# CoreJS Interview #2

JavaScript 7

Objects Built-in methods 7

Know static Object methods 7

Property flags & descriptors (student is able to set property via Object. defineProperty) 7

Know how to create iterable objects, Symbol.iterator usage (optional) 9

ECMAScript Data Types & Expressions 11

Object computed props 11

Be able to loop through Object keys 12

Functional Scope 12

Know global scope and functional scope 12

Know variables visibility areas 13

Understand nested scopes and able work with them 13

Functions Parameters / Arguments 14

Know how to define Function parameters 14

Know difference between parameters passing by value and by reference 14

Know how to handle dynamic amount of Function parameters 15

Closures Advanced 16

Context (lexical environment) 16

Understand function creation context (lexical environment) 17

Be able to explain difference between scope and context 19

Inner/outer lexical environment 22

Understand lexical environment traversing mechanism 23

Understand connection between function and lexical environment 23

ECMAScript Intermediate 23

Function default parameters 23

Know how to use spread operator for Function arguments 24

Be able to compare arguments and rest parameters 24

Spread operator for Array 24

Understand and able to use spread operator for Array concatenation 24

Destructuring assignment 24

Be able to discover destructuring assignment concept 25

Understand variables and Function arguments destructuring assignment 25

Know how for..of loop works (optional) 25

Modules in JavaScript 27

What is module / module pattern? For what purposes they were created? 27

Modules types (AMD, ES6, CommonJS, UMD). 28

Modules syntax. 28

Common modules features (export default, named exports, exports as, etc). 34

Dynamic imports. 34

Advanced Functions 34

this in functions 34

Reference Type & losing this 36

Understand difference between function and method 37

Understand how this works, realize this possible issues 37

Manage this 37

Be able to replace this value 37

Be able to use call and apply Function built-in methods 38

Know how to bind this scope to function 38

Binding, binding one function twice 38

Functional Patterns 38

Callback (Function as argument) 38

Know callback pattern 38

Know IIFE pattern (optional) 38

Understand callback limitations (callback hell) (optional) 38

Carrying and partial functions 38

Object Oriented Programming 39

new keyword 39

Understand how new keyword works 40

Function constructor 40

Know function constructor concept 40

Able to create constructor functions 40

Public, private, static members 40

Know how to create public/static/private members 41

Understand OOP emulation patterns and conventions (optional) 41

ECMAScript Classes 42

Class declaration 42

Know class declaration syntax 42

Understand difference between class and constructor function 42

Getter/setter 42

What does super() do and where we have to use it? 43

Prototypal Inheritance Basics 44

\_\_proto\_\_ property 44

Understand \_\_proto\_\_ object property 44

Able to use [Object.create] and define \_\_proto\_\_ explicitly 44

prototype property 45

Know function prototype property 46

Understand dependency between function constructor prototype and instance \_\_proto\_\_ 46

Able to create 'class' methods using function prototype property 46

Able to set / get object prototype (optional) 46

ECMAScript Advanced Data Types & Expressions 46

Set/Map data types 46

WeakSet/WeakMap data types 48

JavaScript Errors 48

JavaScript Errors (throw, Error class) 48

try..catch statement 49

Error handling 50

Error class 50

error logging 50

async error events 50

Custom errors (optional) 51

ECMAScript Advanced 52

Promises 52

Promise states 52

Promise chaining 53

Promise static methods 53

Be able to compare promise and callback patterns (optional) 53

Be able to handle errors in promises 53

async/await 53

event loop 53

Garbage collector (concept) (optional) 53

JavaScript in Browser: 54

Global object window 54

Location 54

Know browser location structure 54

History API (Global object window) 54

Know browser History APIconcept 54

Be able to navigate within browser history 54

Be able to use history state (optional) 54

Navigator (optional) 54

Know how to parse user agent (optional) 54

Know how to discover client platform, browser 54

Cookies 54

Page Lifecycle 54

Parsing 54

Reflow 54

Repaint 54

Critical rendering path (CRP) (optional) 54

Events Basics (optional) 54

Custom events (optional) 54

Web components (optional) 54

Web components, shadow DOM (concept) (optional) 54

Network requests 55

Fetch (with usage) 55

XMLHTTPRequest (concept) (optional) 55

WebSocket (concept) (optional) 55

Timers (optional) 55

requestAnimationFrame (optional) 55

Be able to explain difference between setTimeout and requestAnimationFrame (optional) 55

Web Storage API & cookies 55

Cookies 55

Difference between localStorage, sessionStorage and cookies 55

Typescript: 56

Ability to write concise TypeScript code using its constructs 56

basic types 56

enums 56

type / interface, differences between them 56

using interfaces with optional properties, read-only properties, etc... 56

function types 56

utitily types (optional) 56

typeguards (optional) 56

creating custom types 56

generic types (concept) 56

understanding TS (ES6) module system 56

Design patterns: 57

Creational Design Patterns 57

Structural Design Patterns 57

Behavioral Design Patterns 57

MVC (optional) 57

Intermediate knowledge of patterns and best practices: 58

SOLID principles 58

design patterns used on a student's project, and able to compare these patterns (optional) 58

