипта.

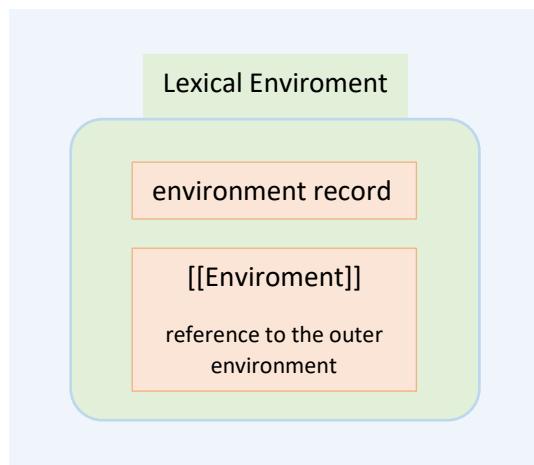
Здесь «идентификатор» — это имя переменной или функции, а «переменная» — это ссылка на объект (сюда входят и функции) или значение примитивного типа.

Объект лексического окружения состоит из двух частей:

1. Запись окружения (environment record) — объект, в котором хранятся все локальные переменные (а также другая информация, типа значение `this`).

2. Ссылка на внешнее окружение (reference to the outer environment) — ссылка, позволяющая обращаться к внешнему (родительскому) лексическому окружению. То есть к тому, что соответствует коду снаружи (снаружи от текущих фигурных скобок).

«Переменная» — это просто свойство специального внутреннего объекта: Environment Record.



Замыкание

Замыкание — запоминание функцией части окружения, где она была задана. Функция замыкает в себе идентификаторы (все, что мы определяем) из лексической области видимости.

Замыкание — это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В JavaScript, все функции изначально являются замыканиями. Они автоматически запоминают, где были созданы, с помощью скрытого свойства `[[Environment]]` и все они могут получить доступ к внешним переменным. Есть только одно исключение: [Синтаксис "new Function"](#).

WebDev:

При создании функции и объявлении внутри неё переменных, эти переменные доступны только локально внутри этой функции. На хранение таких переменных, в т.ч. аргументов, выделяется определённая память. Когда функция заканчивает своё выполнение, эта память очищается. Но если внутри одной функции создать вторую, то вложенная функция получит доступ к переменным, которые были объявлены во внешней функции. Этот механизм называется замыканием, т.е. вложенная функция замыкает на себе переменные и аргументы внешней.

Чтобы создать замыкание, вложенную функцию надо вернуть из внешней:

```
const phrase = (greeting) => {
  return (name) => `${greeting}, ${name}`;
}

let a = phrase('Hello');
console.log(a('Name'));
```

IIFE

immediately-invoked function expressions, что означает функцию, запускаемую сразу после объявления. Для этого надо взять анонимную функцию в скобки и сразу вызвать. Конструкция использовалась раньше, чтобы не загрязнять пространство имён.

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

Чистая функция

Чистая функция – это детерминированная функция без побочных эффектов.

Детерминированность – функция называется детерминированной, когда для одних и тех же входных аргументов она возвращает один и тот же результат. Например, функция, переворачивающая строку, детерминированная. Функция, возвращающая случайное число, не является детерминированной.

Побочный эффект – любые действия, изменяющие среду выполнения. Это любые файловые операции, такие как запись в файл, отправка или приём данных по сети, даже вывод в консоль или чтение файла. Кроме того, побочными эффектами считаются обращения к глобальным переменным (как на чтение, так и запись) и изменение входных аргументов в случае, когда они передаются по ссылке.

Определение от Юры:

Побочный эффект – любое изменение внешнего состояния (внешнего по отношению к функции). Например, запись значения в глобальную переменную, обновление дом-дерева, запись в кэш, в базу данных. Функция может изменять только своё собственное, локальное состояние, переменные внутри себя.

```
чистая функция
const add = (x, y) => x + y;

не чистая функция
let x = 1;
const add = (y) => {
  x += y;
};
```

Функции высшего порядка

Функции высшего порядка – это функции, которые принимают другие функции в качестве аргумента (callback) и/или возвращают другие функции.

Filter, map и reduce – три главные функции высшего порядка.

Передачу функции в качестве параметра в другую функцию хорошо демонстрирует метод [sort](#) для массивов.

Функции как объекты первого класса

Объекты первого класса (также «first-class citizen») – это элементы, которые могут быть переданы как параметр в функцию, возвращены из функции или присвоены переменной. Это всё, что может быть данными: числа, строки и остальные типы данных, в т.ч. функции.

this в JS

Ключевое слово this – это контекст вызова или ссылка на объект, который в данный момент вызывает функцию. this – ключевое слово для доступа к информации внутри объекта.

Значение this вычисляется во время выполнения кода и зависит от контекста.

[this](#) в MDN.

Это может быть:

- глобальный объект;
- объект события.

Контекст вызова: call, apply, bind

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this и прочую служебную информацию.

this – ключевое слово для доступа к информации внутри объекта. Значение this – это объект «перед точкой», который использовался для вызова метода. Значение this вычисляется во время выполнения кода и зависит от контекста.

Заимствование метода – это использование метода одного объекта в рамках (в контексте) другого объекта. В JS реализуется с помощью методов .apply, .call и .bind.

```
const res = foo.call(this, x, y);
const res = foo.apply(this, [x, y]);
const res = (foo.bind(this, x, y))();
```

Они все первым аргументом получают контекст. Разница между ними в том, что:

.call получает аргументы через запятую и сразу же выполняется;

.apply получает аргументы в массиве и сразу же выполняется;

.bind получает аргументы через запятую, но не выполняется. Можно делать частичное применение.

.call

Документация [здесь](#).

.apply

Документация [здесь](#)

.bind

В отличие от call и apply, bind не вызывает функцию сразу же, а возвращает "обёртку", которую можно вызвать позже. Документация [здесь](#).

Прототипное наследование

Все объекты в JS имеют свойство .prototype, которое является ссылкой на другой объект. Если какое-то свойство не найдено в самом объекте, это свойство ищется в прототипе этого объекта.

Чтобы создать объект, у которого нет прототипа, нужно воспользоваться Object.create(null).

Классы, статический метод

Свойства и методы являются статическими, если они не привязаны к конкретному экземпляру класса и имеют одинаковое значение вне зависимости от того, какой экземпляр ссылается на них. Часто используются для создания вспомогательных функций приложения.

Данные свойства и методы внутри класса обозначаются специальным словом «static». Они принадлежат самому классу (функции-конструктору), но не инстансам.

Для того, чтобы их вызвать внутри другого статичного метода, используется ключевое слово this.

ECMAScript

ECMAScript

Это спецификация, стандарт скриптового языка программирования. Он содержит правила и рекомендации, которые должны соблюдаться скриптовым языком, чтобы он считался совместимым с ECMAScript.

Самая революционная версия: ES6, она же ES2015, ECMA-262.

А JavaScript – это непосредственно скриптовый язык, соответствующий спецификации. Другими словами, это одна из реализаций.

Строгий режим

Смотри «Строгий режим - use strict».

Заменяет исключениями некоторые ошибки, которые интерпретатор по умолчанию пропускает. Есть 3 варианта подключения:

1. добавить строку 'use strict' в начале документа;

2. добавить строку 'use strict' в начале функции;

3. в ES6 подключить модуль, автоматом заработает при транспайлинге файла скриптов, babel добавит строгий режим автоматом.

Что меняется:

- нельзя создавать переменные без let и const;
- нельзя присваивать значения запрещённым именам (в undefined, infinity, новые свойства в нерасширяемый объект);
- нельзя удалять неудаляемые свойства;
- имена аргументов в объявлении функций не могут повторяться.

Можно ли изменить const

В константу присваивается значение, и это значение действительно нельзя переписать. Однако если речь идёт об объекте, то его свойства и методы могут быть изменены. Это происходит потому, что в саму константу записывается не значение, а ссылка на объект.

Temporal dead zone

Временная мёртвая зона. Смысл в том, что если переменные объявлены через let и const, до их объявления в коде их нельзя использовать и они находятся в мёртвой зоне. Переменные через var всплывают.

На самом деле, let и const всплывают, но чтобы было легче отлавливать ошибки, связанные со всплытием, в ES-6 для переменных и констант создали Temporal dead zone. Соответственно, переменные будут созданы только тогда, когда зайдёт в область видимости.

Тернарный оператор

Оператор, представленный знаком вопроса. Почти аналог if..else.

Цикл for...of

Данный оператор позволяет выполнить цикл обхода итерируемых сущностей. Раньше был только цикл for-in. Разница между ними в том, что ...in проходит по индексам перебираемых сущностей (также и по свойствам прототипов объекта?), а ...of – по значениям, а также перебирает Map и Set.

Template Literals

Шаблонные литералы (строки) – это способ создавать строки с помощью обратных кавычек. С ними не надо экранировать переносы строк (просто переносишь и всё, форматирование сохраняется) и можно встраивать переменные.

Set, Map, Weak

Set – это множество: массив уникальных элементов. Объекты Set позволяют сохранять уникальные значения любого типа: как примитивы, так и другие типы объектов. [Set](#)

WeakSet – аналогичен Set, но добавлять в WeakSet можно только объекты (не примитивные значения). Объект присутствует в множестве только до тех пор, пока доступен где-то ещё, или сборщик мусора удалит его. Документация [здесь](#).

Map – это коллекция пар ключ/значение, как и Object.

Отличие в том, что Map позволяет использовать ключи любого типа, включая функции, объекты и примитивы. В отличие от объектов, ключи не приводятся к строкам. [MND](#).

WeakMap – это Map-подобная коллекция пар ключ-значение, позволяющая использовать в качестве ключей только объекты, а значения могут быть произвольных типов. Пары автоматически удаляются, как только ключи (объекты) становятся недостижимы иными путями. Документация [здесь](#).

! Разница между стрелочными функциями и обычными

1. стрелочные не имеют arguments.
2. у стрелочных нет своего this, оно берётся снаружи.
3. не могут быть использованы как конструктор с new.
4. не всплывают.

Spread, rest

Синтаксис одинаковый, но противоположные задачи.

Spread – разделение коллекций на отдельные элементы.

Rest – соединение отдельных значений в массив.

Spread может использоваться при копировании объектов или массивов.

Деструктуризация

Деструктуризация – способ извлекать значения из объектов или массивов и помещать их в переменные.

Общий паттерн такой

```
let [переменная, переменная] = массив[элемент1, элемент2]
```

```
let { <ключ объекта>: <переменная>, в которую надо сохранить значение ключа} = объект
```

Async JS

Синхронные и асинхронные функции

Синхронные функции являются блокирующими (поток выполнения), а асинхронные – нет. Интерпретатор ждёт выполнения синхронных функций, а асинхронных – нет.

! Promises

Promise – это объект, возвращаемый функцией, которая ещё не завершила свою работу.

Когда вызванная функция асинхронно завершает работу, этот объект переходит в соответствующее состояние и вызывает обработчик для дальнейшей работы с результатом асинхронной операции.

