Вопросы и ответы к собеседованию фронтенд-разработчика на JavaScript

Artur Allaev Follow
Jun 3 · 30 min read

Перевод руководства Яншун Тая «Front End Interview Handbook».

Автор иллюстрации: yangheng

Что это такое?

В отличие от типичного собеседования с разработчиками ПО, на собеседованиях фронтенд-разработчиков меньше внимания уделяется алгоритмам. Большая часть вопросов касается специфичных знаний и компетенций в таких областях, как HTML, CSS, JavaScript.

Несмотря на то, что существуют ресурсы, призванные помочь в подготовке к собеседованию, они сильно отличаются по полноте материалов от тех же ресурсов для разработчиков ПО. Среди того, что существует на сегодняшний день, наиболее полезным может быть сборник вопросов Front-end Developer Interview

Questions. К сожалению, на многие вопросы я не смог найти в сети полные и удовлетворяющие ответы. Поэтому в документе ниже я постарался самостоятельно ответить на них.

. . .

Вопросы по JavaScript

Ответы на Вопросы кандидату на должность фронтенд-разработчика—Вопросы по Javascript.

Объясните делегирование событий

Делегирование событий—это приём, заключающийся в добавлении обработчиков событий к родительскому элементу, а не к дочерним элементам. Обработчик будет срабатывать всякий раз, когда событие будет запущено на дочерних элементах благодаря всплытию событий в DOM. Преимущества этого приёма:

- Экономит объем используемой памяти, т.к. для родительского элемента требуется только один обработчик.
- Не нужно привязывать или убирать обработчики при добавлении и удалении элементов.

Ссылки

- https://davidwalsh.name/event-delegate
- https://stackoverflow.com/questions/1687296/what-is-dom-event-delegation

Объясните, как `this` работает в JavaScript

Нельзя в двух словах объяснить работу ключевого слова `this`; это одно из самых запутанных понятий в JavaScript. Говоря максимально простым языком, значение `this` зависит от того, как вызывается функция. Я прочитал много объяснений о работе `this`, и считаю объяснение Arnav Aggrawal наиболее понятным. Применяются следующие правила:

1. Если ключевое слово `new` используется при вызове функции, `this` внутри функции является совершенно новым объектом.

2. Если для вызова/создания функции используются `apply`, `call` или `bind`, то `this` внутри функции—это объект, который передается в качестве аргумента.

3. Если функция вызывается как метод, например, `obj.method()`, то `this` —это объект, к которому принадлежит функция.

4. Если функция вызывается без контекста, то есть она вызывается без условий, описанных в пунктах выше, то `this` является глобальным объектом. В браузере это объект `window`. В строгом режиме (`'use strict'`), `this` будет `undefined` вместо глобального объекта.

5. Если применяются несколько из вышеперечисленных правил, то правило, которое выше выигрывает и устанавливает значение `this`.

6. Если функция является стрелочной функцией, то она игнорирует все вышеописанные правила и получает значение `this` из лексического окружения во время ее создания.

Чтобы получить более подробное объяснение, ознакомьтесь с его статьей на Medium.

Ссылки

- https://codeburst.io/the-simple-rules-to-this-in-javascript-35d97f31bde3
- https://stackoverflow.com/a/3127440/1751946

Расскажите, как работает прототипное наследование

Этот вопрос очень часто задают на собеседованиях. Все объекты в JavaScript имеют свойство `prototype`, которое является ссылкой на другой объект. Когда происходит обращение к свойству объекта, и если свойство не найдено в этом объекте, то механизм JavaScript просматривает прототип объекта, затем прототип прототипа и т.д. До тех пор, пока не найдет определенное свойство на одном из прототипов или до тех пор, пока он не достигнет конца цепочки прототипов. Такое поведение имитирует классическое наследование, но на самом деле это скорее делегирование, чем наследование.

Ссылки

- https://www.quora.com/What-is-prototypal-inheritance/answer/Kyle-Simpson
- https://davidwalsh.name/javascript-objects

Что вы думаете о AMD против CommonJS?

Оба являются способами реализации системы модулей, которая изначально не присутствовала в JavaScript до появления ES2015. CommonJS является синхронным, в то время как AMD (Asynchronous Module Definition, асинхронное определение модуля)—соответственно, асинхронным. CommonJS разработан с учетом разработки на стороне сервера, в то время как AMD с поддержкой асинхронной загрузки модулей больше предназначена для браузеров.

Я считаю синтаксис AMD довольно многословным, а CommonJS ближе к стилю, который используется в выражениях импорта в других языках. В большинстве случаев я считаю AMD ненужным, потому что если вы разместите весь свой код в одном объединенном файле, то вы не сможете воспользоваться свойствами асинхронной загрузки. Кроме того, синтаксис CommonJS ближе к стилю написания модулей Node, и поэтому происходит меньше путаницы при переключении между клиентской и серверной разработкой на JavaScript.

Я рад, что с появлением модулей ES2015, которые поддерживают как синхронную, так и асинхронную загрузку, мы, наконец, можем придерживаться одного подхода. Несмотря на то, что они не полностью поддерживаются во всех браузерах и Node, мы можем использовать транспайлеры для преобразования нашего кода.

Ссылки

- https://auth0.com/blog/javascript-module-systems-showdown/
- https://stackoverflow.com/questions/16521471/relation-between-commonjs-amd-and-requirejs

Объясните, почему это не является IIFE: `function foo(){ }();`. Что необходимо изменить, чтобы это стало IIFE??

IIFE расшифровывается как Immediately Invoked Function Expression—немедленно вызываемое функциональное выражение. Синтаксический анализатор JavaScript читает

`function foo(){ } ();` как `function foo(){ }` и `();`, где первое выражение—это объявление функции, а второе (пара скобок)—попытка вызова функции, но так как имя не указано, он выдает ошибку `Uncaught SyntaxError: Unexpected token`.

Вот два способа исправить это, которые заключаются добавление дополнительных скобок: `(function foo(){ })()` и `(function foo(){ }())`. Выражения, начинающиеся с `function`, считаются **объявлениями функций**. Оборачивая эту функцию внутри `()`, она становится **функциональным выражением**, которое затем может быть выполнено с последующим `()`. Подобные функции не отображаются в глобальной области видимости, и вы можете даже не указывать им имя, если вы не будете на них ссылаться.

Вы также можете использовать оператор `void` — `void function foo(){ }()`. К сожалению, с таким подходом есть одна проблема. Выполнение данного выражения всегда возвращает `undefined`, поэтому, если ваше IIFE возвращает что-либо, вы не можете его использовать. Пример:

```
const foo = void function bar() { return 'foo'; }();

console.log(foo); // undefined
```

Ссылки

- http://lucybain.com/blog/2014/immediately-invoked-function-expression/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void

В чем различие между переменными, значение которых: `null`, `undefined` и не объявлено? Как бы вы проверили их на каждое из этих значений?