Software Development Methodologies (optional) 58

Agile 58

Scrum / Kanban / Waterfall 58

Estimation 58

Testing (optional) 58

Testing Types 58

Test Pyramid 58

Testing approaches (optional) 58

FIRST 58

TDD и BDD 58

Frameworks (optional) 58

Web Communication Protocols: (optional) 59

HTTP vs HTTPS 59

HTTP 1.x, 2.x, 3.x 59

HTTP methods, headers, responses, body 59

HTTP status codes groups (1xx, 2xx, 3xx, 4xx, 5xx) 59

RESTful API 59

Common web-security knowledge (optional) 60

Basic understanding of most common security terms (CORS, XSS) (optional) 60

XSS 60

CORS 60

OWASP Top 10 60

Auth (JWT, OAuth, Basic, etc.) 60

Coding tasks: 61

Function.prototype.bind implement polyfill 61

Object.create implement polyfill 61

Array.flat implement polyfill 61

Array.reduce implement polyfill 61

'hello world'.repeating(3) -> 'hello world hello world hello world'. How to implement? 61

myFunc('!', 4, -10, 34, 0) -> '4!-10!34!0`. How to implement? 61

five(plus(seven(minus(three())))) -> 9. How to implement? 62

add(5)(9)(-4)(1) -> 11. How to implement? 63

periodOutput(period) method should output in the console once per every period how mach time has passed since the first function call. Example: periodOutput(100) -> 100(after 100 ms), 200(after 100 ms), 300(after 100 ms), ... 63

extendedPeriodOutput(period) method should output in the console once per period how mach time has passed since the first function call and then increase the period. Example: // extendedPeriodOutput(100) -> 100(after 100 ms), 200(after 200 ms), 300(after 300 ms) 63

## JavaScript

### Objects Built-in methods

#### Know static Object methods

##### Метод Object.create()

Метод Object.create() позволяет создавать новые объекты и соединять их с прототипами существующих объектов.

##### Метод Object.keys()

Метод Object.keys() создает массив ключей объекта.

##### Метод Object.values()

Метод Object.values() создает массив значений объекта.

Методы Object.keys() и Object.values() позволяют возвращать данные объекта.

##### Метод Object.entries()

Метод Object.entries() создает вложенный массив пар «ключ-значение» объекта.

Метод Object.entries() возвращает только свойства экземпляра объекта, а не унаследованные свойства прототипа.

##### Метод Object.assign()

Метод Object.assign() копирует значения из одного объекта в другой.

##### Метод Object.freeze()

Метод Object.freeze() предотвращает модификацию свойств и значений объекта и добавление или удаление свойств объекта.

Метод Object.isFrozen() позволяет определить, был ли объект заморожен или нет, и возвращает логическое значение.

##### Метод Object.seal()

Метод Object.seal()предотвращает добавление новых свойств объекта, но позволяет изменять существующие свойства.

##### Метод Object.getPrototypeOf()

Метод Object.getPrototypeOf() используется для получения внутреннего скрытого [[Prototype]] объекта, также доступного через свойство \_\_proto\_\_.

Источник: https://www.8host.com/blog/metody-obektov-v-javascript/

#### Property flags & descriptors (student is able to set property via Object. defineProperty)

##### Флаги свойств

Помимо значения value, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

* writable – если true, свойство можно изменить, иначе оно только для чтения.
* enumerable – если true, свойство перечисляется в циклах, в противном случае циклы его игнорируют.
* configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Когда мы создаём свойство «обычным способом», все они имеют значение true.

Метод Object.getOwnPropertyDescriptor позволяет получить полную информацию о свойстве.

Его синтаксис:

let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);

obj - объект, из которого мы получаем информацию.

propertyName - имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

Например:

let user = {

name: "John"

};

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );

/\* дескриптор свойства:

{

"value": "John",

"writable": true,

"enumerable": true,

"configurable": true

}

\*/

Чтобы изменить флаги, мы можем использовать метод Object.defineProperty.

Его синтаксис:

Object.defineProperty(obj, propertyName, descriptor)

obj, propertyName - объект и его свойство, для которого нужно применить дескриптор.

descriptor - применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false.

let user = { };

Object.defineProperty(user, "name", {

value: "John",

// для нового свойства необходимо явно указывать все флаги, для которых значение true

enumerable: true,

configurable: true

});

alert(user.name); // John

user.name = "Pete"; // Ошибка

#### Know how to create iterable objects, Symbol.iterator usage (optional)

##### Перебираемые объекты

Перебираемые (или итерируемые) объекты – это обобщение массивов. Концепция, которая позволяет использовать любой объект в цикле for..of.

Конечно же, сами массивы являются перебираемыми объектами. Но есть и много других встроенных перебираемых объектов, например, строки.

Если объект не является массивом, но представляет собой коллекцию каких-то элементов (список, набор), то удобно использовать цикл for..of для их перебора, так что давайте посмотрим, как это сделать.

###### Symbol.iterator

Например, у нас есть объект range, который представляет собой диапазон чисел.

Чтобы сделать range итерируемым (и позволить for..of работать с ним), нам нужно добавить в объект метод с именем Symbol.iterator (специальный встроенный Symbol, созданный как раз для этого).

Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть итератор – объект с методом next.

Дальше for..of работает только с этим возвращённым объектом.

Когда for..of хочет получить следующее значение, он вызывает метод next() этого объекта.

Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что цикл завершён, в противном случае value содержит очередное значение.

Вот полная реализация range с пояснениями:

let range = {

from: 1,

to: 5

};

// 1. вызов for..of сначала вызывает эту функцию

range[Symbol.iterator] = function() {

// ...она возвращает объект итератора:

// 2. Далее, for..of работает только с этим итератором, запрашивая у него новые значения

return {

current: this.from,

last: this.to,

// 3. next() вызывается на каждой итерации цикла for..of

next() {

// 4. он должен вернуть значение в виде объекта {done:.., value :...}

if (this.current <= this.last) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

};

// теперь работает!

for (let num of range) {

alert(num); // 1, затем 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, объект 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 };

}

}

};

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next(), и он запоминает текущее состояние итерации в this.current. Короче? Да. И иногда такой способ тоже хорош.

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.

### ECMAScript Data Types & Expressions

#### Object computed props

Мы можем использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства.

Пример:

let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

alert( bag.apple ); // 5, если fruit="apple"

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit.

И если посетитель введёт слово "apple", то в объекте bag теперь будет лежать свойство {apple: 5}.