Промис возвращает какое-то значение в будущем: либо результат операции, либо ошибку, по которой операция не была выполнена.

Объект Promise может находиться в трёх состояниях (свойство state, нет прямого доступа):

Pending ожидание начальное состояние, не выполнен и не отклонён.

Fulfilled исполнено операция завершена успешно.

Rejected отклонено операция завершена с ошибкой.

Результат выполнения записывается в свойство result (нет прямого доступа) и может быть равен:

undefined с самого начала

value при вызове resolve(value)

error при вызове reject(error)

Event Loop

Цикл событий. В общих чертах.

JS является однопоточным, т.е. в одну единицу времени выполнять только одну операцию.

Для такого потока выделяется область памяти, которая называется стек. В стеке хранятся фреймы – это локальные переменные и аргументы вызываемых функций.

Список событий, которые должны обрабатываться, формируют очередь событий. Когда стек освобождается, движок может обработать событие из этой очереди. Координация этого процесса и происходит в Event Loop. Движок ждёт, когда поступит новое событие.

AJAX

Asynchronous Javascript and XML.

Это не технология, а термин, который описывает несколько существующих технологий для работы на стороне клиента. С помощью AJAX, приложения могут отправлять данные на сервер и получать их асинхронно. Таким

образом происходит разделение логики отрисовки от логики обмена данными и страницы могут изменять содержимое без полной перезагрузки.

На практике для передачи и получения данных используют формат JSON.

Для работы с асинхронными запросами используют новый метод Fetch взамен XMLHttpRequest.

Same-origin policy

Переводится как «принцип одного источника». Определяет, как документ или скрипт, загруженный из одного источника, может взаимодействовать с ресурсом из другого источника. Этот принцип не позволяет JS выполнять запросы за границы домена. Это помогает изолировать вредоносные документы и сценарии.

Чтобы разрешить междоменные запросы используется CORS.

Cors

В целях безопасности, браузеры ограничивают кросс-доменные запросы: XMLHttpRequest и Fetch следуют политике одного источника, или Same-origin policy: не позволяет скрипту делать запросы за границами домена.

Cross-origin resource sharing – механизм, который использует дополнительные http-заголовки, который даёт возможность браузеру пользователю получать разрешение на доступ к ресурсам сервера или домена, который отличается от того, который браузер использует в данный момент.

DOM JS

Что такое DOM

DOM – это Document Object Model, объектная модель документа. Браузер создаёт эту модель на основании HTML-кода, полученного от сервера.

Html – это просто текст страницы, а dom – набор связанных объектов, созданных браузером при парсинге этого текста.

Это программный интерфейс (API), используя который JS может взаимодействовать с элементами на странице. Позволяет программам и скриптам получить доступ к содержимому HTML, XHTML- и XML-документов и изменять его. Точка входа – document.

Что такое события и обработчики событий?

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM).

Вот список самых часто используемых DOM-событий:

События мыши: click, contextmenu, mouseover, mouseout.

События на элементах управления: submit, focus.

События документа: DOMContentLoaded

CSS events: transitionend – когда CSS-анимация завершена.

Обработчик событий – функция, которая сработает, как только событие произошло. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя.

Распространение события (event propagation)

Это механизм, который отрабатывает, когда какое-либо событие происходит в документе. Событие распространяется от объекта Window, до вызывающего его элемента. При этом событие последовательно затрагивает всех предков целевого элемента.

Есть 3 основные фазы распространения события:

1. Capture phase

Событие начинается от корня документа и проходит по dom-дереву вниз до целевого элемента.

2. Target phase

Событие достигает целевого элемента. Обычно его называют event target.

3. Bubbling phase

Всплытие. Событие возвращается обратно до window, параллельно вызывая это событие на родительских элементах.

Делегирование событий

Делегирование – это приём разработки, когда вместо того, чтобы вешать кучу однотипных обработчиков на все элементы, добавляют один на общего предка.

e.preventDefault и e.stopPropagation

.preventDefault отключает поведение элемента по умолчанию. Например, при клике на ссылку – переход по ней, при перетаскивании объекта – создание прозрачной копии, при клике на submit - отправка формы и т.д. С помощью .preventDefault это всё можно предотвратить.

.stopPropagation отключает всплытие или погружение (т.е. распространение события).

.stopPropagation и .stopImmediatePropagation

event.stopPropagation()	выполняет текущую обработку события и останавливает всплытие
event.stopImmediatePropagation()	останавливает и текущую обработку события, и всплытие

event.target и event.currentTarget

event.target – это элемент, который вызвал событие.

event.currentTarget – элемент, в котором в данный момент происходит обработка события. Например, в результате всплытия.

event.target	ссылка на объект, который был инициатором события
event.currentTarget	ссылка на элемент, в котором в данный момент обрабатывается событие (this)

load и DOMContentLoaded

Это события. Оба срабатывают при загрузке страницы в браузере. Разница:

DOMContentLoaded событие, когда:

- браузер полностью загрузил HTML и было построено DOM-дерево, но внешние ресурсы (стили, скрипты, картинки) ещё не прогружены.

Load событие, когда

- браузер загрузил HTML и все зависимые внешние ресурсы.

Из этого следует, что он всегда срабатывает после.

Document:DOMContentLoaded event

Если встречается тег script, то событие ждёт, пока код выполнится полностью. Это нужно на случай, если скрипт что-то дописывает в документ.

Два исключения:

1. скрипты с атрибутом async не блокируют событие;

2. генерированные динамически скрипты «document.createElement('script')» не блокируют событие.

Событие не ждёт пока подгрусятся внешние стили.

Исключение: если после стилей есть тег script. Потому что script может использовать информацию из стилей: размеры и координаты и другие свойства, которыми может оперировать код.

После наступления события DOMContentLoaded происходит встроенное в браузер автозаполнение полей.

window.load

Событие «load» на объекте «window» наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы.

Теперь можно получить размеры картинок и всего остального, что не загрузилось в DOMContentLoaded.

Attribute и property у DOM-элементов

Атрибут – значение, которое в большинстве случаев добавляется в html, статично и неизменяемо.

Свойство – это вычисленное значение дом-элемента, которое может динамически изменяться.

Пример – поле input. У поля есть атрибут value, которое можно задать в dom, но потом оно будет меняться в html в одностороннем порядке.

При первом рендеринге страницы getAttribute и value будут одинаковыми, но если пользователь его изменит, то getAttribute останется старым, а value – новым.

HTMLCollection и NodeList

HTMLCollection – динамическая коллекция, которая представляет массивоподобный итерируемый объект дочерних элементов. Коллекция живая (динамическая).

NodeList – статический список нод (узлов), в который входят все найденные узлы. Не изменится, даже если изменился HTML код страницы.

```
HTMLCollection  
elem.getElementsByClassName(className)  
  
NodeList  
elem.querySelectorAll(css)
```

Методы поиска элементов в DOM

```
Element.querySelector()  
Element.querySelectorAll()
```

```
Document.getElementById()
```

```
Document.getElementsByName()  
Element.getElementsByClassName()  
Element.getElementsByTagName()
```

```
Element.closest()
```

```
Document.forms  
Document.images  
Document.links
```

Web technologies

Методы HTTP-запроса

Методы запроса определяют действие, которое хочет произвести пользователь. Основных методов 9, но на практике самые частые – первые 4:

```
GET
```

Запрос на получение данных, может только получать данные.

```
HEAD
```

Запрашивает ресурс, как и get, но без тела ответа

```
POST
```

для отправки данных. Вызывает изменение состояния и побочные эффекты на сервере.

PUT

Заменяет все текущие представления ресурса данными запроса. Используется для редактирования.

DELETE

Удаление указанных данных

PATCH

Используется для частичного изменения ресурса

OPTIONS

Используется для описания параметров для соединения с ресурсом

TRACE

Вызов возвращаемого тестового сообщения с ресурса

CONNECT

Устанавливает туннель к серверу или определённому ресурсу

Cookie

HTTP – это протокол без сохранения состояния, значит каждая новая пара запрос-ответ не связана с предыдущими. Это не удобно, потому что надо запомнить авторизацию пользователя или хранить данные корзины с товаром. Поэтому для хранения такой информации используются HTTP-cookie.

Это небольшой фрагмент данных, отправляемый сервером на браузер пользователя, который браузер может сохранить и отсылать обратно с новыми запросами к серверу.

Cookie могут использоваться для:

- управления сеансом (логины, корзины)
- мониторинга поведения пользователя
- персонализации (пользовательские предпочтения)

Получив запрос, вместе с ответом сервер может отправить заголовок «set cookie». Куки обычно запоминаются браузером и посылаются значения заголовка HTTP cookie. Для них можно создать срок действия, после которого они будут обновлены или не будут отправляться.

Feature detection, feature inference и анализ строки user-agent

Все три подхода предназначены для определения браузерных возможностей пользователя. Два последних не рекомендуется использовать, т.к. первый – самый надёжный.

Feature detection, или определение возможностей браузера – определение, поддерживает ли браузер определённый блок кода. Если блок не поддерживается, то будет выполнен аналог или полифил. Хороший пример – библиотека modernizr.

feature inference – подход применяет функцию, которая предполагает, что определённая возможность уже существует

user-agent – строка, сообщаемая браузеру, которая позволяет определить тип приложения, ОС, провайдера и т.д. Доступ к ней можно получить через navigator.userAgent.

Веб-компоненты

Веб-компоненты — совокупность стандартов, которая позволяет создавать новые, пользовательские HTML-элементы со своими свойствами, методами, инкапсулированными DOM и стилями.

custom events

API для создания собственных HTML-элементов

HTML templates

Тег позволяет реализовать изолированные dom-элементы

Shadow dom

Изолирует dom и стили в разных элементах

Html import

Импорт html документов

JS Patterns

Шаблон проектирования или паттерн – повторяемая [архитектурная конструкция](#), представляющая собой решение типовой проблемы [проектирования](#).

<https://trello.com/c/QhkSUVX6/35-javascript-design-patterns>

```
#0 Введение (Introduction)
#1 Singleton (Одиночка)
#2 Factory Method (Фабричный метод)
#3 Abstract Factory (Абстрактная фабрика)
#4 Prototype (Прототип)
#5 Builder (Строитель)
#6 Decorator (Декоратор)
#7 Facade (Фасад)
#8 Proxy (Заместитель)
#9 Adapter (Адаптер)
#10 Composite (Компоновщик)
#11 Bridge (Мост)
#12 Flyweight (Легковес)
#13 Mediator (Посредник)
#14 Iterator (Итератор)
#15 Chain of Responsibility (Цепочка обязанностей)
#16 Strategy (Стратегия)
#17 Memento (Снимок)
#18 Template Method (Шаблонный метод)
#19 Visitor (Посетитель)
#20 Command (Команда)
#21 Observer (Наблюдатель)
#22 State (Состояние)
```

Что такое HTTP

HyperText Transfer Protocol — «протокол передачи гипертекста». Это прикладной протокол для передачи гипертекстовых документов типа HTML. В настоящее время используется для передачи произвольных данных. Создан в основном для обмена данными между браузерами и серверами. Следует отметить, что это протокол без сохранения состояния, т.е. сервер не сохраняет никаких данных между парами запрос-ответ, а также следует классической клиент-серверной модели: клиент открывает соединение для создания запроса, а затем ждёт ответа.