Необъявленные переменные создаются, когда вы присваиваете значение идентификатору, который не был ранее создан при помощи `var`, `let` или `const`. Необъявленные переменные будут определены глобально, вне текущей области видимости. В строгом режиме, будет ошибка `ReferenceError`,

когда вы попытаетесь назначить значение необъявленной переменной. Необъявленные переменные плохи так же, как и глобальные переменные. Избегайте их любой ценой! Чтобы проверить на их наличие, оберните код в блок `try` / `catch` .

```
function foo() {
  x = 1; // ReferenceError в строгом режиме
}
foo();
console.log(x); // 1
```

Переменная `undefined` —это переменная, которая была объявлена, но ей не было присвоено значение. Ее тип `undefined` . Если переменной присвоить функцию, которая не возвращает никакого значения, то переменная также будет иметь значение `undefined` . Чтобы проверить это, сравните, используя оператор строгого равенства (`===`) или `typeof` , который вернет строку `undefined` . Обратите внимание, что вам не следует использовать оператор абстрактного сравнения для проверки, так как он также вернет `true` , если значение равно `null` .

```
var foo;
console.log(foo); // undefined
console.log(foo === undefined); // true
console.log(typeof foo === 'undefined'); // true
console.log(foo == null); // true. Неправильно, не
используйте это для проверки!
function bar() {}
var baz = bar();
console.log(baz); // undefined
```

Переменной со значением `null` было явно присвоено значение `null`. Она отличается от `undefined` тем, что она была назначена явно. Чтобы проверить на `null`, просто сравните, используя оператор строгого равенства. Обратите внимание, что, как и выше, вы не должны использовать оператор абстрактного равенства (`==`) для проверки, так как он также вернет `true`, если значение равно `undefined`.

```
var foo = null;

console.log(foo === null); // true

console.log(typeof foo === 'object'); // true

console.log(foo == undefined); // true. Неправильно, не используйте это для проверки!
```

Личная привычка—я никогда не оставляю свои переменные необъявленными или неприсвоенными. Я явно назначаю им `null` после объявления, если я не собираюсь их пока использовать. Если вы используете линтер в своем рабочем процессе, он обычно также проверяет, что вы не ссылаетесь на необъявленные переменные.

Ссылки

- https://stackoverflow.com/questions/15985875/effect-of-declared-and-undeclared-variables
- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/undefined

Что такое замыкание и как/для чего его используют?

Замыкание—это комбинация функции и лексического окружения, в которой эта функция была объявлена. Слово “лексический” относится к тому факту, что лексическая область видимости использует место, где переменная объявлена в исходном коде, чтобы определить, где эта переменная доступна. Замыкания—это функции, которые имеют доступ к переменным внешней (замыкающей) функции—цепочке областей видимости даже после того, как внешняя функция вернулась.

Для чего его используют?

- Конфиденциальность данных / эмуляция скрытых методов при помощи замыканий. Обычно используется в модульном паттерне.
- Частичное применение функций или каррирование.

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
- https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36

Можете ли вы описать основное различие между циклом `.forEach` и циклом `.map()`? И в каких случаях каждый из них используется?

Чтобы понять разницу между ними, давайте посмотрим, что делает каждая функция.

forEach

- Перебирает элементы в массиве.
- Вызывает callback-функцию для каждого элемента.
- Не возвращает значение.

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

const doubled = a.forEach((num, index) => {

  // Делаем что-либо с num и/или index.

});

// doubled = undefined
```

map

- Перебирает элементы в массиве.

- “Сопоставляет” каждый элемент с новым элементом, вызывая функцию для каждого элемента, создавая в результате новый массив.

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

const doubled = a.map(num => {

  return num * 2;

});

// doubled = [2, 4, 6]
```

Основное различие между `.forEach` и `.map()` состоит в том, что `.map()` возвращает новый массив. Если вам нужен результат, но вы не хотите изменять исходный массив, `.map()` — очевидный выбор. Если вам просто нужно перебрать массив, то стоит воспользоваться `forEach`.

Ссылки

- https://codeburst.io/javascript-map-vs-foreach-f38111822c0f

В каких случаях обычно используются анонимные функции?

Они могут использоваться в IIFE для инкапсуляции кода в локальную область видимости, чтобы объявленные в ней переменные не попадали в глобальную область видимости.

```
(function() {

  // Здесь код функции.

})();
```

Как callback-функция, которая используется один раз и не должна использоваться где-либо еще. Код будет казаться более автономным и читаемым, когда обработчики будут определены прямо внутри вызывающего их кода, а не искать в другом месте, чтобы найти тело функции.

```
setTimeout(function() {
  console.log('Hello world!');
}, 1000);
```

Аргументы в конструкциях функционального программирования или Lodash (аналогично callback-функциям).

```
const arr = [1, 2, 3];
const double = arr.map(function(el) {
  return el * 2;
});
console.log(double); // [2, 4, 6]
```

Ссылки

- https://www.quora.com/What-is-a-typical-usecase-for-anonymous-functions
- https://stackoverflow.com/questions/10273185/what-are-the-benefits-to-using-anonymous-functions-instead-of-named-functions-fo

Как вы организуете свой код? (module pattern, classical inheritance)

В прошлом я использовал Backbone, который поощряет ООП подход, создавая Backbone модели и добавляя к ним методы.

Модульный паттерн до сих пор хорош, но в настоящее время я использую React/Redux, который использует однонаправленный поток данных на основе архитектуры Flux. Я создаю модели своего приложения при помощи простых объектов и пишу чистые функции для управления этими объектами. Состояние управляется при помощью экшенов и редьюсеров, как в любом другом приложении Redux.

Я избегаю использования наследования классов, где это возможно. Если же мне это необходимо сделать, то я придерживаюсь этих правил.

В чем разница между host-объектами и нативными объектами?

Нативные объекты—это объекты, которые являются частью языка JavaScript, определенного в спецификации ECMAScript, такие как `String`, `Math`, `RegExp`, `Object`, `Function` и т.д.

Хост-объекты предоставляются средой выполнения (браузером или Node), такие как `` `window` ``, `` `XMLHTTPRequest` `` и т.д.

Ссылки

- https://stackoverflow.com/questions/7614317/what-is-the-difference-between-native-objects-and-host-objects

В чем разница между: `function Person(){}`, `var person = Person()`, и `var person = new Person()`?

Этот вопрос не совсем понятен. Я полагаю, что суть вопроса о конструкторах в JavaScript. Строго говоря, `function Person(){}` — это обычное объявление функции. Принято называть с заглавной буквы функции, которые предназначены для использования в качестве конструкторов.

`var person = Person()` вызывает `Person` как функцию, а не как конструктор. Вызов как таковой является распространенной ошибкой, если функция предназначена для использования в качестве конструктора. Как правило, конструктор ничего не возвращает, поэтому при вызове конструктора как обычной функции возвращается `undefined`, и это присваивается переменной, предназначенной в качестве экземпляра.

`var person = new Person()` создает экземпляр объекта `Person` с помощью оператора `new`, который наследуется от `Person.prototype`. Альтернативой может быть использование `Object.create`, например: `Object.create(Person.prototype)`.

```
function Person(name) {
  this.name = name;
```

```
}

var person = Person('John');

console.log(person); // undefined

console.log(person.name); // Uncaught TypeError: Cannot read
property 'name' of undefined

var person = new Person('John');

console.log(person); // Person { name: "John" }

console.log(person.name); // "john"
```

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new

В чем разница между `.call` и `.apply`?