Мы можем использовать и более сложные выражения в квадратных скобках:

let fruit = 'apple';

let bag = {

[fruit + 'Computers']: 5 // bag.appleComputers = 5

};

Квадратные скобки дают намного больше возможностей, чем запись через точку. Они позволяют использовать любые имена свойств и переменные, хотя и требуют более громоздких конструкций кода.

#### Be able to loop through Object keys

Object.keys(obj) – возвращает массив ключей.

let user = {

name: "John",

age: 30

};

Object.keys(user) = ["name", "age"]

### Functional Scope

#### Know global scope and functional scope

##### Область видимости

Область видимости - это текущий контекст выполнения, в котором значения и выражения "видны" или на которые можно ссылаться. Если переменная или выражение не находятся в текущей области видимости, они будут недоступны для использования. Области видимости также могут быть выстроены в иерархию, так что дочерние области видимости имеют доступ к родительским областям видимости, но не наоборот.

В JavaScript существуют следующие виды областей видимости:

* Глобальная область видимости: Область по умолчанию для всего кода, выполняемого в режиме сценария.
* Область видимости модуля: Область видимости для кода, выполняемого в режиме модуля.
* Область видимости функции: Область видимости, создаваемая с помощью функции.

Функция создает область видимости, поэтому (например) переменная, определенная исключительно в функции, не может быть доступна извне функции или из других функций.

Однако следующий код является корректным, поскольку переменная объявлена вне функции, что делает ее глобальной:

const x = "declared outside function";

exampleFunction();

function exampleFunction() {

console.log("Inside function");

console.log(x);

}

console.log("Outside function");

console.log(x);

Блоки имеют область видимости только для деклараций let и const, но не для деклараций var.

https://developer.mozilla.org/en-US/docs/Glossary/Scope

#### Know variables visibility areas

##### Блоки кода

Если переменная объявлена внутри блока кода {...}, то она видна только внутри этого блока.

##### Вложенные функции

Функция называется «вложенной», когда она создаётся внутри другой функции.

##### Для «var» не существует блочной области видимости

Область видимости переменных var ограничивается либо функцией, либо, если переменная глобальная, то скриптом. Такие переменные доступны за пределами блока.

https://learn.javascript.ru/closure

https://learn.javascript.ru/var

#### Understand nested scopes and able work with them

Когда запускается функция, в начале ее вызова автоматически создается новое лексическое окружение для хранения локальных переменных и параметров вызова.

В процессе вызова функции у нас есть два лексических окружения: внутреннее (для вызываемой функции) и внешнее (глобальное).

Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.

Если переменная не была найдена, это будет ошибкой в строгом режиме (use strict). Без строгого режима, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким же именем.

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter = makeCounter();

В начале каждого вызова makeCounter() создается новый объект лексического окружения, в котором хранятся переменные для конкретного запуска makeCounter.

Все функции помнят лексическое окружение, в котором они были созданы. Все функции имеют скрытое свойство [[Environment]], которое хранит ссылку на лексическое окружение, в котором была создана функция.

Таким образом, counter.[[Environment]] имеет ссылку на {count: 0} лексического окружения. Так функция запоминает, где она была создана, независимо от того, где она вызывается. Ссылка на [[Environment]] устанавливается один раз и навсегда при создании функции.

Впоследствии, при вызове counter(), для этого вызова создается новое лексическое окружение, а его внешняя ссылка на лексическое окружение берется из counter.[[Environment]].

Теперь, когда код внутри counter() ищет переменную count, он сначала ищет ее в собственном лексическом окружении (пустом, так как там нет локальных переменных), а затем в лексическом окружении внешнего вызова makeCounter(), где находит count и изменяет ее.

Переменная обновляется в том лексическом окружении, в котором она существует.

Если мы вызовем counter() несколько раз, то в одном и том же месте переменная count будет увеличена до 2, 3 и т.д.

https://learn.javascript.ru/closure

### Functions Parameters / Arguments

#### Know how to define Function parameters

Мы можем передать внутрь функции любую информацию, используя параметры.

Значение, передаваемое в качестве параметра функции, также называется аргументом.

Другими словами:

Параметр – это переменная, указанная в круглых скобках в объявлении функции.

Аргумент – это значение, которое передаётся функции при её вызове.

Мы объявляем функции со списком параметров, затем вызываем их, передавая аргументы.

###### Значения по умолчанию

Если при вызове функции аргумент не был указан, то его значением становится undefined.

Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

function showMessage(from, text = "текст не добавлен") {

alert( from + ": " + text );

}

showMessage("Аня"); // Аня: текст не добавлен

https://learn.javascript.ru/function-basics

#### Know difference between parameters passing by value and by reference

We have functions in JavaScript and arguments that we feed into those functions. However, it’s not always apparent how JavaScript handles the data you’re giving in. In JavaScript, there is no such thing as “pass by reference” for any variable. All variables and arguments have a value given to them, however the value of an object’s variable is a reference. As a result, if you supply an object and alter its members inside the method, those changes will remain outside of the function. This makes it appear as if it’s a pass-by-reference system. Primitive values like integer, string, and boolean are provided by value, but objects and arrays, as previously stated, are passed by reference.

https://levelup.gitconnected.com/pass-by-value-vs-pass-by-reference-advance-javascript-series-f579ce75e7f8

#### Know how to handle dynamic amount of Function parameters

##### Остаточные параметры (...)

Остаточные параметры могут быть обозначены через три точки .... Буквально это значит: «собери оставшиеся параметры и положи их в массив».

Например, соберём все аргументы в массив args:

function sumAll(...args) { // args — имя массива

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

##### Переменная "arguments"

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами.

Хотя arguments похож на массив, и его тоже можно перебирать, это всё же не массив. Он не поддерживает методы массивов, поэтому мы не можем, например, вызвать arguments.map(...).