Всё ПО разделяется на 3 категории:

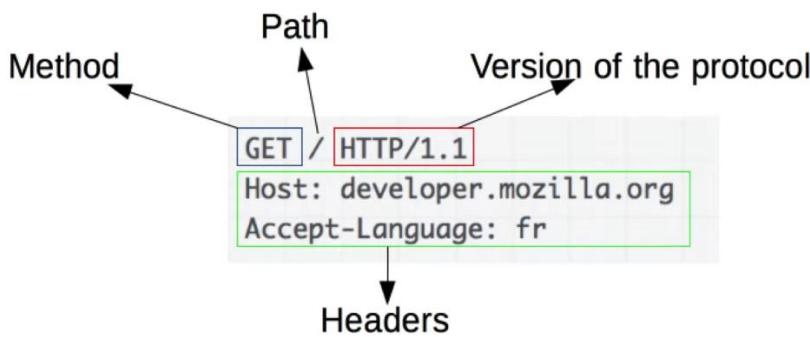
Клиент

Сервер

Прокси (посредник)

Из чего состоит HTTP запрос?

Содержит следующие элементы:



1. метод: get, post, put, delete и т.д.
2. путь к ресурсу.
3. версия HTTP протокола.
4. заголовки: дополнительная информация на сервер, не обязательны.

В запросе может быть тело, которое содержит информацию. Например, если используется запрос Put, то в теле запроса могут указаны ключи и значения, которые должны быть обновлены.

Основные принципы ООП

В современном мире под ООП имеют в виду полиморфизм, наследование, инкапсуляцию и абстракцию.

1. Абстракция

Отделение концепции от реализации. Основная идея – представить объект минимальным набором полей и методов для решения поставленной задачи.

2. Наследование

Способность объекта использовать методы родительского класса. При наследования класса, потомок получает все свойства и методы родителя. При условии, что они не переопределены и не являются приватными.

3. Инкапсуляция

Объединение и скрытие от прямого обращения данных (в т.ч. функций) в рамках одной структуры. Функции называют «методами», а данные – «свойствами».

4. Полиморфизм

способность функции обрабатывать данные разных типов. Виды полиморфизма:

- параметрический полиморфизм – исполнение одного и того же кода для разных типов данных.
- Ad-hoc-полиморфизм – обработка аргументов в зависимости от их типа, т.е. подразумевается исполнение разного кода для разных типов аргументов (например, через ветвление).

Solid

Аббревиатура для использования 5 принципов, которые используются в ООП.

single responsibility, open–closed, Liskov substitution, interface segregation и dependency inversion.

Принцип единственной ответственности (single responsibility principle)

Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

Принцип открытости/закрытости (open–closed principle)

«программные сущности ... должны быть открыты для расширения, но закрыты для модификации».

Принцип подстановки Лисков (Liskov substitution principle)

«объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы». Производный класс должен быть взаимозаменяем с родительским классом.

Принцип разделения интерфейса (interface segregation principle)

«много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения»

Принцип инверсии зависимостей (dependency inversion principle)

«Зависимость на Абстракциях. Нет зависимости от деталей»

Babel

[Babel](#) – это транспилятор. Он переписывает современный JavaScript-код в предыдущий стандарт.

У Babel две части:

1. **транспилятор**, который переписывает код в старый стандарт и после этого код отправляется на сайт.

Современные сборщики проектов типа [webpack](#) или [brunch](#) предоставляют возможность запускать транспилятор автоматически после каждого изменения кода, что позволяет экономить время.

2. **полифил**. Термин «полифил» означает, что скрипт «заполняет» пробелы и добавляет современные функции.

На WebDev говорят, что трансплаер может переписывать код с одного языка на другой.

HTML, CSS

Документация по HTML на [MDN](#). Много полезного.

Все элементы HTML [здесь](#).

Подробнее про формы и их создание [здесь](#).

HTTP: главная страница с темами [тут](#).

Обзор HTTP, как работают сайты, что такое запросы и всё такое [тут](#).

Как запускается страница?

1. Браузер делает DNS-запрос с именем домена для получения IP-адреса сервера.
2. По IP выполняется соединение с сервером.

Domain Name System, «система доменных имён» - система для получения информации о доменах. Чаще всего используется для получения IP-адреса по имени хоста.

DNS-запрос – запрос от клиента (или сервера) серверу.

Что такое Doctype и зачем он нужен?

Элемент <!DOCTYPE> предназначен для указания типа текущего документа — DTD (document type definition, описание типа документа).

Добавляется первой строкой любого HTML или XHTML документа. Служит для того, чтобы браузер понимал, как ему интерпретировать текущую веб-страницу, и в соответствии с каким стандартом осуществлять парсинг документа, поскольку HTML существует в нескольких версиях, кроме того, имеется XHTML, похожий на HTML, но отличающийся с ним по синтаксису. Исходя из этого он будет понимать, какие теги являются валидными, а какие – устаревшими.

Из DOM:

Обратите внимание, что геометрические свойства верхнего уровня (вычисление размеров окна или элементов) могут работать немного иначе, если в HTML нет <!DOCTYPE HTML>. Возможны странности. В современном HTML мы всегда должны указывать DOCTYPE.

Что такое HTML и для чего он используется?

Язык гипертекстовой разметки. Это именно язык разметки, а не программирования. Это стандартизованный язык, позволяющий создавать форматированный текст. Данный текст интерпретируется браузером, после чего браузер отображает его на странице.

Что такое XHTML?

extensible hypertext markup language — расширяемый язык гипертекстовой разметки — семейство языков разметки веб-страниц на основе XML, повторяющих и расширяющих возможности HTML 4.

Развитие XHTML остановлено; новые версии XHTML не выпускаются; рекомендуется использовать HTML.

Главное отличие XHTML от HTML заключается в обработке документа. Документы XHTML обрабатываются своим модулем (парсером) аналогично документам XML. В процессе этой обработки ошибки, допущенные разработчиками, не исправляются.

Опишите базовую структуру HTML-страницы

1. Сначала идёт Doctype.
2. Основной тег HTML. Это основная обёртка страницы. Внутри он содержит 2 основных тега.
3. Head вспомогательный блок, который содержит все необходимые данные о документе: заголовок, описание, сео информация, подключение стилей, шрифтов, мета информация. Данные, указанные внутри этого блока, не отрисовываются на странице.
4. Body – тег, который содержит всю разметку документа. Именно эта разметка будет отображаться в браузере.

Что такое семантика? Какие семантические теги знаете?

Семантика – использование правильных тегов, описывающих содержимое контента внутри себя. Семантический тег – это тег, который носит смысловое объяснение. По тегу должно быть понятно, какой контент внутри него находится.

```
header  
footer  
aside  
nav  
main  
section  
  
p  
h1  
em  
strong  
ul
```

Какая разница между тегами strong, em, и b, i?

 и имеют семантическое значение. Предназначены для логического выделения. При чтении страницы роботами на них будет сделан акцент.

 и <i> просто визуально меняют текст без добавления семантики.

Что такое CSS и для чего он используется?

CSS – Cascading Style Sheets, каскадные таблицы стилей. Это декларативный язык, который отвечает за то, как страницы выглядят в веб браузере.

CSS правило состоит из селектора и набора свойств с их значениями.

Какие варианты добавления CSS стилей на страницу?

1. использование inline-стилей, когда CSS добавляется через атрибут style.
2. использование глобальных стилей, когда в <head> помещается тег <style>, внутри которого описываются стили.
3. использование внешнего файла .css
4. импорты внутри файла стилей.

Типы позиционирования в CSS

1. static – стандартный по умолчанию.
2. relative – элемент позиционируется относительно своего текущего положения в потоке.
3. absolute – позиционируется относительно ближайшего элемента, у которого тип позиционирования не static.
4. fixed – позиционирование только относительно окна браузера.
5. sticky – в видимой области экрана ведёт себя как fixed, а при дальнейшей прокрутке скроллится вместе с родителями.

Что такое блочная модель CSS?

Блочная модель позволяет просчитать, какое итоговое пространство будет занимать элемент на странице. В неё входят:

1. сам контент
2. padding, внутренний отступ

3. margin, внешний отступ
4. border, граница элемента

Конечный размер элемента – сумма длин этих частей. Если надо указать размер, больше которого элемент не должен расти, используется свойство box-sizing: border-box.

Несколько HTML API:

- Canvas API. Создание рисунков
- Drag and Drop API
- Fetch API

Что такое валидация, какие есть типы проверок?

Валидация – это проверка документа специальной программой на соответствие установленным веб-стандартам и для обнаружения ошибок. Эти стандарты называются спецификациями, разработанными консорциумом World Wide Web (w3c).

Сначала определяется тип документа, который указывается в doctype.

Затем проверяется html код на правильность и отсутствие ошибок, в т.ч. правильность имён тегов и их вложенности.

Основные 4 типа проверки:

1. Проверка синтаксиса
2. вложенности тегов
3. наличие доктайп
4. проверка на наличие посторонних элементов или тегов.

Валидный HTML-код, валидная разметка — это HTML-код, который написан в соответствии с определёнными стандартами.

Валидатор – это программа, используя которую можно проверить HTML-код страниц и CSS-код на соответствие современным нормам. Валидатор от W3C [тут](#).

Типы input элементов в HTML

Очень большой вопрос. Надо перечислить как можно больше типов инпут и назвать несколько их особенностей. В html для коммуникации с пользователем существуют инпут-элементы, которые предназначены для получения вводимых данных. У них есть атрибут type, и в зависимости от значения этого атрибута, будет применён тот или иной тип.

[<input>](#) на MDN

`<input type="...">`

`text`

для ввода букв, цифр и специальных символов.

`password`

для ввода паролей. Вводимые символы отображаются как звёздочки.

`email`

для ввода почты пользователя. Если открыть на мобиле, то клавиатура будет содержать специальные кнопки, чтобы ускорить ввод адреса.

`number`

позволяет вводить только числовые значения. С мобилы открывает соответствующую клавиатуру.

`button` или `submit`

поле для ввода в этом случае превращается в кнопку, можно отправить форму.

`checkbox`, `radio``button`

специальные элементы для выбора нескольких возможных или одного элемента.

Date, month, daytime local

предназначены для ввода даты. Могут содержать дополнительные элементы в виде выпадающего календаря.



Как сверстать кнопку?

В произвольном месте документа. функциональность прикрутить на JS.

```
<button type="button"> Кнопка </button>
```

Для отправки формы

```
<button type="submit/post"> Кнопка </button>
```

Тоже в форме через input

```
<input type="submit" value="button">
```

Стилизовать ссылку под кнопку

```
<a href="#"> Кнопка </a>
```

Главным преимуществом HTML-элемента [`<button>`](#) в сравнении с элементом [`<input>`](#) в том, что [`<input>`](#) может принимать только простой текст, а [`<button>`](#) позволяет использовать весь HTML для создания более стилизованного текста внутри.