Сходство заключается в том, что и `.call`, и `.apply` используются для вызова функций, а также первый параметр будет использоваться как значение `this` внутри функции. А разница в том, что `.call` в качестве следующих аргументов принимает аргументы, разделенные запятыми, в то время как `.apply` в качестве следующих аргументов принимает массив аргументов.

```
function add(a, b) {

  return a + b;

}

console.log(add.call(null, 1, 2)); // 3

console.log(add.apply(null, [1, 2])); // 3
```

Что делает и для чего нужна функция `Function.prototype.bind`?

Взято дословно с MDN:

> *Метод bind() 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.*

По моему опыту, это наиболее полезно для привязки значения `this` в методах классов, которые вы хотите передать в другие функции. Это часто делается в компонентах React.

Ссылки

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_objects/Function/bind

В каких случаях используется `document.write()`?

`document.write()` записывает строку текста в поток документа, открытого при помощи `document.open()`. Когда `document.write()` выполняется после загрузки страницы, он вызывает `document.open`, который очищает весь документ (`<head>` и `<body>` будут удалены!) и заменяет содержимое на заданное значение параметра. Подобный подход считается опасным и не рекомендуется его использовать.

В Интернете есть несколько ответов, которые объясняют, что `document.write()` используется в коде отслеживания или когда вы хотите добавить стили, которые должны работать только при включенном JavaScript. Он даже используется в шаблоне HTML5 для параллельной загрузки скриптов и сохранения порядка выполнения! Тем не менее, я подозреваю, что эти причины могут быть устаревшими, и в наши дни они могут быть достигнуты без использования `document.write()`. Пожалуйста, поправьте меня, если я ошибаюсь по этому поводу.

Ссылки

- https://www.quirksmode.org/blog/archives/2005/06/three_javascrip_1.html
- https://github.com/h5bp/html5-boilerplate/wiki/Script-Loading-Techniques#documentwrite-script-tag

В чем разница между feature detection (определение возможностей браузера),

feature inference (предположение возможностей) и анализом строки user-agent?

Feature detection (определение возможностей браузера)

Определение возможностей браузера заключается в определении, поддерживает ли браузер определенный блок кода - и если нет, то будет выполняться другой код, так что браузер всегда сможет обеспечить работоспособность и предотвратить сбои/ошибки в некоторых браузерах. Например:

```
if ('geolocation' in navigator) {
  // Можно использовать navigator.geolocation
} else {
  // Обработка отсутствия возможности
}
```

Modernizr - отличная библиотека для обработки таких функций.

Feature inference (предположение возможностей)

Предположение возможностей проверяет на наличие определенных возможностей, как и предыдущий подход, но использует другую функцию, которая предполагает, что определенная возможность уже существует, например:

```
if (document.getElementsByTagName) {
  element = document.getElementById(id);
}
```

Этот подход не рекомендуется. Первый подход более надежен.

Строка User Agent

Это строка, сообщаемая браузером, которая позволяет узлам сетевого протокола определить тип приложения, операционную систему, поставщика программного обеспечения или версию программного обеспечения пользователя от которого исходит запрос. Доступ к ней можно получить через `navigator.userAgent`. Тем не менее, строка User Agent сложна для обработки и может быть подделана. Например, браузер Chrome идентифицируется как Chrome, так и Safari. Таким образом, чтобы обнаружить браузер Safari, вы должны проверить на наличие строки Safari и отсутствие строки Chrome. Избегайте этого метода.

Ссылки

- https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Feature_detection
- https://stackoverflow.com/questions/20104930/whats-the-difference-between-feature-detection-feature-inference-and-using-th
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Browser_detection_using_the_user_agent

Расскажите об Ajax как можно более подробно

Ajax (асинхронный JavaScript и XML) - это набор методов веб-разработки, использующих множество веб-технологий на стороне клиента для создания асинхронных веб-приложений. С помощью Ajax веб-приложения могут отправлять данные на сервер и получать их с сервера асинхронно (в фоновом режиме), не влияя на отображение и поведение открытой страницы. Отделяя уровень обмена данными от уровня представления, Ajax позволяет веб-страницам и, в частности, веб-приложениям динамически изменять содержимое без необходимости перезагрузки всей страницы. На практике для получения/передачи данных используется формат данных JSON вместо XML из-за того, что JSON основан на JavaScript.

Раньше для асинхронного взаимодействия использовали `XMLHttpRequest` API, а сейчас принято использовать `fetch` API.

Ссылки

- https://en.wikipedia.org/wiki/Ajax_(programming)

- https://developer.mozilla.org/en-US/docs/AJAX

Какие преимущества и недостатки в использовании Ajax?

Преимущества

- Повышение интерактивности. Новые данные с сервера могут быть добавлены динамически без перезагрузки всей страницы.
- Сокращение количества подключений к серверу, поскольку скрипты и таблицы стилей нужно запрашивать только один раз.
- Состояние может быть сохранено на странице. Переменные JavaScript и состояние DOM сохраняется, поскольку главная страница контейнера не перезагружается.
- Большая часть преимуществ SPA.

Недостатки

- Сложнее реализовать добавление динамической веб-страницы в закладки.
- Не работает, если в браузере отключен JavaScript.
- Некоторые поисковые роботы не выполняют JavaScript и не видят данные, загружаемые при помощи JavaScript.
- Большая часть недостатков SPA.

Объясните, как работает JSONP (и почему это не совсем AJAX)

JSONP (JSON с набивкой) - это способ, часто используемый для обхода политики ограничения домена в браузерах, потому что Ajax-запросы с текущей страницы к серверу, находящемуся в другом домене, запрещены.

JSONP работает, отправляя запрос к серверу в другом домене через тег `<script>` и обычно с параметром запроса `callback`, например: `https://example.com?callback=printData`. Затем сервер обернет данные внутри функции с именем `printData` и вернет их клиенту.

HTML:

```
<!-- https://mydomain.com -->
<script>
function printData(data) {
  console.log(`My name is ${data.name}!`);
}
</script>
<script src="https://example.com?callback=printData">
</script>
```

JS:

```
// Файл загружен с https://example.com?callback=printData
printData({ name: 'Yang Shun' });
```

У клиента должна быть функция `printData` в своей глобальной области видимости, и эта функция будет выполнена клиентом, когда будет получен ответ с сервера из другого домена.

JSONP может быть небезопасным и иметь повышенный риск. Поскольку JSONP - это действительно JavaScript, и он может делать все остальное, что может делать JavaScript, то вы должны быть уверены в надежности поставщика данных JSONP.

В наши дни, CORS является рекомендуемым подходом и JSONP является способом для его обхода.

Ссылки

- https://stackoverflow.com/a/2067584/1751946

Вы когда-нибудь использовали шаблонизацию на JavaScript? Если да, то какие библиотеки вы использовали?

Да. Handlebars, Underscore, Lodash, AngularJS, и JSX. Мне не нравилась шаблонизация в AngularJS, потому что там активно

использовались строки в директивах и легко допустить ошибку при опечатке. JSX - мой новый фаворит, так как он ближе к JavaScript и почти не имеет дополнительного синтаксиса. В настоящее время вы даже можете использовать строковые литералы шаблонов ES2015 в качестве быстрого способа создания шаблонов, не полагаясь на сторонний код.

```
const template = `<div>My name is: ${name}</div>`;
```

Однако следует помнить о возможном XSS в вышеприведенном подходе, поскольку содержимое не экранируется, в отличие от библиотек шаблонизации.