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

// Объект arguments можно перебирать

// for (let arg of arguments) alert(arg);

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined (второго аргумента нет)

showName("Илья");

К тому же, arguments всегда содержит все аргументы функции — мы не можем получить их часть. А остаточные параметры позволяют это сделать.

Стрелочные функции не имеют "arguments".

##### Оператор расширения

Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов.

Для Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5 (оператор "раскрывает" массив в список аргументов)

https://learn.javascript.ru/rest-parameters-spread-operator

### Closures Advanced

#### Context (lexical environment)

##### Контекст выполнения (Execution Context)

Контекст выполнения - это среда, в которой выполняется оценка текущего кода.

Так как Javascript является однопоточным (single-threaded), в любой момент времени может быть запущен только один контекст выполнения!

Каждый раз, когда запускаем функцию, мы добавляем скобки {}, а затем выполняем или запускаем её.

// глобальный контекст выполнения

function printMyName () {

// новый контекст выполнения

return `Alex`;

}

function sayMyName () {

// новый контекст выполнения

return printMyName();

}

sayMyName();

// глобальный контекст выполнения

Как только движок JavaScript увидит эти скобки {}, он создаст контекст выполнения.

Первое, что сделает движок JavaScript - создаст глобальный контекст выполнения, и тем самым даст нам две вещи: глобальный объект (global object), и ключевое слово this. В браузере глобальный объект и this являются window.

window === this // true

У нас есть доступ к этим двум вещам, потому что браузер создал для нас глобальный контекст выполнения. Это самый первый шаг, который движок JavaScript делает для нас и эти два объекта будут равны друг другу.

Если мы используем что-то вроде NodeJS, глобальный объект не будет window. Вместо этого он будет называться global.

Поэтому, как только в нашем движке JavaScript будет сделана первая фаза - фаза создания, наступит вторая - фаза исполнения, в которой мы, фактически, запустим наш код.

##### Лексическая среда (Lexical Environment)

Лексическая среда означает, место где код был написан.

Каждый раз, когда мы создаем контекст выполнения, он проверяет, где были написаны слова (функции) и их местоположение.

Если у нас функция, которая объявлена внутри другой функции printMyName, вторая функция написана в другой лексической среде - лексической среды первой функции printMyName.

// глобальная лексическая среда

function printMyName() {

// лексическая среда функции printMyName

function() {

console.log('Alex')

}

}

Лексическая среда означает место, где написан код, во время компиляции. И в зависимости от того, когда компилятор или интерпретатор увидит наш код, он будет знать разные вещи об этом коде.

Контекст выполнения говорит нам, какая лексическая среда работает в данный момент.

Лексическая среда - это место, в котором написан код. Таким образом лексическая область (lexical scope) представляет собой доступные данные и переменные, для которых была определена функция. То есть то место, где мы пишем функцию, определяет, являются ли доступные ей переменные. Так что важно где мы создаем функцию, а не где мы её вызываем.

Заключение

* лексическая среда означает, место где код был написан;
* глобальная среда является родительской средой для всех других сред, созданных в коде;
* в браузере глобальная среда называется window;
* в NodeJS глобальная среда называется global.

https://frontend-stuff.com/blog/execution-context-and-lexical-environment/

#### Understand function creation context (lexical environment)

Существует две фазы контекста выполнения JavaScript:

* Фаза создания: На этой фазе движок JavaScript создает контекст выполнения и устанавливает окружение сценария. Он определяет значения переменных и функций и устанавливает цепочку областей видимости для контекста выполнения.
* Фаза выполнения: На этой фазе движок JavaScript выполняет код в контексте выполнения. Он обрабатывает любые утверждения или выражения в сценарии и оценивает любые вызовы функций.

##### Creation Phase



Execution Context

Let's take this simple example once again:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

ar square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

At the very beginning, the JavaScript engine executes the entire source code, creates a global execution context, and then does the following things:

1. Creates a global object that is window in the browser and global in NodeJs.
2. Sets up a memory for storing variables and functions.
3. Stores the variables with values as undefined and function references.

This is called the creation phase. Here's a diagram to help explain it:



Creation Phase in Execution Context

After this creation phase, the execution context will move to the code execution phase.

<https://www.freecodecamp.org/news/how-javascript-works-behind-the-scene-javascript-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 (мы не используем его в данном примере) и прочую служебную информацию.

Один вызов функции имеет ровно один контекст выполнения, связанный с ним.

Когда функция производит вложенный вызов, происходит следующее:

* Выполнение текущей функции приостанавливается.
* Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – стеке контекстов выполнения.
* Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
* После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

https://learn.javascript.ru/recursion

#### Be able to explain difference between scope and context

##### Контекст выполнения

Контекст выполнения (execution context) — это, если говорить упрощённо, концепция, описывающая окружение, в котором производится выполнение кода на JavaScript. Код всегда выполняется внутри некоего контекста.

###### Типы контекстов выполнения

В JavaScript существует три типа контекстов выполнения:

* Глобальный контекст выполнения. Это базовый, используемый по умолчанию контекст выполнения. Если некий код находится не внутри какой-нибудь функции, значит этот код принадлежит глобальному контексту. Глобальный контекст характеризуется наличием глобального объекта, которым, в случае с браузером, является объект window, и тем, что ключевое слово this указывает на этот глобальный объект. В программе может быть лишь один глобальный контекст.
* Контекст выполнения функции. Каждый раз, когда вызывается функция, для неё создаётся новый контекст. Каждая функция имеет собственный контекст выполнения. В программе может одновременно присутствовать множество контекстов выполнения функций. При создании нового контекста выполнения функции он проходит через определённую последовательность шагов, о которой мы поговорим ниже.
* Контекст выполнения функции eval. Код, выполняемый внутри функции eval, также имеет собственный контекст выполнения. Однако функцией eval пользуются очень редко, поэтому здесь мы об этом контексте выполнения говорить не будем.