Что такое инлайновый стиль, можно ли его переопределить?

Инлайновый стиль – стиль, применённый к конкретному элементу и указанный в HTML-файле через атрибут style. Переопределить его можно только при помощи директивы «!important».

```
h1 {  
    color: green !important;  
}
```

Для какого тега используется атрибут alt и зачем он нужен?

Добавляется для тега картинки, чтобы в случае, если она не отображается на странице, вместо неё отображался альтернативный текст. Этот атрибут является обязательным. Кроме того, робот зачитывает его при чтении страницы.

Как семантически правильно сверстать картинку с подписью?

Можно воспользоваться двумя тегами:

```
<figure>  
    <img>  
    <figcaption> текст </figcaption>  
</figure>
```

Что такое CSS-спрайт и для чего он используется?

Это картинка, которая объединяет несколько изображений в одно большое. Обычно такой подход используется для специфичного отображения иконок. В первую очередь, это сокращает кол-во обращений к серверу: вместо

нескольких запросов достаточно сделать только 1. Дополнительно, это выполнение предзагрузки: не будет видно мигания при смене картинок.

Что такое CSS-правило?

Надо описать базовый синтаксис. CSS-правило формируется из 2-х основных составляющих:

1. селектор – правило, по которому будет происходить выборка элементов для стилизации.
2. блок объявления – структура, содержащая фигурные скобки, внутри которых описываются свойства и их значения, после чего описанные стили будут применены к найденным элементам.

Что такое селектор? Какие селекторы существуют?

Селектор – часть CSS-правила, которая сообщает браузеру, к какому элементу будет применён стиль. Важно упомянуть, что селекторы можно комбинировать.

```
tag  
.class  
#id  
  
*  
a[href="test"]  
  
h1, h2, span  
div p  
li > a  
a:hover  
li:last-child
```

Что такое специфичность селектора, как считать вес?

Специфичность – это способ, с помощью которого браузеры определяют, какие значения свойств CSS наиболее соответствуют элементу и, следовательно, будут применены. [MDN](#)

Специфичность имеет значение только в том случае, если один элемент соответствует нескольким правилам. Согласно спецификации CSS, правило для непосредственно соответствующего элемента всегда будет иметь больший приоритет, чем унаследованные от предка.

Взаимное расположение элементов в дереве документа не влияет на специфичность.

Для того, чтобы посчитать вес селектора, достаточно просчитать все входящие в него элементы (тупа математически).

1	10	100	1000
element pseudo-element	class attribute pseudo-class	id	inline

important в css

Модификатор технически не имеет специфичность, но он переопределяет всё ранее указанное.

important в html

Сильнее, чем important в CSS.

```
.title {  
    color: green !important;  
}
```

Универсальный селектор (*), комбинаторы (+, >, ~, ' ') и отрицающий псевдокласс (:not()) не влияют на специфичность. (Однако селекторы, объявленные *внутри* :not(), влияют)

Какая разница между классом и индикатором в CSS?

Добавлять стили через класс – правильный подход.

Id используется на странице только один раз (уникальный элемент).

Класс можно задавать много раз.

Id у элемента только один.

Классов у элемента может быть много.

Вес у Id – 100.

Вес у класса – 10.

Типы списков в HTML

Маркированный: ul

Нумерованный: ol

Список определений: dl, dt, dd

Какая разница между Reset.css и Normalize.css?

Практически каждый элемент имеет встроенные стили по умолчанию. Проблема в том, что в разных браузерах эти стили по умолчанию разные. Файлы с именем Reset.css и Normalize.css решают эти проблемы.

Это файлы, который подключается в самом начале документа, и либо сбрасывает все стили в ноль (стилей по умолчанию просто нет, всё на нуле), либо нормализирует их для различных браузеров, т.е. делает их одинаковыми.

Разница между margin и padding

Margin – внешний отступ, т.е. пространство от границы блока до другого элемента.

Padding – внутренний отступ, т.е. пространство от границы блока до контента.

Оба значения входят в т.н. блочную модель, которые помогают определить финальный размер блока на странице.

Как скрыть элемент со страницы?

`visibility: hidden;`

физически скрыт, но место остаётся занятым.

Для поисковых систем доступен.

`visibility: collapse;`

Элемент скрыт, место под ним занимается другими элементами.

`display: none;`

Элемент скрыт со страницы, как если бы его вообще не было в HTML.

Для поисковиков не доступен.

`HTMLElement.hidden`

Атрибут и DOM-свойство, технически работает как `style="display:none"`, но применение проще.

`opacity: 0;`

Элемент просто становится прозрачным, занимает место

Можно сделать эффект плавного появления с помощью `transition`.

`color: transparent`

костыль

Разница между display: none и visibility: hidden

Оба правила предназначены для того, чтобы скрывать элемент со страницы.

Display:none полностью убирает элемент с html-страницы. Элемент удаляется из основного потока документа. Единственное место, где он остаётся доступен – это dom-дерево. Также, контент внутри него становится недоступен для поисковых роботов.

visibility: hidden несмотря на то, что контент на странице не видно, из основного потока он не исчезает и по-прежнему занимает отведённое для него место. Также, остаётся доступен для поисковых роботов.

Как разместить элемент в середине?

position

Этот вариант надо использовать либо вместе с margin (с точными значениями смещения в px или в %, что неудобно), либо с transform.

```
position: absolute;
top: 50%;
left: 50%;

margin: -25% -25%;
transform: translate3d(-50%, -50%, 0);
```

flex

В родительский элемент прописать свойства «display» «justify-content» и «align-items»

```
.outer {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

margin

Так можно выровнять по горизонтали только блочный элемент (display: block), у которого явно задана ширина (иначе он растянется на 100% и отступать будет не от чего). По высоте выравнивание только через margin не работает.

```
.inner {
  width: 100px;
  height: 100px;
  background-color: blueviolet;
  margin: 0 auto;
}
```

Разница между блочными и строчными (инлайновыми) элементами

Основная разница в том, что блочный элемент по умолчанию всегда будет занимать всю доступную ширину экрана. Строчный элемент подстраивается под размер контента. Поэтому блочные всегда идут один под другим, а строчных в одном ряду может быть несколько.

Блочным элементам можно задавать размеры: ширину и высоту, а также margin и padding.

Для строчных элементов margin сверху и снизу не работает.

Разница между адаптивным и отзывчивым дизайном

Адаптивный дизайн – это по сути несколько версий сайта, которые загружаются пользователю в зависимости от того, на каком устройстве открыт сайт. Предполагаются несколько разных наборов файлов для разных устройств. Отзывчивый дизайн предполагает перестройку макета в зависимости от ширины экрана. Он предполагает только один набор файлов. Но правила CSS меняются в зависимости от медиа-запросов.

Что такое глобальные атрибуты в HTML5?

Атрибуты, которые используются всеми тегами. Например, class, id, dropzone, lang, style, data-.

Global attributes на mdn.

class

список разделённых пробелами классов элемента.

id

Уникальный идентификатор элемента.

data-*

пользовательские данные

contenteditable

Булевый, нужно ли предоставить пользователю возможность редактировать элемент.

spellcheck

Булевый. Может ли содержимое элемента быть проверено на наличие орфографических ошибок.

title

Содержит текст, предоставляющий консультативную информацию об элементе. Показывается пользователю в виде всплывающей подсказки.

dir

направление текста в элементе

lang

Участвует в определении языка элемента, языка написания нередактируемых элементов или языка, на котором должны быть написаны редактируемые элементы. Например, разные кавычки в цитатах в немецком и английском языках.

hidden

скрыть элемент со страницы. В DOM он остаётся.

style

инлайновые стили

tabindex

может ли элемент получать фокус, участвует ли он в последовательной навигации с клавиатуры, и если да, то в какой позиции. Значения:

отрицательное число – означает, что элемент фокусируемый, но он не может получить фокус посредством последовательной навигации с клавиатуры.

0 – означает, что элемент фокусируемый и может получить фокус посредством последовательной навигации с клавиатуры, но порядок его следования определяется платформой.

положительное значение – означает, что элемент фокусируемый и может получить фокус посредством последовательной навигации с клавиатуры. Порядок его следования определяется значением атрибута – последовательно возрастающего числа tabindex. В случае, когда несколько элементов имеют одинаковое значение атрибута tabindex, порядок их следования при навигации определяется их местом в документе.

Что такое элемент canvas и для чего он используется?

Холст – это html5 элемент, который можно использовать для вставки изображений, градиентов и сложной анимации, а также он создаёт область, в которой при помощи JS можно рисовать объекты. По сути, это API, предназначенный для отрисовки графики.

Для чего используют data-атрибуты?

В data-атрибуты можно записывать пользовательскую информацию, чтобы она парсилась вместе с HTML в DOM-дерево и её можно было использовать в JS. Минус такого подхода – угроза безопасности, потому что используя консоль разработчика эти атрибуты можно перезаписать.

Фреймворки решают эту проблему.

Разница между <script>, <script async> и <script defer>

У скриптов есть несколько особенностей:

1. когда браузер встречает тег «script», он его загружает. Если вверху страницы объёмный скрипт, он блокирует страницу. Пользователи не видят содержимое страницы, пока он не загрузится и не запустится.
 2. скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
- Есть два атрибута тега «script», чтобы решить эту проблему: defer и async.

HTMLScriptElement.defer

- загружается в фоновом режиме, браузер продолжит обрабатывать страницу.
- выполнится только после готовности DOM-дерева, но до события DOMContentLoaded.
- несколько скриптов загружаются параллельно, но выполняются последовательно, как расположены в документе.
- работает только со внешними скриптами.

HTMLScriptElement.async

- загружается в фоновом режиме
- выполнится сразу же, не ждётDOMContentLoaded.
- несколько скриптов загружаются параллельно и выполняются в порядке загрузки (не ждут друг друга).

Для чего используется элемент datalist?

Это HTML-5 элемент, который используется для создания выпадающего списка с возможностью реализации в JS его автодополнения. Добавляется вот так, [MDN](#):