Расскажите, что такое поднятие (hoisting)

Поднятие (hoisting) - это термин, используемый для объяснения поведения объявлений переменных в вашем коде. Переменные, объявленные или инициализированные при помощи ключевого слова `var`, будут перемещены в верхнюю часть текущей области, что мы называем "поднятием". Однако, "поднимается" только объявление переменной, присвоение значения (если оно имеется) останется на прежнем месте.

Обратите внимание, что объявление фактически не перемещается - движок JavaScript анализирует объявления во время компиляции и узнает о объявлениях и их областях видимости. Просто легче понять подобное поведение, представляя объявления как перемещение наверх своей области видимости. Давайте рассмотрим несколько примеров.

```
// объявления переменных через var поднимаются.
console.log(foo); // undefined
var foo = 1;
console.log(foo); // 1
// объявления переменных через let/const НЕ поднимаются.
console.log(bar); // ReferenceError: bar is not defined
```

```
let bar = 2;

console.log(bar); // 2
```

При объявлении функции ее тело поднимается наверх, в то время как у функциональных выражений (когда переменной присваивается функция) поднимается только переменная.

```
// Объявление функции

console.log(foo); // [Function: foo]

foo(); // 'FOOOOO'

function foo() {

  console.log('FOOOOO');

}

console.log(foo); // [Function: foo]

// Функциональное выражение

console.log(bar); // undefined

bar(); // Uncaught TypeError: bar is not a function

var bar = function() {

  console.log('BARRRR');

};

console.log(bar); // [Function: bar]
```

Объясните, что такое всплытие событий (event bubbling)

Когда событие срабатывает на элементе DOM, оно попытается обработать событие (если привязан обработчик), затем событие всплывет вверх к своему родителю и это повторится снова. Подобное всплытие проходит по всем предкам элемента вплоть

до `document`. Всплытие событий является механизмом, на котором основано делегирование событий.

В чем разница между "атрибутом" (attribute) и "свойством" (property)?

Атрибуты определены в разметке HTML, а свойства определены в DOM. Чтобы проиллюстрировать разницу, представьте, что у нас есть это текстовое поле в HTML: `<input type="text" value="Hello">`.

```
const input = document.querySelector('input');

console.log(input.getAttribute('value')); // Hello

console.log(input.value); // Hello
```

Но после того, как вы измените значение текстового поля, добавив к нему "World!", будет:

```
console.log(input.getAttribute('value')); // Hello

console.log(input.value); // Hello World!
```

Ссылки

- https://stackoverflow.com/questions/6003819/properties-and-attributes-in-html

Почему не следует расширять нативные JavaScript-объекты?

Расширение встроенного/нативног объекта JavaScript означает добавление свойств/функций к его прототипу. Хотя на первый взгляд это может показаться хорошей идеей, на практике это опасно. Представьте, что ваш код использует несколько библиотек, которые расширяют `Array.prototype`, добавляя один и тот же метод `contains`. В результате код будет работать неверно, если поведение этих двух методов не будет одинаковым.

Единственный случай, при котором можно расширить нативный объект—это при создании полифила, создав собственную реализацию метода, который является частью спецификации JavaScript, но может отсутствовать в устаревших браузерах.

Ссылки

- http://lucybain.com/blog/2014/js-extending-built-in-objects/

В чем разница между событием `load` и событием `DOMContentLoaded`?

Событие `DOMContentLoaded` вызывается, когда исходный HTML-документ полностью загружен и обработан, не дожидаясь окончания загрузки таблиц стилей, изображений и скриптов.

Событие `load` происходит только после загрузки DOM и всех зависимых ресурсов.

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/Events/DOMContentLoaded
- https://developer.mozilla.org/en-US/docs/Web/Events/load

В чем разница между `==` и `===`?

`==` —это оператор абстрактного сравнения, а `===` —оператор строгого сравнения. Оператор `==` будет сравнивать на равенство после выполнения любых необходимых преобразований типов. Оператор `===` не будет выполнять преобразование типов, поэтому, если два значения не одного типа, `===` просто вернет `false`. При использовании `==` могут происходить такие странные вещи, как:

```
1 == '1'; // true
1 == [1]; // true
1 == true; // true
0 == ''; // true
0 == '0'; // true
0 == false; // true
```

Мой совет—никогда не используйте оператор `==`, за исключением удобного сравнения с `null` или `undefined`, где выражение `a == null` вернет `true`, если `a` принимает значение `null` или `undefined`.

```
var a = null;

console.log(a == null); // true

console.log(a == undefined); // true
```

Ссылки

- https://stackoverflow.com/questions/359494/which-equals-operator-vs-should-be-used-in-javascript-comparisons

Объясните same-origin policy в контексте JavaScript

Same-origin policy (принцип одинакового источника) не позволяет JavaScript выполнять запросы за границы домена. Источник определяется как комбинация схемы URI, имени хоста и номера порта. Этот принцип не позволяет вредоносному сценарию на одной странице получить доступ к конфиденциальным данным на другой через объектную модель документа этой страницы.

Ссылки

- https://en.wikipedia.org/wiki/Same-origin_policy

Сделайте так, чтобы этот код работал:

```
duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]
```

Решение:

```
function duplicate(arr) {
```

```
    return arr.concat(arr);

  }

  duplicate([1, 2, 3, 4, 5]); // [1,2,3,4,5,1,2,3,4,5]
```

Почему тернарный оператор так называется?

“Тернарный” означает три. Троичное выражение принимает три операнда: условие, выражение “then” и выражение “else”. Тернарные операторы не являются исключительными для JavaScript, и я не знаю, почему подобный вопрос был добавлен в этот список.

Ссылки

- https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

Что делает строчка ”use strict”;? Какие достоинства и недостатки от ее использования?

‘use strict’ это директива, используемая для включения строгого режима во всем скрипте или отдельных функциях.

Преимущества:

- Не позволяет случайно создавать глобальные переменные.
- Любое присваивание, которое в обычном режиме завершается неудачей, в строгом режиме выдаст исключение.
- При попытке удалить неудаляемые свойства, выдаст исключение (в то время как в нестрогом режиме никакого действия бы не произошло).
- Требует, чтобы имена параметров функции были уникальными.
- `this` в глобальной области видимости равно undefined.
- Перехватывает распространенные ошибки, выдавая исключения.

- Исключает неочевидные особенности языка.

Недостатки:

- Нельзя использовать некоторые особенности языка, к которым привыкли некоторые разработчики.
- Нет доступа к `function.caller` и `function.arguments`.
- Объединение скриптов, написанных в строгом режиме может вызвать проблемы.

В целом, я думаю, что преимущества перевешивают недостатки, и мне никогда не приходилось полагаться на функции, которые заблокированы в строгом режиме. Я бы порекомендовал использовать строгий режим.

Ссылки

- http://2ality.com/2011/10/strict-mode-hatred.html
- http://lucybain.com/blog/2014/js-use-strict/

Напишите цикл, который перебирает числа до 100, возвращая "fizz" на числа кратные 3, "buzz" на числа кратные 5 и "fizzbuzz" на числа кратные 3 и 5.