##### Стек выполнения

Стек выполнения (execution stack), который ещё называют стеком вызовов (call stack), это LIFO-стек, который используется для хранения контекстов выполнения, создаваемых в ходе работы кода.

Когда JS-движок начинает обрабатывать скрипт, движок создаёт глобальный контекст выполнения и помещает его в текущий стек. При обнаружении команды вызова функции движок создаёт новый контекст выполнения для этой функции и помещает его в верхнюю часть стека.

Движок выполняет функцию, контекст выполнения которой находится в верхней части стека. Когда работа функции завершается, её контекст извлекается из стека и управление передаётся тому контексту, который находится в предыдущем элементе стека.

Изучим эту идею с помощью следующего примера:

let a = 'Hello World!';

function first() {

console.log('Inside first function');

second();

console.log('Again inside first function');

}

function second() {

console.log('Inside second function');

}

first();

console.log('Inside Global Execution Context');

Когда вышеприведённый код загружается в браузер, JavaScript-движок создаёт глобальный контекст выполнения и помещает его в текущий стек вызовов. При выполнении вызова функции first() движок создаёт для этой функции новый контекст и помещает его в верхнюю часть стека.

При вызове функции second() из функции first() для этой функции создаётся новый контекст выполнения и так же помещается в стек. После того, как функция second() завершает работу, её контекст извлекается из стека и управление передаётся контексту выполнения, находящемуся в стеке под ним, то есть, контексту функции first().

Когда функция first() завершает работу, её контекст извлекается из стека и управление передаётся глобальному контексту. После того, как весь код оказывается выполненным, движок извлекает глобальный контекст выполнения из текущего стека.

###### О создании контекстов и о выполнении кода

До сих пор мы говорили о том, как JS-движок управляет контекстами выполнения. Теперь поговорим о том, как контексты выполнения создаются, и о том, что с ними происходит после создания. В частности, речь идёт о стадии создания контекста выполнения и о стадии выполнения кода.

Стадия создания контекста выполнения

Перед выполнением JavaScript-кода создаётся контекст выполнения. В процессе его создания выполняются три действия:

1. Определяется значение this и осуществляется привязка this (this binding).
2. Создаётся компонент LexicalEnvironment (лексическое окружение).
3. Создаётся компонент VariableEnvironment (окружение переменных).

Привязка this

В глобальном контексте выполнения this содержит ссылку на глобальный объект (как уже было сказано, в браузере это объект window).

В контексте выполнения функции значение this зависит от того, как именно была вызвана функция. Если она вызвана в виде метода объекта, тогда значение this привязано к этому объекту. В других случаях this привязывается к глобальному объекту или устанавливается в undefined (в строгом режиме).

Лексическое окружение

В соответствии со спецификацией ES6, лексическое окружение (Lexical Environment) — это термин, который используется для определения связи между идентификаторами и отдельными переменными и функциями на основе структуры лексической вложенности ECMAScript-кода. Лексическое окружение состоит из записи окружения (Environment Record) и ссылки на внешнее лексическое окружение, которая может принимать значение null.

Проще говоря, лексическое окружение — это структура, которая хранит сведения о соответствии идентификаторов и переменных. Под «идентификатором» здесь понимается имя переменной или функции, а под «переменной» — ссылка на конкретный объект (в том числе — на функцию) или примитивное значение.

В лексическом окружении имеется два компонента:

1. Запись окружения. Это место, где хранятся объявления переменных и функций.
2. Ссылка на внешнее окружение. Наличие такой ссылки говорит о том, что у лексического окружения есть доступ к родительскому лексическому окружению (области видимости).

Существует два типа лексических окружений:

1. Глобальное окружение (или глобальный контекст выполнения) — это лексическое окружение, у которого нет внешнего окружения. Ссылка глобального окружения на внешнее окружение представлена значением null. В глобальном окружении (в записи окружения) доступны встроенные сущности языка (такие, как Object, Array, и так далее), которые связаны с глобальным объектом, там же находятся и глобальные переменные, определённые пользователем. Значение this в этом окружении указывает на глобальный объект.
2. Окружение функции, в котором, в записи окружения, хранятся переменные, объявленные пользователем. Ссылка на внешнее окружение может указывать как на глобальный объект, так и на внешнюю по отношении к рассматриваемой функции функцию.

Существует два типа записей окружения:

1. Декларативная запись окружения, которая хранит переменные, функции и параметры.
2. Объектная запись окружения, которая используется для хранения сведений о переменных и функциях в глобальном контексте.

В результате в глобальном окружении запись окружения представлена объектной записью окружения, а в окружении функции — декларативной записью окружения.

Обратите внимание на то, что в окружении функции декларативная запись окружения, кроме того, содержит объект arguments, который хранит соответствия между индексами и значениями аргументов, переданных функции, и сведения о количестве таких аргументов.

Окружение переменных

Окружение переменных (Variable Environment) — это тоже лексическое окружение, запись окружения которого хранит привязки, созданные посредством команд объявления переменных (VariableStatement) в текущем контексте выполнения.

Так как окружение переменных также является лексическим окружением, оно обладает всеми вышеописанными свойствами лексического окружения.

В ES6 существует одно различие между компонентами LexicalEnvironment и VariableEnvironment. Оно заключается в том, что первое используется для хранения объявлений функций и переменных, объявленных с помощью ключевых слов let и const, а второе — только для хранения привязок переменных, объявленных с использованием ключевого слова var.

Стадия выполнения кода

На этой стадии выполняется присвоение значений переменным и осуществляется выполнение кода.

Обратите внимание на то, что если в процессе выполнения кода JS-движок не сможет найти в месте объявления значение переменной, объявленной с помощью ключевого слова let, он присвоит этой переменной значение undefined.

https://habr.com/ru/companies/ruvds/articles/422089/

#### Inner/outer lexical environment

Когда запускается функция, в начале ее вызова автоматически создается новое лексическое окружение для хранения локальных переменных и параметров вызова.