```
<input list="id">
<datalist id="id">
  <option value="india">
  <option value="italy">
  <option value="iran">
</datalist>
```

Что такое вендорные префиксы и для чего они используются?

Вендорный префикс – это приставка к CSS-свойству, которая обеспечивает поддержку данного свойства браузерами, в которых оно не внедрено на постоянной основе. Т.е. свойство введено в спецификацию CSS, но в браузере оно либо на стадии разработки, либо в стадии тестирования. Либо это экспериментальное или нестандартное свойство. [Vendor Prefix](#).

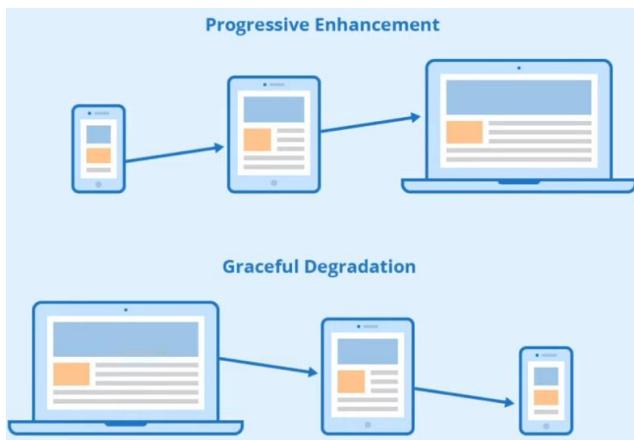
-moz-	Firefox
-webkit-	Safari, Chrome
-o-	Opera
-ms-	Internet Explorer

Разница между Progressive Enhancement и Graceful Degradation?

Оба подхода используются для создания кросс-платформенных и кросс-браузерных приложений.

Progressive Enhancement – поэтапное создание веб-интерфейсов от простого к сложному. На каждом из этапов создаётся законченный веб-интерфейс как улучшенная версия предыдущего. Например, создание приложения на мобильном устройстве, а затем его трансформация на планшет и десктоп-версию.

Graceful Degradation – наоборот, функционал сайта двигается от сложного к простому. Также, это может быть изготовление интерфейса от новейших версий браузеров к более древним.



Что такое псевдоэлементы? И для чего они используются?

Псевдоэлемент – это ключевое слово, добавляемое к селектору, которое позволяет стилизовать определённую часть выбранного элемента. [Псевдоэлементы](#).

Список стандартных псевдоэлементов

5 основных

::after	выделяет первую букву текста
::before	выделяет первую строку текста
::first-letter	выделяет первую букву текста
::first-line	выделяет первую строку текста
::selection	часть документа, выделенная пользователем

::cue	
::slotted	
::backdrop	
::placeholder	
::marker	
::spelling-error (en-US)	
::grammar-error	

Схлопывание границ, Margin collapsing

Эффект, при котором внешние отступы у блочных элементов не суммируются. В качестве отступа используется самый большой margin.

Кросбраузерность

Это корректная адаптивная вёрстка для правильного отображения в разных браузерах и на разных устройствах.

Для этого используются разные техники:

- семантическая вёрстка (?)
- использование reset или normalize
- добавление вендерных префиксов
- использование медиа запросов
- применение Progressive Enhancement и Graceful Degradation
- использование полифилов и трасплаеров (бабель)

CSS препроцессор

Это программа, которая позволяет генерировать стандартный css из собственного нестандартного синтаксиса. Можно делать вложенности, переменные, примеси и т.д. Самые популярные - Sass less.

Прогрессивный рендеринг

Обобщёное название технологий, которые используются для ускорения отрисовки веб-страниц. Например:

- ленивая загрузка изображений: сначала загружаются только те, которые находятся в видимой области экрана.
- приоретизация видимого контента: сначала подгружаются стили и скрипты видимой части.
- SSR – сервер сайда рендеринг

Чем сверстать форму?

руководство по формам
хороший перечень

<form>	атрибуты action и method обязательны
<button>	
<fieldset>	группировка нескольких элементов формы
<label>	метка для элемента формы напротив инпут-полей
<legend>	заголовок, вставленный в рамку формы или fieldset
<select>	содержит меню опций <option> внутри
<optgroup>	группирует опции внутри элемента select
<datalist>	работает как т9 с options внутри
<option>	пункт для <select>, <optgroup> или <datalist>
<textarea>	
<output>	элемент вывода чего-либо
<progress>	
<meter>	
<input>	для получения данных от пользователя
type	по умолчанию text. Подробно в этой таблице
text	однострочное текстовое поле
password	текстовое поле со скрытыми символами
number	только числовые значения. Надо для мобильных устройств.
email	для редактирования электронной почты, надо для мобильных устройств.
tel	ввод номера телефона. Надо для мобильных устройств.
submit	Кнопка для отправления формы
reset	Кнопка сброса формы
image	Кнопка отправки формы в виде изображения
<checkbox>	Флаг, выбор несколько опций. Использовать с label
<radio>	Можно выбрать только одно значение из нескольких. Группу через общее имя
<date>	Элемент для ввода даты: год, месяц и день.
<datetime>	Элемент для ввода даты и времени: час, минута, секунда и доля секунды.
<range>	Полоса с бегунком для установки значения.
<color>	интерфейс выбора цвета (в текстовом значении)

Единицы измерения в CSS

px	
em	относительно текущего шрифта
rem	относительно размера шрифта, указанного для элемента <html>
%	относительно размера родителя
vw	1% ширины окна
vh	1% высоты окна
vmin	наименьшее из (vw, vh), в IE9 обозначается vm
vmax	наибольшее из (vw, vh)

Свойства display

```
display: none;  
  
display: block;  
display: inline;  
display: inline-block;  
  
display: flex;  
display: grid;  
display: table;
```

Центрирование и выравнивание

Вертикально

text-align для инлайновых элементов
vertical-align центрирование в строке
margin для блоков

vertical-align:

baseline по умолчанию
middle по середине
sub вровень с <sub>
super вровень с <sup>
text-top верхняя граница вровень с текстом
text-bottom нижняя граница вровень с текстом

Горизонтально

position: absolute
margin
line-height

flexbox

```
.outer {  
    display: flex;  
    justify-content: center;    Центрирование по горизонтали  
    align-items: center;        Центрирование по вертикали  
}
```

Свойства font-size и line-height

font-size размер шрифта, в частности, определяющий высоту букв.
line-height высота строки.

Свойство white-space

управляет тем, как обрабатываются пробельные символы внутри элемента.

[white-space на MDN](#)

white-space: normal; несколько пробелов объединяются в один, перенос слова только автоматом
white-space: nowrap; переносы строки запрещены, всё будет в одну строку
white-space: pre; текст ведёт себя, будто оформлен в тег <pre>. Перенос слов только явно.
white-space: pre-wrap; как и pre, но строки автоматом переносятся, если не влезают
white-space: pre-line; как и pre, но строки переносятся вручную и автоматом + пробелы объед.
white-space: break-spaces;

Сохранены переводы строк, ничего не вылезает, но пробелы интерпретированы в режиме обычного HTML. (не понял)

Свойство outline

Задаёт дополнительную рамку вокруг элемента, за пределами его CSS-блока

Отличия от border:

1. рамка outline не участвует в блочной модели CSS: не занимает места и не меняет размер элемента.
2. можно задать только со всех сторон

свойство outline-offset задаёт отступ outline от внешней границы элемента.

Часто используют для стилей :hover и других аналогичных, когда нужно выделить элемент, но чтобы ничего при этом не прыгало.

[outline](#) на MDN

```
color, style, width  
outline: green solid 3px;
```

outline-offset задаёт отступ outline от внешней границы элемента

Свойство box-sizing

Свойство box-sizing определяет как вычисляется общая ширина и высота элемента.

[box-sizing](#) на MDN

```
box-sizing: content-box
```

width и height задаются для контента, а ширина границ и внутренних отступов будет добавлена

```
box-sizing: border-box
```

width и height задают высоту ширину всего элемента (с отступами и границей)

Свойство margin

Определяет внешний отступ на всех четырёх сторонах элемента.

Особенности:

1. Вертикальные отступы поглощают друг друга, горизонтальные – нет.

Из двух вертикальных отступов выбирается и применяется наибольший.

2. Отрицательные значения margin-top и margin-left

Смещают элемент со своего обычного места. В отличие от position:relative, при сдвиге через margin соседние элементы занимают освободившееся пространство. Элемент продолжает полноценно участвовать в потоке. Исключительно полезное средство позиционирования

3. Отрицательные margin-right/bottom

Не сдвигают элемент, а «укорачивают» его. Хотя сам размер блока не уменьшается, но следующий элемент будет думать, что он меньше на указанное в margin-right/bottom значение.

Свойство overflow

Управляет тем, как ведёт себя содержимое блочного элемента, если его размер превышает допустимую длину/ширину. Обычно блок увеличивается в размерах при добавлении в него элементов, заключая в себе всех потомков. Если размеры блока указаны явно и он переполняется, то можно использовать overflow.

[overflow](#) на MDN

`overflow`
`overflow-x`
`overflow-y`

`visible`
Содержимое может вылезать за границы блока.

`hidden`
Контент обрезается, без предоставления прокрутки.

`scroll`
Содержимое обрезается, элементы прокрутки отображены всегда.

`auto`
Предоставляется прокрутка, если содержимое переполняет блок.

Особенности `height` в %

Если высота внешнего блока вычисляется по содержимому, то высота в % не работает, и заменяется на `height:auto`. Кроме случая, когда у элемента стоит `position:absolute`.

то `height %` работает, если:

- высота внешнего блока задана точно
- внешний блок имеет `position:absolute`

Если у родительского элемента не установлено `height`, а указано `min-height`, то, чтобы дочерний блок занял 100% высоты, нужно родителю поставить свойство `height: 1px;`

CSS-спрайты

CSS-спрайт – способ объединить много изображений в одно, чтобы:

- Сократить количество обращений к серверу.
- Если у изображений сходная палитра, то объединённое изображение будет весить меньше, чем несколько картинок.

Сдвиг фона `background-position` позволяет выбирать, какую именно часть спрайта видно.

Для автоматизированной сборки спрайтов используются специальные сервисы.

Что такое куки? Что такое кеш?

[HTTP-кэширование \(HTTP Cache\)](#)

[HTTP-куки \(HTTP cookies\)](#)

Задачи с собеса