Взгляните на версию FizzBuzz от Paul Irish:

```
for (let i = 1; i <= 100; i++) {
  let f = i % 3 == 0,
    b = i % 5 == 0;
  console.log(f ? (b ? 'FizzBuzz' : 'Fizz') : b ? 'Buzz' : i);
}
```

Хотя я бы не советовал вам использовать этот код во время интервью. Просто придерживайтесь длинного, но ясного подхода. Также можете взглянуть на разные безумные реализации FizzBuzz, по ссылке ниже.

Ссылки

- https://gist.github.com/jaysonrowe/1592432

Почему считается хорошим тоном оставить глобальную область видимости (global scope) в нетронутом состоянии?

Каждый скрипт имеет доступ к глобальной области видимости, и если каждый будет использовать глобальное пространство имен для определения своих переменных, то могут возникнуть конфликты. Используйте модульный паттерн (используя IIFE) для инкапсуляции ваших переменных в локальное пространство имен.

Для чего используют событие `load`? Есть ли у этого события недостатки? Знаете ли вы какие-либо альтернативы, и в каких случаях бы стали их использовать?

Событие `load` происходит в конце процесса загрузки документа. На этом этапе все объекты в документе находятся в DOM, и все изображения, скрипты и ссылки загрузились.

Событие DOM `DOMContentLoaded` будет запущено после создания DOM для страницы, но не будет ждать окончания загрузки других ресурсов. Оно предпочтительно в тех случаях, когда вам не нужно загружать страницу целиком перед инициализацией.

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onload

Расскажите, что такое одностраничное приложение, и как сделать его SEO-оптимизированным.

Текст ниже взят из замечательного руководства по фронтенду от Grab, который по счастливой случайности тоже был написан мной.

В наши дни веб-разработчики называют свои продукты веб-приложениями, а не веб-сайтами. Хотя между этими двумя терминами нет строгой разницы, веб-приложения, как правило, очень интерактивны и динамичны, что позволяет пользователю выполнять действия и получать мгновенный ответ. Традиционно

браузер получает HTML с сервера и отображает его. Когда пользователь переходит на другой URL-адрес, требуется полное обновление страницы, и сервер отправляет свежий HTML-код на новую страницу. Это называется рендерингом на стороне сервера.

Однако в современных SPA вместо этого используется рендеринг на стороне клиента. Браузер загружает начальную страницу с сервера вместе со скриптами (фреймворками, библиотеками, кодом приложения) и таблицами стилей, необходимыми для всего приложения. Когда пользователь переходит на другие страницы, обновление страницы не происходит. URL-адрес страницы обновляется при помощи HTML5 History API. Новые данные, необходимые для страницы (обычно в формате JSON), извлекаются браузером посредством запросов AJAX к серверу. Затем SPA динамически обновляет страницу данными через JavaScript, которые были получены при начальной загрузке страницы. Эта модель похожа на работу нативных мобильных приложений.

Преимущества:

- Приложение становится более отзывчивым, и пользователи не видят мерцание при навигации, т.к. страница не обновляется целиком.
- На сервер поступает меньше HTTP-запросов, так как одни и те же ресурсы не нужно загружать снова для каждой загрузки страницы.
- Четкое разделение на клиент и сервер. Вы можете легко создавать новые клиентские приложения для разных платформ (например, для мобильных устройств, чат-ботов, умных часов) без необходимости изменять код сервера. Вы также можете изменить технологический стек на клиенте и сервере независимо, пока между ними существует интерфейс.

Недостатки:

- Более тяжелая первоначальная загрузка страницы из-за загрузки кода фреймворка, самого приложения и ресурсов.
- Ваш сервер должен быть сконфигурирован так, чтобы он направлял все запросы к единой точке входа, и переложил

обязанности по навигации на сторону клиента.

- Для отображения содержимого SPA полагается на JavaScript, но не все поисковые системы выполняют JavaScript во время индексации, и они могут не увидеть содержимое страницы. Это вредит поисковой оптимизации (SEO) вашего приложения. Тем не менее, в большинстве случаев, когда вы создаете приложения, SEO не является наиболее важным фактором, так как не весь контент должен индексироваться поисковыми системами. Чтобы преодолеть это, вы можете либо рендерить свое приложение на стороне сервера, либо использовать такие сервисы, как Prerender, чтобы "рендерить ваш javascript в браузере, сохранять статический HTML и передавать его поисковым роботам".

Ссылки

- https://github.com/grab/front-end-guide#single-page-apps-spas
- http://stackoverflow.com/questions/21862054/single-page-app-advantages-and-disadvantages
- http://blog.isquaredsoftware.com/presentations/2016-10-revolution-of-web-dev/
- https://medium.freecodecamp.com/heres-why-client-side-rendering-won-46a349fadb52

Насколько вы опытны в работе с промисами (promises) и/или их полифилами?

Обладаю практическими знаниями о них. Промис—это объект, который может вернуть одно значение в будущем: либо выполненное значение, либо причина, по которой оно не было выполнено (например, произошла ошибка сети). Промис может находиться в одном из 3 возможных состояний: выполнено, отклонено или ожидает выполнения. При использовании промисов можно добавлять callback-функции для обработки выполненного значения или причины отказа.

Некоторыми распространенными полифилами являются `$.deferred`, Q и Bluebird, но не все они соответствуют спецификации. ES2015 поддерживает промисы "из коробки", и в настоящее время полифилы обычно не нужны.

Ссылки

- https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-promise-27fc71e77261

Какие преимущества и недостатки при использовании промисов вместо колбэков (callbacks)?

Преимущества

- Помогает избежать “callback hell”, который может быть нечитаемым.
- Упрощает написание последовательного удобочитаемого асинхронного кода с помощью `.then()`.
- Упрощает написание параллельного асинхронного кода с помощью `Promise.all()`.
- С использованием промисов можно избежать следующих проблем, которые возникают при использовании callback-функций:
- *Колбэк-функция была вызвана слишком рано*
- *Колбэк-функция была вызвана слишком поздно (или вовсе не была вызвана)*
- *Колбэк-функция была вызвана слишком мало или слишком много раз*
- *Не удалось передать необходимую среду/параметры*
- *Были пропущены ошибки/исключения*

Недостатки

- Чуть более сложный код (спорно).
- В старых браузерах, где не поддерживается ES2015, вам нужно загрузить полифил, чтобы их использовать.

Ссылки

- https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch3.md

Каковы преимущества и недостатки написания JavaScript-кода на языке, который компилируется в JavaScript?

Вот несколько языков, которые компилируются в JavaScript: CoffeeScript, Elm, ClojureScript, PureScript и TypeScript.

Преимущества:

- Исправляют некоторые давние проблемы в JavaScript и препятствует использованию анти-паттернов в JavaScript.
- Позволяют писать более короткий код, предоставляя синтаксический сахар поверх JavaScript, которого, как мне кажется, не хватало в ES5, но с приходом ES2015 все изменилось.
- Статическая типизация идеальна (в случае TypeScript) для больших проектов, которые необходимо поддерживать с течением времени.