В процессе вызова функции у нас есть два лексических окружения: внутреннее (для вызываемой функции) и внешнее (глобальное).

У внутреннего лексического окружения есть ссылка на внешнее outer.

Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.

Если переменная не была найдена, это будет ошибкой в строгом режиме (use strict). Без строгого режима, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким же именем.

#### Understand lexical environment traversing mechanism

#### Understand connection between function and lexical environment

### ECMAScript Intermediate

#### Function default parameters

Если при вызове функции аргумент не был указан, то его значением становится undefined.

Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

function showMessage(from, text = "текст не добавлен") {

alert( from + ": " + text );

}

showMessage("Аня"); // Аня: текст не добавлен

Теперь, если параметр text не указан, его значением будет "текст не добавлен".

Иногда имеет смысл присваивать значения по умолчанию для параметров не в объявлении функции, а на более позднем этапе.

function showMessage(text) {

// ...

if (text === undefined) { // если параметр отсутствует

text = 'пустое сообщение';

}

alert(text);

}

showMessage(); // пустое сообщение

#### Know how to use spread operator for Function arguments

function sumAll(...args) { // args — имя массива

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

alert( sumAll(1, 2) ); // 3

alert( sumAll(1, 2, 3) ); // 6

https://learn.javascript.ru/rest-parameters-spread-operator

#### Be able to compare arguments and rest parameters

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Юлий Цезарь

// Оставшиеся параметры пойдут в массив

// titles = ["Консул", "Император"]

alert( titles[0] ); // Консул

alert( titles[1] ); // Император

alert( titles.length ); // 2

}

showName("Юлий", "Цезарь", "Консул", "Император");

#### Spread operator for Array

#### Understand and able to use spread operator for Array concatenation

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(...arr1, ...arr2) ); // 8

#### Destructuring assignment

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

// у нас есть массив с именем и фамилией

let arr = ["Ilya", "Kantor"];

// деструктурирующее присваивание

// записывает firstName = arr[0]

// и surname = arr[1]

let [firstName, surname] = arr;

alert(firstName); // Ilya

alert(surname); // Kantor

Деструктурирующее присваивание также работает с объектами.

let options = {

title: "Menu",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

https://learn.javascript.ru/destructuring-assignment

#### Be able to discover destructuring assignment concept

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

let a, b, rest;

[a, b] = [10, 20];

console.log(a);

// Expected output: 10

console.log(b);

// Expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);

// Expected output: Array [30, 40, 50]

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\_assignment

#### Understand variables and Function arguments destructuring assignment

#### Know how for..of loop works (optional)

Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно. А также это короче.

Например, объект range, который представляет собой диапазон чисел:

let range = {

from: 1,

to: 5

};

// Мы хотим, чтобы работал for..of:

// for(let num of range) ... num=1,2,3,4,5

Чтобы сделать range итерируемым (и позволить for..of работать с ним), нам нужно добавить в объект метод с именем Symbol.iterator (специальный встроенный Symbol, созданный как раз для этого).

1. Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть итератор – объект с методом next.
2. Дальше for..of работает только с этим возвращённым объектом.
3. Когда for..of хочет получить следующее значение, он вызывает метод next() этого объекта.
4. Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что цикл завершён, в противном случае value содержит очередное значение.

Вот полная реализация range с пояснениями:

let range = {

from: 1,

to: 5

};

// 1. вызов for..of сначала вызывает эту функцию

range[Symbol.iterator] = function() {

// ...она возвращает объект итератора:

// 2. Далее, for..of работает только с этим итератором, запрашивая у него новые значения

return {

current: this.from,

last: this.to,

// 3. next() вызывается на каждой итерации цикла for..of

next() {

// 4. он должен вернуть значение в виде объекта {done:.., value :...}

if (this.current <= this.last) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

};

// теперь работает!

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

https://learn.javascript.ru/array#perebor-elementov

### Modules in JavaScript

#### What is module / module pattern? For what purposes they were created?

Сначала программы на JavaScript были небольшими — в прежние времена они использовались для изолированных задач, добавляя при необходимости немного интерактивности веб-страницам, так что большие скрипты в основном не требовались. Прошло несколько лет, и вот мы уже видим полномасштабные приложения, работающие в браузерах и содержащие массу кода на JavaScript; кроме того, язык стал использоваться и в других контекстах (например, Node.js).

Таким образом, в последние годы появились причины на то, чтобы подумать о механизмах деления программ на JavaScript на отдельные модули, которые можно импортировать по мере необходимости. Node.js включал такую возможность уже давно, кроме того, некоторые библиотеки и фреймворки JavaScript разрешали использование модулей (например, CommonJS и основанные на AMD системы модулей типа RequireJS, а позднее также Webpack и Babel).

К счастью, современные браузеры стали сами поддерживать функциональность модулей, о чем и рассказывает эта статья. Этому можно только порадоваться — браузеры могут оптимизировать загрузку модулей, что было бы гораздо эффективнее использования библиотеки, и взять на себя обработку на стороне клиента и прочие накладные расходы.

Встроенная обработка модулей JavaScript связана с функциями import и export, которые поддерживаются браузерами.

https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Modules

#### Modules types (AMD, ES6, CommonJS, UMD).

#### Modules syntax.

В современном JavaScript осталось два основных стандарта модульных систем. Это CommonJS, которая является основной для платформы Node.js, и ESM (ECMAScript 6 модули), которая была принята как стандарт для языка и внесена в спецификацию ES2015.