```
let name = 'Maria';

const sayName = () => {
  console.log(name);
}

setTimeout(() => {
  let name = '123';
  sayName();
}, 1000);
```

```
// maria
```

```
const base = 'base_1';

const buildPath = () => {
  const base = 'base_2';

  return (url) => {
    return `${base} + ${url}`;
  }
};

const getImage = (url) => {
  const base = 'base_3';

  return buildPath()(url);
};

console.log( getImage('result') );
// base_2 + result
```

HTML

CSS

Дорожная карта

Страница автора с курсами [тут](#).

Видео с картой на ютубе [тут](#).

Дашборд [тут](#).

Basics JS

JS Patterns (это шаблоны для решения типовых задач)

Data Structures ([тут](#), стек, очередь, связанные списки, графы и т.д.)

Algorithms ([тут](#))

Big O ([тут](#))

ECMAScript (ES6+) (официальная [тут](#) и статья [тут](#))

TypeScript?

Асинхронное программирование

Сетевые запросы

Basics HTML, CSS

Validity

Semantic

Accessibillity (читать [тут](#))

Flexbox (читать [тут](#))

CSS Grid

RWD

Media Queries

Mobile First

SASS (препроцессор)

Bootstrap (фреймворк)

Frameworks

React

Vue

Angular

State Management

Redux

RxJS

VueX

Tools basics

Gulp (Project Building, сборщик проектов)

Git (VCS – система контроля версий)

Вопросы

Html

что такое Doctype и зачем он?

Семантика, знать все теги. Сколько навигаций, как должны располагаться блоки? Различные типы полей input.

Вам нужно сверстать форму, какими тегами и как вы это сделаете (Дженнифер Роббинс - «HTML5. Карманный справочник»)?

CSS

Дэвид Макфарланд - Новая большая книга CSS

Блочная модель

Flexbox

Специфичность векторов

Видимость элементов

Базовое CSS Grid

Float

Позиционирование

Выравнивание и центрирование элементов

Очистка потока

Responsive design, мобилы

Кроссбраузерность

Сверстать минимум 5 шаблонов (10)

JS

Основы JS

Структуры данных: строки, числа, объекты, массивы

Методы различные, переворот слова, поиск

Замыкания и области видимости

Методы объектов и контекст вызова: call, apply, bind, чем различаются, уметь применить.

JS Dom, передвижение, поиск разными способами, вешать обработчики событий.

Знать модель распространения событий, уметь их перехватить, обработать или изменить. Это раздел «основы работы с событиями».

Синтаксис ES6 в базовых понятиях (fetch, промисы, асинк, эвейт без сурговой практики). Они не должны быть выучены наизусть.

Git

Комментировать изменения, забирать чужие, добавлять свои изменения в репозиторий, отменять их. Это 5 команд на практике.

Это минимум, который надо знать.

Желательно знать:

CSS препроцессоры: SASS или LESS, буквально основы. Вложенности и создание переменных. Максимум – миксины и инклюды.

CSS framework:

Bootstrap или фандейшн. Не надо учить все классы, просто представлять, как выглядит архитектура фреймворка и как эта архитектура используется. Правило простое: чтобы застасливовать элемент, просто добавьте к нему нужный класс. Рассчитано на то, что на проекте уже есть куча классов и ты можешь найти существующие стили и применять их.

JS framework

Знание основ, а не абсолютное использование даст перевес. Реакт, вью, энгуляр. 10 уроков туториалов максимум. Найти вопросы по ним с собеседованиям.

jQuery актуален.

JS Фреймворки

jQuery

набор функций JavaScript, фокусирующийся на взаимодействии JavaScript и HTML. Библиотека jQuery помогает легко получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, манипулировать ими. Также библиотека jQuery предоставляет удобный [API](#) для работы с [AJAX](#).

Русская документация [тут](#).

Документация [тут](#).

Подключение

Можно скачать библиотеку с оф. сайта и подключить её как модуль, но лучше использовать CDN версию. Content Delivery Network – географически распределённая сетевая инфраструктура, позволяющая оптимизировать доставку и дистрибуцию содержимого конечным пользователям.

Представляют разные компании, например майкрософт [тут](#).

Надо скопировать ссылку на нужную версию библиотеки и подключить её в body или в head вот так:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.5.1.min.js"></script>
<script src="assets/js/app.js"></script>
```

Чтобы скрипт срабатывал только после загрузки всей страницы, нужно размещать его внутри этих функций:

```
$(function() { код; });
$(document).ready(function() { код; });
```

Использование

\$ - это краткое обращение к функции jQuery.

Общий принцип работы такой: нашёл что-то через селектор и сохранил в переменную, затем навешиваешь методы.

```
let a = $("selector").method
let b = jQuery("selector").method
```

data-

С помощью data можно хранить в тегах нестандартные свойства, а потом обращаться к ним и использовать в реализации логики. Получить такое значение можно через свойство `.data('name')`:

```
<h1 data-name="abc"> ... </h1>
let a = $('h1').data('name'); // abc

<h1 data-name-property="abc"> ... </h1>
let a = $('h1').data('nameProperty'); // abc
```

В уроке Валака в data- хранились id якорей различных секций на странице, к которым осуществлялась плавная прокрутка. Если что `[attr]` – это селектор по атрибуту.

```
<nav class="nav">
  <a href="#" data-scroll="#services">Services</a>
  <a href="#" data-scroll="#clients">Clients</a>
</nav>

$("[data-scroll]").on("click", function(event) {
  event.preventDefault();

  let scrollEl = $(this).data("scroll");      // вернёт готовый #id
  let scrollElPos = $(scrollEl).offset().top; // позиция элемента от верха страницы

  $("html, body").animate({ scrollTop: scrollElPos - headerH }, 500);
});
```

Селекторы

Список селекторов [тут](#) (оригинал) и [тут](#) (перевод).

Могут быть как обычные из CSS, так и специальные типа :parent, :empty и другие.

События

Документация тут, перевод [тут](#).

Клик мышки, нажатие клавиатуры, скролл, ошибки и т.д.

Методы

.CSS

Использование любых css правил

```
$( 'h1' ).css( { 'color' : 'red' } )
```

.on

Документация [тут](#) и перевод [тут](#).

Устанавливает обработчики событий на выбранные элементы страницы.

```
.on( events [,selector] [,data], handler )
```

При клике на h1 напечатается текст:

```
$( 'h1' ).on("click", () => console.log('Click!'));
```

.animate

CSS препроцессоры

SASS

Sass — это препроцессор, метаязык на основе CSS, предназначенный для увеличения уровня абстракции CSS-кода и упрощения файлов каскадных таблиц стилей.

Препроцессор — это компьютерная программа, принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, компилятора). О данных на выходе препроцессора говорят, что они находятся в препроцессированной форме

Sass имеет два синтаксиса:

sass — отличается отсутствием фигурных скобок, в нём вложенные элементы реализованы с помощью отступов;
SCSS (Sassy CSS) — использует фигурные скобки, как и сам CSS.

CSS фреймворки

Bootstrap

или фандейшн. Не надо учить все классы, просто представлять, как выглядит архитектура фреймворка и как эта архитектура используется. Правило простое: чтобы застилизовать элемент, просто добавьте к нему нужный класс. Рассчёт на то, что на проекте уже есть куча классов и ты можешь найти существующие стили и применять их.

</>

Перечень методов (новый)

Стандартные встроенные объекты

Перечень [Стандартные встроенные объекты](#) здесь.

Array

[Array](#) на MDN

.length

Отражает количество элементов в массиве.

Методы Array

Array(7)

Передан 1 аргумент-число – создаст пустые слоты.

Array.of()

Создаёт новый экземпляр массива из переданных аргументов.

Передан аргумент-число 7 – создаст один элемент «7», а не length 7.

Array.from(arrayLike)

Создаёт новый экземпляр Array из массивоподобного или итерируемого объекта.

Array.isArray(obj)

Возвращает true, если значение является массивом, иначе возвращает false.

Методы изменения

.pop()

Удаляет последний элемент из массива и возвращает его.

.push()

Добавляет один или более элементов в конец массива и возвращает новую длину массива.

.shift()

Удаляет первый элемент из массива и возвращает его.

.unshift()

Добавляет один или более элементов в начало массива и возвращает новую длину массива.

.reverse()

Переворачивает порядок элементов в массиве – первый элемент становится последним, а последний – первым.

.sort()

Сортирует элементы массива на месте и возвращает отсортированный массив.

.splice()

Добавляет и/или удаляет элементы из массива.

.copyWithin()

Копирует последовательность элементов массива внутри массива.

.fill()

Заполняет все элементы массива от начального индекса до конечного индекса указанным значением.

Проверки и поиск

.includes()

Определяет, содержится ли в массиве указанный элемент, возвращая true или false.

.indexOf()

Возвращает первый индекс элемента внутри массива, равный указанному значению или удовлетворяющий результату проверки. -1, если значение не найдено.

.lastIndexOf()

Возвращает последний (наибольший) индекс элемента внутри массива, равный указанному значению; или -1, если значение не найдено.

.findIndex()

Возвращает искомый индекс в массиве, если элемент в массиве удовлетворяет условию проверяющей функции или -1, если такое значение не найдено.

Отличается от .indexOf тем, что ждёт функцию в качестве первого параметра для поиска более сложных, не примитивных значений внутри массива.

.every()

Возвращает true, если каждый элемент в массиве удовлетворяет условию проверяющей функции.

.some()

Возвращает `true`, если хотя бы один элемент в массиве удовлетворяет условию проверяющей функции.

.find()

Возвращает искомое значение в массиве, если элемент в массиве удовлетворяет условию проверяющей функции или `undefined`, если такое значение не найдено.

Методы доступа

.concat()

Возвращает новый массив, состоящий из данного массива, соединённого с другим массивом и/или значением (списком массивов/значений).

.join()

Объединяет все элементы массива в строку.

.slice()

Извлекает диапазон значений и возвращает его в виде нового массива.

Методы обхода

.forEach()

Вызывает функцию для каждого элемента в массиве.

.map()

Создаёт новый массив с результатами вызова указанной функции на каждом элементе данного массива.

.filter()

Создаёт новый массив со всеми элементами этого массива, для которых функция фильтрации возвращает `true`.

.reduce()

Применяет функцию к аккумулятору и каждому значению массива (слева-направо), сводя его к одному значению.

.reduceRight()

Применяет функцию к аккумулятору и каждому значению массива (справа-налево), сводя его к одному значению.

Остальные

.toSource()

Возвращает литеральное представление указанного массива; вы можете использовать это значение для создания нового массива.

.toString()

Возвращает строковое представление массива и его элементов.

.toLocaleString()

Возвращает локализованное строковое представление массива и его элементов.

.entries()

Возвращает новый объект итератора массива `Array Iterator`, содержащий пары ключ / значение для каждого индекса в массиве.

.keys()

Возвращает новый итератор массива, содержащий ключи каждого индекса в массиве.

.values()

Возвращает новый объект итератора массива `Array Iterator`, содержащий значения для каждого индекса в массиве.

`Array.prototype[@@iterator]()`

Возвращает новый объект итератора массива `Array Iterator`, содержащий значения для каждого индекса в массиве.

[Object](#) в MDN

`delete user.age`

это не свойство, а оператор.

Статические методы:

[Object.assign\(\)](#)

Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.

`Object.assign(target, ...sources)`

[Object.create\(proto, \[descriptors\]\)](#)

Создаёт новый объект с указанными объектом в качестве прототипа и дескрипторами.

`Object.create(proto, [descriptors])`

[Object.fromEntries\(\)](#)

Returns a new object from an iterable of [key, value] pairs.

[Object.is\(\)](#)

Определяет, являются ли два значения одинаковыми

Ключи и значения

[Object.keys\(\)](#)

Возвращает массив, содержащий имена всех собственных перечислимых свойств переданного объекта.

[Object.values\(\)](#)

Returns an array containing the values that correspond to all of a given object's own enumerable string properties.

[Object.entries\(\)](#)

Returns an array containing all of the [key, value] pairs of a given object's own enumerable string properties.

[Object.getOwnPropertyNames\(\)](#)

Возвращает массив, содержащий имена всех переданных объекту **собственных** перечисляемых и неперечисляемых свойств.

[Object.getOwnPropertySymbols\(\)](#)

Возвращает массив всех символьных свойств, найденных непосредственно в переданном объекте.