Недостатки:

- Необходима сборка/компиляция кода, так как браузеры запускают только JavaScript, и ваш код должен быть скомпилирован в JavaScript перед тем, как он будет передан в браузеры.
- Отладка может быть трудной, если карты кода (source maps) плохо сопоставляются с исходным кодом.
- Большинство разработчиков не знакомы с этими языками и должны будут изучить их. Если ваша команда будет использовать их для своих проектов, это приведет к увеличению затрат.
- Меньшее сообщество (зависит от языка), что означает, что будет труднее найти ресурсы, учебные пособия, библиотеки и инструменты.
- Может отсутствовать поддержка в IDE/редакторе.
- Эти языки всегда будут позади последнего стандарта JavaScript.
- Разработчики должны знать, во что компилируется их код— потому что это то, что будет запускаться в браузере, и это наиболее важно.

По большому счету, ES2015 значительно улучшил JavaScript и сделал разработку на нем намного удобнее. Я не вижу причин использовать CoffeeScript в наши дни.

Ссылки

- https://softwareengineering.stackexchange.com/questions/72569/what-are-the-pros-and-cons-of-coffeescript

Какие инструменты и методы вы используете при отладке кода?

React и Redux:

- React Devtools
- Redux Devtools

Vue:

- Vue Devtools

JavaScript:

- Chrome Devtools
- Выражение `debugger`
- Отладка при помощи старого доброго `console.log`

Ссылки

- https://hackernoon.com/twelve-fancy-chrome-devtools-tips-dc1e39d10d9d
- https://raygun.com/blog/javascript-debugging/

Какие языковые конструкции вы используете для итерации по свойствам объекта и элементам массива?

Для объектов:

- `for-in` циклы— `for (var property in obj) { console.log(property); }` . Тем не менее, он также будет перебирать его унаследованные свойства, и вам нужно добавить проверку `obj.hasOwnProperty(property)` перед его использованием.

- `Object.keys()` — `Object.keys(obj).forEach(function (property) { … })`. `Object.keys()` - это статический метод, который возвращает все перечисляемые свойства объекта.
- `Object.getOwnPropertyNames()` — `Object.getOwnPropertyNames(obj).forEach(function (property) { … })`. `Object.getOwnPropertyNames()` —это статический метод, который возвращает все перечисляемые и неперечисляемые свойства объекта.

Для массивов:

- Циклы `for` — `for (var i = 0; i < arr.length; i++)`. Распространенной ошибкой здесь является то, что `var` находится в области видимости функции, а не в блочной области видимости, и в большинстве случаев нам нужна переменная-итератор блочной области. ES2015 позволяет использовать `let`, который имеет блочную область видимости, и рекомендуется использовать его вместо `var`. В итоге: `for (let i = 0; i < arr.length; i++)`.
- `forEach` — `arr.forEach(function (el, index) { … })`. Эта конструкция иногда может быть более удобной, потому что вам не нужно использовать `inde` `, если все, что вам нужно, это элементы массива. Существуют также методы `every` и `some`, которые позволят вам досрочно завершить итерацию.
- `for-of` циклы— `for (let elem of arr) { … }`. ES6 представил новый цикл for-of, который позволяет перебирать объекты, которые соответствуют итерируемому протоколу такие как `String`, `Array`, `Map`, `Set`, и т.д. Он сочетает в себе преимущества цикла `for` и метода `forEach()`. Преимущество цикла `for` заключается в том, что его можно преждевременно завершить, а преимущество `forEach()` заключается в том, что он более лаконичен, чем цикл `for`, поскольку вам не нужна переменная счетчика. С циклом `for-of` вы получаете возможность выхода из цикла и более сжатый синтаксис.

В большинстве случаев я бы предпочел метод `.forEach`, но он зависит от того, что вы пытаетесь сделать. До ES6 мы использовали циклы `for`, если нам нужно было преждевременно завершить цикл при помощи `break`. Но теперь с ES6 мы можем сделать это с помощью циклов `for-of`. Я использую циклы `for`, когда мне нужно еще больше гибкости,

например, в случае увеличения итератора более одного раза за цикл.

Кроме того, при использовании цикла `for-of`, если вам нужен доступ как к индексу, так и к значению каждого элемента массива, вы можете сделать это с помощью метода ES6 `entries()` и деструктуризации:

```
const arr = ['a', 'b', 'c'];

for (let [index, elem] of arr.entries()) {

  console.log(index, ': ', elem);

}
```

Ссылки

- http://2ality.com/2015/08/getting-started-es6.html#from-for-to-foreach-to-for-of
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/entries

Объясните разницу между синхронными и асинхронными функциями

Синхронные функции являются блокирующими, а асинхронные —нет. В синхронных функциях одна операция должна завершиться, прежде чем будет запущена следующая операция. В этом случае скрипт выполняется строго по порядку операций, и выполнение скрипта приостанавливается, если одна из операций занимает очень много времени.

Асинхронные функции обычно принимают callback-функцию в качестве параметра, и выполнение продолжается на следующей строке сразу после вызова асинхронной функции. Callback-функция вызывается только тогда, когда асинхронная операция завершена и стек вызовов пуст. Ресурсоемкие операции, такие как загрузка данных с веб-сервера или запросы к базе данных, должны выполняться асинхронно, чтобы основной поток мог продолжать выполнять другие операции вместо блокировки до

завершения этой долгой операции (в случае браузеров пользовательский интерфейс будет зависать).

Что такое цикл событий (event loop)? В чем разница между стеком вызовов (call stack) и очередью событий (task queue)?

Цикл событий—это однопоточный цикл, который контролирует стек вызовов и проверяет, есть ли какая-либо работа, которую необходимо выполнить в очереди задач. Если стек вызовов пуст и в очереди задач есть callback-функции, то функция удаляется из очереди и помещается в стек вызовов для выполнения.

Рекомендую ознакомиться с докладом о цикле событий от Philip Robert. Это одно из самых популярных видео о JavaScript.

Ссылки

- https://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html
- http://theproactiveprogrammer.com/javascript/the-javascript-event-loop-a-stack-and-a-queue/

Объясните разницу при использовании `foo` в `function foo() {}` и `var foo = function() {}`

Первое—объявление функции, а второе—функциональное выражение. Ключевое отличие состоит в том, что тело функции при объявлении поднимается наверх, а тело функциональных выражений—нет (они имеют такое же поведение поднятия, что и переменные). Для получения более подробной информации, обратитесь к вопросу выше о поднятии. Если вы попытаетесь вызвать выражение функции до того, как оно будет определено, вы получите ошибку `Uncaught TypeError: XXX is not a function`.

Объявление функции

```
foo(); // 'FOOOOO'

function foo() {

  console.log('FOOOOO');

}
```

Функциональное выражение

```
foo(); // Uncaught TypeError: foo is not a function

var foo = function() {

  console.log('FOOOOO');

};
```

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function

В чем различие между переменными, созданными при помощи `let`, `var` и `const`?