Прототипы

[Object.getPrototypeOf\(obj\)](#)

Возвращает прототип указанного объекта: внутреннее свойство `[[Prototype]]`.

[Object.setPrototypeOf\(obj, prototype\)](#)

Устанавливает прототип указанному объекту: внутреннее свойство `[[Prototype]]`.

[Object.create\(proto, \[descriptors\]\)](#)

Создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto`, и необязательными дескрипторами свойств `descriptors`.

[obj.__proto__](#)

Это геттер/сеттер для `[[Prototype]]`. По спецификации `__proto__` должен поддерживаться только браузерами, но по факту его поддерживают все среды. Это API не было стандартизовано.

[instance.constructor](#)

Specifies the function that creates an object's prototype.

Дескрипторы

[Object.defineProperty\(\)](#)

Добавляет к объекту именованное свойство, описываемое переданным дескриптором.

[Object.defineProperties\(\)](#)

Добавляет к объекту именованные свойства, описываемые переданными дескрипторами.

[Object.getOwnPropertyDescriptor\(\)](#)

Возвращает дескриптор свойства для именованного свойства объекта.

Запечатывание

[Object.freeze\(\)](#)

Замораживает объект: другой код не сможет удалить или изменить никакое свойство.

[Object.isFrozen\(\)](#)

Определяет, был ли объект заморожен.

[Object.preventExtensions\(\)](#)

Предотвращает любое расширение объекта.

[Object.isExtensible\(\)](#)

Определяет, разрешено ли расширение объекта.

[Object.seal\(\)](#)

Предотвращает удаление свойств объекта другим кодом.

[Object.isSealed\(\)](#)

Определяет, является ли объект запечатанным (sealed).

Методы инстансов

[.hasOwnProperty\(\)](#)

Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain.

[.isPrototypeOf\(\)](#)

Returns a boolean indicating whether the object this method is called upon is in the prototype chain of the specified object.

[.toString\(\)](#)

Returns a string representation of the object.

[.valueOf\(\)](#)

Returns the primitive value of the specified object.

[Object.observe\(\)](#)

Асинхронно наблюдает за изменениями в объекте.

String

[String на MDN](#)

[.charAt\(\)](#) Возвращает символ по указанному индексу.

[.repeat\(\)](#) возвращает строку, повторённую указанное количество раз.

[.trim\(\)](#) Обрезает пробельные символы в начале и в конце строки.

Перегон в массив

[.split\(\)](#) разбивает String на массив, разделённый указанной строкой на подстроки.

Склейки и извлечение

[.concat\(\)](#) объединяет текст двух строк и возвращает новую строку.

[.slice\(\)](#) извлекает часть строки и возвращает новую строку.

[.substring\(\)](#) Возвращает символы в строке между двумя индексами.

[.substr\(\)](#) Возвращает указанное количество символов в строке, начинающихся с указанной позиции.

Проверки

.includes() находится ли строка внутри другой строки.
.startsWith() начинается ли строка символами другой строки.
.endsWith() заканчивается ли строка символами другой строки.

.indexOf() индекс первого вхождения указанного значения или -1, если вхождений нет.
.lastIndexOf() индекс последнего вхождения указанного значения или -1, если вхождений нет.

МЦСМ компания стек делала поверку

Регистр

.toLowerCase() все символы в нижнем регистре.
.toUpperCase() все символы в верхнем регистре.

Остальные

.toString() Возвращает строковое представление указанного объекта.
.valueOf() Возвращает примитивное значение указанного объекта.

.localeCompare()

???

.prototype[@@iterator]()

Возвращает новый объект итератора Iterator, который итерируется по кодовым точкам строки и возвращает каждую кодовую точку в виде строкового значения.

Регулярные выражения

.match() Используется для сопоставления строке регулярного выражения.
.matchAll() Возвращает итератор по всем результатам при сопоставлении строки с регулярным выражением.
.replace() сопоставление строке рег. выражения и для замены совпавшей подстроки на новую
.search() Выполняет поиск совпадения регулярного выражения со строкой.

Number

Number на MDN

Статические методы

Number.isNaN() является ли переданное значение значением NaN.
Number.isFinite() является ли переданное значение конечным числом
Number.isInteger() является ли число – целым значением.

Number.parseInt() возвращает целое число из строки

Number.parseFloat() возвращает дробное число из строки

Статические свойства

Number.NEGATIVE_INFINITY
Number.POSITIVE_INFINITY

Методы инстансов

.toFixed(digits) Returns a string representing the number in fixed-point notation.

Math

Math на MDN

<code>Math.abs(x)</code>	Возвращает абсолютное значение числа.
<code>Math.max(x, y)</code>	Возвращает наибольшее число из аргументов.
<code>Math.min(x, y)</code>	Возвращает наименьшее число из аргументов.
<code>Math.pow(x, y)</code>	Возвращает x в степени y .
<code>Math.sqrt(x)</code>	Возвращает положительный квадратный корень числа.

`Math.random()` псевдослучайное число в диапазоне от 0 до 1

Округление

<code>Math.ceil(x)</code>	округление к большему целому
<code>Math.floor(x)</code>	округление к меньшему целому
<code>Math.round(x)</code>	округление к ближайшему целому
<code>Math.trunc(x)</code>	возвращает целую часть числа, убирая дробные цифры.

Случайные числа

Случайное дробное число в заданном интервале

```
function getRandomArbitrary(min, max) {
    return Math.random() * (max - min) + min;
}
```

Случайное целое число в заданном интервале

```
function getRandomInt(min, max) {
    // Максимум не включается, минимум включается
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min)) + min;
}

function getRandomIntInclusive(min, max) {
    // Максимум и минимум включаются
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Function

Function в MDN

Свойства экземпляра

`.arguments`

Массив с переданными функции аргументами. Это устаревшее свойство объекта `Function`. Вместо него используйте объект `arguments` (доступный внутри функции).

`.length`

Содержит количество аргументов в функции.

`.name`

Имя функции.

Методы экземпляра

`.call`

`.apply`

`.bind`

`.toString()` Возвращает строку с исходным кодом функции

Promise

[Promise](#) в MDN

Методы

[Promise.all\(iterable\)](#)

Ожидает исполнения всех промисов или отклонения любого из них. Если любой из промисов будет отклонён, `Promise.all` будет также отклонён.

[Promise.allSettled\(iterable\)](#)

Ожидает завершения всех полученных промисов (как исполнения так и отклонения).

[Promise.race\(iterable\)](#)

Возвращает промис, который будет выполнен или отклонён первым.

[Promise.reject\(reason\)](#)

Возвращает промис, отклонённый из-за `reason`.

[Promise.resolve\(value\)](#)

Возвращает промис, исполненный с результатом `value`.

</>

Методы

Методы объектов

[Reflect.ownKeys\(obj\)](#)

Статический метод, возвращает массив имен, а также `Symbols` собственных полей объекта. [MDN](#).

Возвращаемое значение - `Array` собственных полей объекта `target`.

```
Reflect.ownKeys(target)
```

Эквивалентом этого метода является:

```
Object.getOwnPropertyNames(target).concat(Object.getOwnPropertySymbols(target)).
```

[Object.is\(\)](#)

определяет, являются ли два значения [одинаковыми значениями](#). [MDN](#).

```
let isSame = Object.is(value1, value2);
```

работает с `Nan`

```
Object.is(NaN, NaN);  
// true;
```

```
Отличает нули  
Object.is(0, -0);  
// false
```

Метод сравнивает значения как равенство, но, в отличие от равенства, также работает в двух ситуациях:

1. Работает с NaN: `Object.is(NaN, NaN) === true`
2. Показывает, что значения 0 и -0 разные: `Object.is(0, -0) === false`, это редко используется, но технически эти значения разные.

Отличие от оператора ==

Оператор == использует приведение типов обоих операндов перед проверкой на равенство (в результате получается, что проверка "" == false даёт true).

Метод Object.is приведение типов не выполняет.

Отличие от оператора ===

Оператор === считает числовые значения -0 и +0 равными, а значение Number.NaN не равным самому себе.
Во всех других случаях Object.is(a, b) идентичен a === b.

Этот способ сравнения часто используется в спецификации JavaScript. Когда внутреннему алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is ([Определение SameValue](#)).

Symbol.toPrimitive

Символ, который описывает свойство объекта как функцию, которая вызывается при преобразовании объекта в соответствующее примитивное значение.

Функция вызывается со строковым аргументом `hint`, который передает желаемый тип примитива. Значением аргумента `hint` может быть одно из следующих значений "number", "string", и "default".

Метод Symbol.toPrimitive, обязан возвращать примитив, иначе будет ошибка.

(в отличие от методов `toValue` и `toString`, результат которых в случае не примитива будет проигнорирован).

[MDN](#).

```
const object1 = {  
  
  [Symbol.toPrimitive](hint) {  
    if (hint === 'number') {  
      return 42;  
    }  
    return null;  
  }  
};  
  
console.log(+object1);  
// 42
```

Методы для дескрипторов

Object.getOwnPropertyDescriptor()

Возвращает дескриптор свойства для именованного свойства объекта.

```
Object.getOwnPropertyDescriptor(obj, key)
```

Object.defineProperty()

Добавляет к объекту именованное свойство, описываемое переданным дескриптором. Т.е. метод определяет новое свойство или изменяет существующее и возвращает изменённый объект.

См. Дескрипторы, там подробно всё про этот метод.

Определять свойства объекта можно явно при создании, а можно через этот метод.

```
Object.defineProperty(obj, prop, descriptor)
```

obj

Объект, на котором определяется свойство

prop

Имя (ключ) определяемого или изменяемого свойства

descriptor

Дескриптор – параметры свойства или метода, который надо создать (но не само свойство как обычно). Представляет из себя объект с ключами:

configurable (по умолчанию = false)

enumerable (по умолчанию = false)

Дескриптор данных может содержать дополнительные ключи:

value (по умолчанию = undefined)

writable (по умолчанию = false)

Дескриптор доступа может содержать дополнительные ключи:

get (по умолчанию = undefined)

set (по умолчанию = undefined)

Object.defineProperties()

Добавляет к объекту именованные свойства, описываемые переданными дескрипторами.

Object.assign()

Создаёт новый объект путём копирования значений всех собственных перечислимых свойств из одного или более исходных объектов в целевой объект.

Первый аргумент target — целевой объект.

Остальные аргументы src1, ..., srcN (может быть столько, сколько нужно) являются исходными объектами.

Метод копирует свойства всех исходных объектов src1, ..., srcN в целевой объект target. Если принимающий объект уже имеет свойство с таким именем, оно будет перезаписано

Возвращает объект target.

```
Object.assign(target, ...sources)
```

```
var object = { key1: 'value1', key2: 'value2' };
var copy = Object.assign({}, object);
```

```
let user = { name: "Иван" };
let permissions1 = { canView: true };
let permissions2 = { canEdit: true };
```

```
// копируем все свойства из permissions1 и permissions2 в user
```

```
Object.assign(user, permissions1, permissions2);
// теперь user = { name: "Иван", canView: true, canEdit: true }
```

Дескрипторы свойств работают на уровне конкретных свойств.

Есть методы, которые ограничивают доступ ко всему объекту:

Object.preventExtensions(obj)

Запрещает добавлять новые свойства в объект.

`Object.seal(obj)`

Запрещает добавлять/удалять свойства. Устанавливает `configurable: false` для всех существующих свойств.

`Object.freeze(obj)`

Запрещает добавлять/удалять/изменять свойства. Устанавливает `configurable: false, writable: false` для всех существующих свойств.

А также есть **методы для их проверки**:

`Object.isExtensible(obj)`

Возвращает `false`, если добавление свойств запрещено, иначе `true`.

`Object.isSealed(obj)`

Возвращает `true`, если добавление/удаление свойств запрещено и для всех существующих свойств установлено `configurable: false`.

`Object.isFrozen(obj)`

Возвращает `true`, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено `configurable: false, writable: false`.

Методы для прототипов

`obj.__proto__`

Указывает на объект, который использовался в качестве прототипа при инстанцировании объекта.

Это API не было стандартизовано.

По сути, это геттер/сеттер для `[[Prototype]]`.

Это не то же самое, что `[[Prototype]]`.

Так работают современные методы:

```
let animal = {
  eats: true
};

// создаём новый объект с прототипом animal
let rabbit = Object.create(animal);

console.log(rabbit.eats);
// true, работает

console.log(Object.getPrototypeOf(rabbit) === animal);
// animal - это прототип для rabbit

Object.setPrototypeOf(rabbit, {});
// заменяем прототип на {}
```

`Object.getPrototypeOf(obj)`

Метод возвращает прототип (то есть, внутреннее свойство `[[Prototype]]`) указанного объекта.

Замена геттеру `_proto`.

```
Object.getPrototypeOf(obj)
```

Object.setPrototypeOf()

Метод устанавливает прототип (то есть, внутреннее свойство `[[Prototype]]`) указанного объекта в другой объект или `null`.

Замена сеттеру `_proto`.

```
Object.setPrototypeOf(obj, prototype);
```

`obj`

Объект, которому устанавливается прототип

`prototype`

Новый прототип объекта (объект или `null`)

Object.prototype

Свойство представляет объект прототипа `Object`.

В документации написано `Object`, но речь идёт именно о функции-конструкторе `Foo()`, потому что обычные объекты не имеют свойства `.prototype`.

Важный момент: `.prototype` и `._proto_` - это разные вещи.

Object.constructor

Свойство возвращает ссылку на функцию `Object`, создавшую прототип экземпляра. Обратите внимание, что значение этого свойства является ссылкой на саму функцию, а не строкой, содержащей имя функции.

Object.create(proto, [descriptors])

Создаёт пустой объект со свойством `[[Prototype]]`, указанным как `proto`, и необязательными дескрипторами свойств `descriptors`.

```
Object.create(proto[, propertiesObject])
```

`proto`

Объект, который станет прототипом вновь созданного объекта

`propertiesObject`

Необязательный. дескрипторы свойств.

propertiesObject

Про этот параметр плохо написано в документации. Здесь надо перечислить дескрипторы, как в примере ниже.

Пример я взял из задачи к уроку, поэтому прототип = `null`, но смысл понятен, как правильно прикручивать свойства к объекту:

```
let dictionary = Object.create(null, {
  toString: {
    value() {
      return Object.keys(this).join();
    }
  }
});
```

console.dir

В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования: console.dir хорошо всё покажет.

Lodash методы

_.has

Альтернатива встроенному методу Object.hasOwnProperty.

```
_.has(object, path)
```

object – объект, у которого проверяется наличие свойства

path – путь до свойства (это строка или массив).

Возвращает true — если такое свойство по указанному пути есть в объекте, и false — если нет.

Преимущества _.has:

- Более надёжна (очень сомнительно).

В JavaScript программист может переназначить свойство hasOwnProperty, и вместо true или false метод будет делать что-то другое. Функция has всегда будет возвращать булевые значения.

- Позволяет проверять цепочки свойств:

```
const object = { a: { b: { b: 2 } } };

// вызываем функцию has
console.log(_.has(object, 'a')); // => true

// вторым параметром передаём путь до вложенного свойства
console.log(_.has(object, 'a.b')); // => true

// вместо строки, путь можно передать в форме массива
console.log(_.has(object, ['a', 'b'])); // => true

// путь в форме строки
console.log(_.has(object, 'a.b.b')); // => true
// путь в форме массива
console.log(_.has(object, ['a', 'b', 'b'])); // => true
```

_.get

Документация [здесь](#).

Gets the value at path of object. If the resolved value is undefined, the defaultValue is returned in its place.

```
_.get(object, path, [defaultValue])
```

Arguments

1. object (Object): The object to query.
2. path (Array/string): The path of the property to get.
3. [defaultValue] (*): The value returned for undefined resolved values.

Returns

(*): Returns the resolved value.

_.bindAll(obj)

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например [_.bindAll\(obj\)](#) в lodash.

Методы функций

.call

Существует специальный метод функции .call, который позволяет вызывать функцию, явно устанавливая `this`. Метод вызывает функцию с указанным значением `this` и индивидуально предоставленными аргументами. Документация [здесь](#).

Функция сразу же запущена.

```
foo.call(context, arg1, arg2, ...)
```

`context`

устанавливает `this` для функции `foo`

`arg1, arg2`

аргументы функции `foo`

Обратите внимание, что `this` может не быть реальным значением, видимым этим методом: если метод является функцией в [нестрогом режиме](#), значения `null` и `undefined` будут заменены глобальным объектом, а примитивные значения будут упакованы в объекты.

```
function foo(arg) {  
  console.log(this.name, arg);  
}  
  
const obj = {name: 'Glad Valakas'};  
  
foo('мудак');  
// undefined мудак  
  
foo.call(obj, 'мудак');  
// Glad Valakas мудак
```

.apply

Метод вызывает функцию с указанным значением `this` и аргументами, предоставленными в виде массива (либо массивоподобного объекта).

разница в синтаксисе между `call` и `apply` состоит в том, что `call` ожидает список аргументов, в то время как `apply` принимает массив (или псевдомассив) аргументов.

Документация [здесь](#)

Запускает функцию сразу же.

```
fun.apply( thisArg, [arg1, arg2] )
```

`thisArg`

значение `this` (контекст), предоставляемое для вызова функции

`[arg1, arg2]`

аргументы функции `foo`

Обратите внимание, что `this` может не быть реальным значением, видимым этим методом: если метод является функцией в [нестрогом режиме](#), значения `null` и `undefined` будут заменены глобальным объектом, а примитивные значения будут упакованы в объекты.

.bind

Метод создаёт новую функцию, которая при вызове устанавливает в качестве контекста выполнения `this` предоставленное значение. В метод также передаётся набор аргументов, которые будут установлены перед переданными в привязанную функцию аргументами при её вызове.

Документация [здесь](#).

`bind` создаёт обёртку над функцией, которая подменяет контекст этой функции. Похоже на `call` и `apply`, но, в отличие от них, `bind` не вызывает функцию, а лишь возвращает "обёртку", которую можно вызвать позже.

Можно запутаться с «аргументами, которые будут установлены перед переданными в привязанную функцию аргументами». Это такие аргументы, которые можно частично применить к функции. Получается, что в обёртку записывается не только функция с контекстом, но и часть аргументов этой функции, или даже все аргументы. Получается, что через `.bind` можно применить к функции не только контекст, но и аргументы.

Это делать не обязательно, потому что можно вообще не указывать аргументы при `.bind`, а просто передать их обёртке при её вызове. Все аргументы передаются к `boundFoo` как к обычной функции при вызове в скобках.

```
let boundFoo = foo.bind(context);

let a = foo.bind(this, arg1, arg2...);

this
Значение, передаваемое в качестве this в целевую функцию при вызове привязанной функции

arg1, arg2...
Аргументы целевой функции, передаваемые перед аргументами привязанной функции
```

Пример:

```
function f() {
  alert(this);
}

var wrapped = f.bind('abc');

f();          // [object Window]
wrapped();    // abc
```

Методы фиксируются точно так же:

```
const object = {
  name: 'value',
  sayHi() {
    console.log(`Hello, ${this.name}!`)
  }
}

const a = object.sayHi.bind(object)

a()
// Hello, value!
```

[.bindAll](#)

Некоторые JS-библиотеки предоставляют встроенные функции для удобной массовой привязки контекста, например [`.bindAll\(obj\)`](#) в lodash.

Если у объекта много методов и мы планируем их активно передавать, то можно привязать контекст для них всех в цикле. После такого привязывания в цикле, все эти методы теперь можно присваивать любым другим переменным, даже другим объектам (в примере не показаны другие объекты). И они всё ещё остаются методами исходного объекта и исправно работают с ним:

_.partial

Частичное применение

Методы символов

Symbol()

Создать локальный символ.

```
let id = Symbol([описание])
```

Symbol.description

Документация [здесь](#).

The read-only description property is a string returning the optional description of Symbol objects.

Работает с локальными, глобальными и известными символами:

```
const a = Symbol('описание');
console.log( a.description );
// описание

console.log( Symbol.for('foo').description );
// "foo"

console.log(Symbol.iterator.description);
// "Symbol.iterator"
```

Symbol.for(key)

Принимает имя и создаёт (или возвращает) глобальный символ.

Для чтения (или, при отсутствии, создания) символа из глобального реестра.

Ищет глобальный символ по заданному ключу (имени, идентификатору, описанию) и возвращает его.

Если символа с таким описанием в реестре нет, то создаст новый символ для данного ключа в глобальном реестре.

Метод сначала проверяет, существует ли символ с заданным идентификатором в реестре — и возвращает его, если тот присутствует. Если символ с заданным ключом не найден, то создаст новый глобальный символ.

```
Symbol.for("foo");    // создаёт новый глобальный символ
Symbol.for("foo");    // возвращает глобальный символ, созданный ранее
```

Symbol.keyFor(sym)

Принимает глобальный символ и возвращает его имя.

Возвращает строку с ключом заданного символа, если он есть в глобальном реестре символов, либо [undefined](#), если его там нет.

Этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт undefined.

```
const globalSym = Symbol.for('foo');
// сначала создаёшь

console.log( Symbol.keyFor(globalSym) );
// получаешь "foo"
```

[Symbol.prototype](#)

Содержит прототип конструктора `Symbol`.

[Symbol.length](#)

Содержит длину, всегда равную 0 (нулю).

Для объектов:

[Object.assign](#)

Это метод клонирования объекта. В отличие от цикла `for..in`, скопирует и строковые, и символьные свойства.

[Object.getOwnPropertySymbols\(\)](#)

Возвращает массив всех символьных свойств, найденных непосредственно на переданном объекте.

Поскольку изначально никакой объект не содержит собственных символьных свойств, метод `Object.getOwnPropertySymbols()` будет возвращать пустой массив, пока вы не установите символьные свойства на вашем объекте.

Чистые таблицы