Переменные, объявленные при помощи ключевого слова `var`, относятся к области видимости функции, в которой они созданы. Или, если они созданы вне какой-либо функции—к глобальному объекту. `let` и `const` относятся к блочной области видимости —это означает, что они доступны только в пределах ближайшего набора фигурных скобок (функция, блок if-else или цикл for).

```
function foo() {

  // Все переменные доступны внутри функции.

  var bar = 'bar';

  let baz = 'baz';

  const qux = 'qux';

  console.log(bar); // bar

  console.log(baz); // baz

  console.log(qux); // qux

}

console.log(bar); // ReferenceError: bar is not defined
```

```
console.log(baz); // ReferenceError: baz is not defined

console.log(qux); // ReferenceError: qux is not defined

if (true) {

  var bar = 'bar';

  let baz = 'baz';

  const qux = 'qux';

}

// переменные, объявленные при помощи var, доступны в любом
месте функции.

console.log(bar); // bar

// переменные, объявленные при помощи let и const не
доступны вне блока, в котором были определены.

console.log(baz); // ReferenceError: baz is not defined

console.log(qux); // ReferenceError: qux is not defined
```

`var` позволяет поднимать переменные, что означает, что на них можно ссылаться в коде до их объявления. `let` и `const` не позволяют этого, и выдают ошибку.

```
console.log(foo); // undefined

var foo = 'foo';

console.log(baz); // ReferenceError: can’t access lexical
declaration 'baz' before initialization

let baz = 'baz';

console.log(bar); // ReferenceError: can’t access lexical
declaration 'bar' before initialization

const bar = 'bar';
```

Переопределение переменной с помощью `var` не вызовет ошибку, в отличие от `let` и `const`.

```
var foo = 'foo';

var foo = 'bar';

console.log(foo); // “bar”

let baz = 'baz';

let baz = 'qux'; // Uncaught SyntaxError: Identifier ‘baz’
has already been declared
```

`let` отличается от `const` тем, что изменять значение `const` нельзя.

```
// Это нормально.

let foo = 'foo';

foo = 'bar';

// Это вызывает исключение.

const baz = 'baz';

baz = 'qux';
```

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

В чем разница между классом в ES6 и функцией-конструктором в ES5?

Для начала посмотрим на примеры:

```
// ES5 функция-конструктор
function Person(name) {
  this.name = name;
}
// ES6 класс
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

Они выглядят довольно похоже, если рассматривать простые конструкторы.

Основное отличие в конструкторе возникает при использовании наследования. Если мы хотим создать класс `Student` (который будет являться подклассом класса `Person`) и добавить поле `studentId`, то, в дополнение к вышеописанному, мы должны сделать следующее:

```
// ES5 функция-конструктор
function Student(name, studentId) {
  // Вызов конструктора суперкласса для инициализации производных от суперкласса членов.
  Person.call(this, name);
  // Инициализация собственных членов подкласса.
  this.studentId = studentId;
```

```
}

Student.prototype = Object.create(Person.prototype);

Student.prototype.constructor = Student;

// ES6 класс

class Student extends Person {

  constructor(name, studentId) {

    super(name);

    this.studentId = studentId;

  }

}
```

Наследование в синтаксисе ES5 является намного более многословным, а в ES6 более понятное и усваиваемое.

Ссылки

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Inheritance
- https://eli.thegreenplace.net/2013/10/22/classical-inheritance-in-javascript-es5

Можете ли вы привести пример использования стрелочных функции =>? Чем они отличаются от других функций?

Одним очевидным преимуществом стрелочных функций является упрощение синтаксиса, необходимого для создания функций, без необходимости использования ключевого слова `function`. `This` внутри стрелочных функций также привязано к замыкающей области видимости, в отличие от обычных функций, где `this` определяется контекстом, в котором они вызываются. Лексически привязанное `this` полезно при вызове callback-функций, особенно в компонентах React.

В чем преимущество использования стрелочных функций для метода в

конструкторе?

Основным преимуществом использования стрелочной функции в качестве метода внутри конструктора является то, что значение `this` устанавливается во время создания функции и не может измениться после этого. Таким образом, когда конструктор используется для создания нового объекта, `this` всегда будет ссылаться на этот объект. Например, допустим, у нас есть конструктор Person, который принимает имя в качестве аргумента, имеет два метода для вывода в консоль этого имени, один в качестве обычной функции, а другой в качестве стрелочной:

```
const Person = function(firstName) {
  this.firstName = firstName;
  this.sayName1 = function() { console.log(this.firstName);
};
  this.sayName2 = () => { console.log(this.firstName); };
};
const john = new Person('John');
const dave = new Person('Dave');
john.sayName1(); // John
john.sayName2(); // John
// У обычной функции значение `this` может быть изменено, но
у стрелочной функции нет
john.sayName1.call(dave); // Dave (потому что 'this' сейчас
ссылается на объект dave)
john.sayName2.call(dave); // John
john.sayName1.apply(dave); // Dave (потому что 'this' сейчас
ссылается на объект dave)
john.sayName2.apply(dave); // John
john.sayName1.bind(dave)(); // Dave (потому что 'this'
сейчас ссылается на объект dave)
```

```
john.sayName2.bind(dave)(); // John

var sayNameFromWindow1 = john.sayName1;

sayNameFromWindow1(); // undefined (потому что 'this' сейчас
ссылается на объект window)

var sayNameFromWindow2 = john.sayName2;

sayNameFromWindow2(); // John
```

Смысл заключается в том, что `this` можно изменить для обычной функции, но для стрелочных функций контекст всегда остается неизменным. Поэтому, даже если вы передаете стрелочную функцию в разные части вашего приложения, вам не нужно беспокоиться об изменении контекста.

Это может быть особенно полезно в классовых React-компонентах. Если вы определяете метод класса для чего-то такого, как обработчик клика, используя обычную функцию, а затем передаете этот обработчик в дочерний компонент в качестве `prop` , вам также необходимо привязать `this` в конструкторе родительского компонента , Если вместо этого вы используете стрелочную функцию, то нет необходимости привязывать `this` , так как метод автоматически получит свое значение `this` из замыкающего лексического контекста. (Прочитайте эту статью о стрелочных функциях: https://medium.com/@machnicki/handle-events-in-react-with-arrow-functions-ede88184bbb)

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
- https://medium.com/@machnicki/handle-events-in-react-with-arrow-functions-ede88184bbb

Дайте определение функции высшего порядка

Функция высшего порядка—это любая функция, которая принимает одну или несколько функций в качестве аргументов, которые она использует для работы с данными и/или возвращает функцию в качестве результата. Функции высшего порядка предназначены для абстрагирования некоторой

операции, которая выполняется повторно. Классическим примером является метод `map`, который принимает массив и функцию в качестве аргументов. Затем `map` использует эту функцию для преобразования каждого элемента в массиве, возвращая новый массив с преобразованными данными. Другими популярными примерами в JavaScript являются `forEach`, `filter` и `reduce`. Функции высшего порядка используют не только для манипуляций с массивами, но также и для возврата функции из другой функции, например при использовании `Function.prototype.bind`.

Map

Допустим, у нас есть массив с именами, которые нам нужны о преобразовать в верхний регистр.

```
const names = ['irish', 'daisy', 'anna'];
```

Императивное решение будет выглядеть так:

```
const transformNamesToUppercase = function(names) {
  const results = [];
  for (let i = 0; i < names.length; i++) {
    results.push(names[i].toUpperCase());
  }
  return results;
};
transformNamesToUppercase(names); // ['IRISH', 'DAISY',
'ANNA']
```

Воспользуемся `.map(transformerFn)`, чтобы сделать код декларативным и более коротким:

```
const transformNamesToUppercase = function(names) {

  return names.map(name => name.toUpperCase());

};

transformNamesToUppercase(names); // ['IRISH', 'DAISY',
'ANNA']
```

Ссылки

- https://medium.com/javascript-scene/higher-order-functions-composing-software-5365cf2cbe99
- https://hackernoon.com/effective-functional-javascript-first-class-and-higher-order-functions-713fde8df50a
- https://eloquentjavascript.net/05_higher_order.html

Можете ли вы привести пример деструктуризации объекта или массива?

Деструктуризация—это выражение, доступное в ES6, которое предоставляет краткий и удобный способ извлекать значения из объектов или массивов и помещать их в отдельные переменные.

Деструктуризация массива

```
// Присваивание переменной

const foo = [‘one’, ‘two’, ‘three’];

const [one, two, three] = foo;

console.log(one); // “one”

console.log(two); // “two”

console.log(three); // “three”

// Перестановка переменных местами

let a = 1;

let b = 3;
```

```
[a, b] = [b, a];

console.log(a); // 3

console.log(b); // 1
```

Деструктуризация объекта

```
// Присваивание переменной

const o = { p: 42, q: true };

const { p, q } = o;

console.log(p); // 42

console.log(q); // true
```

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- https://ponyfoo.com/articles/es6-destructuring-in-depth

Шаблонные строки в ES6 намного упрощают создание строк, можете ли вы привести пример их использования?

Шаблонные строки помогают упростить строковую строк или включение переменных в строку. До ES2015 писали так:

```
var person = { name: 'Tyler', age: 28 };

console.log('Hi, my name is ' + person.name + ' and I am ' +
person.age + ' years old!');

// 'Hi, my name is Tyler and I am 28 years old!'
```

С приходом шаблонных строк в ES6 стало намного проще:

```
const person = { name: 'Tyler', age: 28 };

console.log(`Hi, my name is ${person.name} and I am
${person.age} years old!`);

// 'Hi, my name is Tyler and I am 28 years old!'
```

Обратите внимание, что для шаблонных строк используются обратные кавычки, а не простые. Переменные добавляются в подстановки `${}`, обозначаемые знаком доллара и фигурными скобками.

Второй пример использования заключается в создании многострочных литералов. До ES2015 перенос осуществлялся следующим образом:

```
console.log('This is line one.\nThis is line two.');

// This is line one.

// This is line two.
```

Или же, чтобы не приходилось прокручивать длинную строку в текстовом редакторе, можно было разбить код на несколько строк в коде, таким образом:

```
console.log('This is line one.\n' +

'This is line two.');

// This is line one.

// This is line two.
```

Однако шаблонные строки сохраняют любой интервал, который вы добавляете к ним. Например, чтобы создать тот же многострочный литерал, который мы создали выше, вы можете просто написать:

```
console.log(`This is line one.

This is line two.`);

// This is line one.

// This is line two.
```

Еще одним вариантом использования шаблонных строк будет использование в качестве замены библиотек шаблонизации для интерполяции переменных:

```
const person = { name: 'Tyler', age: 28 };

document.body.innerHTML = `

  <div>

    <p>Name: ${person.name}</p>

    <p>Name: ${person.age}</p>

  </div>
`
```

Обратите внимание, что ваш код может быть восприимчив к XSS при использовании `.innerHTML`. Очищайте ваши данные перед отображением, если они получены от пользователя!

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

Можете ли вы привести пример каррированной функции (curry function) и в чем их преимущество?

Каррирование—это паттерн, где функция с более чем одним параметром разбивается на несколько функций, которые при последовательном вызове будут накапливать все необходимые параметры по одному. Этот метод может быть полезен для

облегчения чтения и написания кода, написанного в функциональном стиле. Важно отметить, что каррированная функция должна начинаться как одна функция, а затем разбиваться на последовательность функций, каждая из которых принимает один параметр.

```
function curry(fn) {
  if (fn.length === 0) {
    return fn;
  }
  function _curried(depth, args) {
    return function(newArgument) {
      if (depth - 1 === 0) {
        return fn(…args, newArgument);
      }
      return _curried(depth - 1, […args, newArgument]);
    };
  }
  return _curried(fn.length, []);
}
function add(a, b) {
  return a + b;
}
var curriedAdd = curry(add);
var addFive = curriedAdd(5);
var result = [0, 1, 2, 3, 4, 5].map(addFive); // [5, 6, 7,
8, 9, 10]
```

Ссылки

- https://hackernoon.com/currying-in-js-d9ddc64f162e

В чем преимущества использования spread оператора и чем он отличается от rest оператора?

Spread оператор синтаксиса ES6 очень полезен при написании кода в функциональном стиле, поскольку мы можем легко создавать копии массивов или объектов, не прибегая к `Object.create`, `slice` или функции библиотеки. Эта языковая функция часто используется в проектах с Redux и rx.js.

```
function putDookieInAnyArray(arr) {
  return […arr, 'dookie'];
}
const result = putDookieInAnyArray(['I', 'really', "don't", 'like']); // ["I", "really", "don't", "like", "dookie"]
const person = {
  name: 'Todd',
  age: 29,
};
const copyOfTodd = { …person };
```

В свою очередь, rest оператор синтаксиса ES6 позволяет в сокращенном виде указывать неопределенное количество аргументов, передаваемых в функцию. Можно сказать, что он противоположен spread оператору: собирает данные и добавляет их в массив, вместо разделения массива данных. Он используется в аргументах функций, а также при деструктуризации массивов и объектов.

```
function addFiveToABunchOfNumbers(…numbers) {
  return numbers.map(x => x + 5);
```

```
}

const result = addFiveToABunchOfNumbers(4, 5, 6, 7, 8, 9,
10); // [9, 10, 11, 12, 13, 14, 15]

const [a, b, …rest] = [1, 2, 3, 4]; // a: 1, b: 2, rest: [3,
4]

const { e, f, …others } = {

  e: 1,

  f: 2,

  g: 3,

  h: 4,

}; // e: 1, f: 2, others: { g: 3, h: 4 }
```

Ссылки

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Каким образом можно обмениваться кодом между файлами?

Это зависит от среды выполнения JavaScript.

На клиенте (в среде браузера), пока переменные/функции объявлены в глобальной области видимости (`window`), все скрипты могут на них ссылаться. В качестве альтернативы, используйте Asynchronous Module Definition (AMD) через RequireJS для модульного подхода.

На сервере (Node.js) обычно используется CommonJS. Каждый файл считается модулем, и он может экспортировать переменные и функции, добавляя их к объекту `module.exports` .

ES2015 позволяет использовать модульный синтаксис, который призван заменить как AMD, так и CommonJS. В конечном итоге он будет поддерживаться как в браузере, так и в Node.

Ссылки

- http://requirejs.org/docs/whyamd.html
- https://nodejs.org/docs/latest/api/modules.html
- http://2ality.com/2014/09/es6-modules-final.html

Для чего используются статические члены класса?

Члены статических классов (свойства/методы) не привязаны к конкретному экземпляру класса и имеют одинаковое значение вне зависимости от того, какой экземпляр ссылается на них. Статические свойства обычно являются конфигурационными переменными, а статические методы обычно являются чисто служебными функциями, которые не зависят от состояния экземпляра.

Ссылки

- https://stackoverflow.com/questions/21155438/when-to-use-static-variables-methods-and-when-to-use-instance-variables-methods