##### ESM-модули

###### Именованный импорт/экспорт

В случае, когда необходимо экспортировать несколько сущностей из модуля, применяется именованный экспорт. Он выполняется с помощью инструкции export.

export можно использовать в момент объявления функции, переменной или класса:

export function counter() { /\* ... \*/ }

export const getCurrentDate = () => { /\* ... \*/ }

export const awesomeValue = 42;

export class User { /\* ... \*/ }

Для больших модулей удобнее использовать группированный экспорт, это позволяет наглядно увидеть все экспортируемые сущности внутри модуля:

function counter() { /\* ... \*/ }

const awesomeValue = 42;

export { counter, awesomeValue };

Чтобы импортировать какой-либо метод, необходимо воспользоваться инструкциeй import, указав интересующие части модуля и путь до него:

import { counter, awesomeValue } from './modulePath/index.js';

counter();

console.log('Response:', awesomeValue);

###### Импорт/Экспорт по умолчанию

В случае, когда из файла модуля экспортируется только одна сущность, удобнее использовать экспорт по умолчанию. Для этого необходимо добавить default после инструкции export:

function counter() { /\* ... \*/ }

export default counter;

Импорт модуля в случае экспорта по умолчанию:

/\*\*

Можно использовать любое имя для импортируемой переменной, в связи с тем,

что отсутствует привязка к наименованию внутри модуля

\*/

import rainbowCounter from './modulePath/index.js';

rainbowCounter();

###### Дополнительные возможности

**Переименование**. Для изменения имени метода в момент импорта/экспорта существует инструкция as:

function counter() { /\* ... \*/ }

export { counter as rainbowCounter };

Импорт этой функции будет доступен только по новому имени:

import { rainbowCounter } from './modulePath/index.js';

rainbowCounter();

Переименование в момент импорта:

import { counter as rainbowCounter } from './modulePath/index.js';

rainbowCounter();

Этот синтаксис полезен для случаев, когда имя импортируемой части уже занято. Также можно сократить имя функции/переменной/класса, если она часто используется в файле:

import { debounce } from 'shared';

import { debounce as \_debounce } from 'lodash';

import { awesomeFunctionThatYouNeed as \_helper } from 'awesome-lib';

**Инициализация модуля без импорта его частей**. Используется, когда необходимо выполнить импорт модуля для выполнения кода внутри него, но не импортировать какую-либо его часть:

import './modulePath/index.js';

**Импорт всего содержимого модуля**. Можно импортировать всё содержимое модуля в переменную и обращаться к частям модуля как к свойствам этой переменной:

import \* as customName from './modulePath/index.js';

customName.counter();

console.log('Response:', customName.awesomeValue);

Такой синтаксис не рекомендуется использовать, сборщик модулей (например, Webpack) не сможет корректно выполнить tree-shaking при таком использовании.

**Реэкспорт**. Существует сокращенный синтаксис для реэкспорта модулей. Это бывает полезно, когда нужно собрать модули из разных файлов в одном экспорте:

export { counter, awesomeValue } from './modulePath/index.js';

при таком реэкспорте наименования частей модуля будут сохраняться, но можно изменять их с помощью инструкции as:

export { counter as \_counter , awesomeValue as \_awesomeValue } from './modulePath/index.js';

Аналогичным способом можно реэкспортировать значения по умолчанию:

export { default as moduleName } from './modulePath/index.js';

**Динамические импорты**. Кроме «статических» импортов можно загружать модули ассинхронно, для этого есть специальное выражение import(). Пример использования:

import('./modulePath/index.js')

.then(moduleObject => { /\* ... \*/ })

.catch( error => { /\* ... \*/ })

Это выражение возвращает promise, который при успешном завершении возвращает объект со всеми экспортами модуля, а при исключении — ошибку выполнения импорта. В Webpack синтаксис динамических импортов используется для создания отдельных чанков.

###### Использование модулей в браузере

Современные браузеры нативно поддерживают модули. Для того, чтобы браузер понимал, что мы экспортируем не просто исполняемый JS-файл, а модуль, необходимо в тэг script, где импортируется модуль, добавить атрибут type="module".

Рассмотрим на примере небольшого проекта.

Структура проекта:

┌─index.html

├─main.js

└─dist

├─ module1.js

└─ module2.js

Файл main.js:

import { counter } from './dist/module1';

import { awesomeValue } from './dist/module2';

counter();

console.log('Response:', awesomeValue);

Импорт модуля внутри index.html:

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

По атрибуту type="module" браузер понимает, что экспортирует файл с модулями, и корректно его обработает. Стоит отметить, что пути импорта, указанные в main.js (./dist/module1 и ./dist/module2), будут преобразованы в абсолютные пути относительно текущего расположения, и браузер запросит эти файлы у сервера по адресам /dist/module1 и /dist/module2 соответственно. Практического применения у этой возможности не так много, в основном в проектах используется сборщик (например Webpack), который преобразует ESM-модули в bundle. Однако использование ESM-модулей в браузере может позволить улучшить загрузку страницы за счет разбиения bundle-файлов на маленькие части и постепенной их загрузки.

##### CommonJS

**Экспорт**. Для экспорта в CommonJS используются глобальные объекты module и exports. Для этого необходимо просто добавить новое поле в объект exports.

module.exports.counter = function () { /\* ... \*/ }

module.exports.awesomeValue = 42;

module.exports.getCurrentDate = () => {/\* ... \*/}

module.exports.User = class User { /\* ... \*/ }

Для удобства экспорта части фунциональности в глобальной области существует переменная exports, которая является ссылкой на module.exports. Поэтому возможен и такой синтаксис экспорта:

exports.counter = function () { /\* ... \*/ }

exports.awesomeValue = 42;

В CommonJS cуществует что-то схожее с импортом по умолчанию, для этого необходимо просто присвоить module.exports значению экспортируемой функции:

module.exports = function () { /\* ... \*/ }

Сохранение значения в exports напрямую, в отличие от именованного экспорта, не будет работать:

// Данная функция не будет экспортирована!!!

exports = function () { /\* ... \*/ }

Стоит обратить внимание, что если были экспортированы части модуля, они затрутся и будет экспортировано только последнее значение module.exports:

exports.counter = function () { /\* ... \*/ }

exports.awesomeValue = 42;

module.exports = {};

// counter и awesomeValue не будут экспортированы

**Импорт**. Для импорта необходимо воспользоваться конструкцией require() и указать путь до модуля:

const loadedModule = require('./modulePath/index.js');

loadedModule.counter()

console.log(loadedModule.awesomeValue);

Можно воспользоваться деструктуризацией и получить значение необходимой функции сразу после импорта:

const { counter, awesomeValue } = require('./modulePath/index.js');

counter()

console.log(awesomeValue);

##### Работа с модулями в Node.js

###### Поддержка ESM-модулей

До недавнего времени Node.js поддерживал только CommonJS, но с версии 13.2.0 команда разработчиков анонсировала поддержку ESM (с версии 8.5.0 поддержка модулей ECMAScript 6 была скрыта за экспериментальным флагом). Подробно о том, как работать с модулями ECMAScript 6 в Node.js, можно прочитать в анонсе команды разработчиков Node.js.

###### Поиск модулей

Все относительные пути, начинающиеся c './' или '../', будут обрабатываться только относительно рабочей папки проекта. Пути с '/' будут обрабатываться как абсолютные пути файловой системы. Для остальных случаев Node.js начинает поиск модулей в папке проекта node\_modules (пример: /home/work/projectN/node\_modules). В случае, если интересующий модуль не был найден, Node.js поднимается на уровень выше и продолжает свой поиск там. И так до самого верхнего уровня файловой системы. Поиск необходимой библиотеки будет выглядеть следующим образом:

/home/work/projectN/node\_modules

/home/work/node\_modules

/home/node\_modules

/node\_modules

Если в папках node\_modules не удалось обнаружить искомый модуль, то в запасе у Node.js есть еще места, которые он анализирует в поисках необходимой библиотеки. Это так называемые GLOBAL\_FOLDERS. В них добавляются пути, переданные через переменную окружения NODE\_PATH, и три дополнительных пути, которые существуют всегда:

$HOME/.node\_modules

$HOME/.node\_libraries

$PREFIX/lib/node

/\*\*

$HOME - домашняя директория пользователя,

$PREFIX - node\_prefix. Путь до установленной версии Node.js

\*/

При желании можно посмотреть все возможные директории, где Node.js ищет модули из папки проекта, обратившись к методу paths() внутри require.resolve.

###### Дополнительные свойства у module и require

У module и require есть дополнительные свойства, которые могут быть полезны.

module.id — уникальный идентификатор модуля. Обычно это полностью определенный путь до модуля.

module.children — объект, содержащий модули, которые импортированы в текущем файле. Ключами объекта являются module.id:

// Расположение исполняемого файла в файловой системе /home/work/projectN

const { counter, awesomeValue } = require('./modulePath/index.js');

console.log(module.children);

// { '/home/work/projectN/modulePath/index.js': <Module> }

require.cache — представляет из себя объект с информацией о каждом импортированном модуле. Если при импорте модуля Node.js находит его в кеше, код модуля не будет выполняться повторно, а экспортируемые сущности будут взяты из закешированного значения. При необходимости повторного «чистого» импорта модуля необходимо сбросить закешированное значение, удалив его из кеша:

delete require.cache['./modulePath/index.js'];

##### Что происходит в момент импорта ES-модуля

В момент выполнения файла Javascript-движок выполняет несколько этапов загрузки модулей:

* построение графа зависимостей;
* оценка расположения модулей и загрузка файлов;
* анализ модулей;
* запись информации о модулях и создание полей всех экспортируемых значений (без их состояний);
* выполнение сценария модулей для получение состояний;
* запись состояний экспортируемых частей модулей.

Структура данных, содержащая информацию о модуле (уникальный идентификатор, список зависимостей и состояния всех экспортируемых значений) называется Module Records.

При выполнении скрипта строится граф зависимостей и создается запись по каждому импортируемому модулю внутри него. В момент каждого импорта, вызывается метод Evaluate() внутри модуля Module Records. При первом вызове этой функции выполняется сценарий для получения и сохранения состояния модуля. Подробнее об этом процессе можно прочитать в статье «Глубокое погружение в ES-модули в картинках».

##### Что происходит при повторном импорте модуля

В предыдущей главе мы упомянули метод Evaluate(). При очередном импорте модуля Evaluate() вызывается повторно, но если импорт модуля был успешно выполнен до этого, то метод возвращает undefined и сценарий модуля запущен не будет. Поэтому запись состояния модуля происходит единожды.

Но остался открытым вопрос, создаётся ли новая сущность Module Records при повторном импорте? Например в данном случае:

import { counter } from './modulePath';

import { counter } from './modulePath';

За получение Module Records для каждого import отвечает метод HostResolveImportedModule, который принимает два аргумента:

referencingScriptOrModule — идентификатор текущего модуля, откуда происходит импорт;

specifier — идентификатор импортируемого модуля, в данном случае ./modulePath.

В спецификации говорится, что для одинаковых парах значений referencingScriptOrModule и specifier возвращается один и тот же экземпляр Module Records.

Рассмотрим еще один пример, когда один и тот же модуль импортируется в нескольких файлах:

/\*\* main.js \*/

import moduleA from './moduleA.js'

import moduleB from './moduleB.js'

/\*\* moduleB.js \*/

import moduleA from './moduleA.js

Будут ли здесь создаваться дублирующие Module Records для moduleB.js? Для этого обратимся к спецификации:

Multiple different referencingScriptOrModule, specifier pairs may map to the same Module Record instance. The actual mapping semantic is host-defined but typically a normalization process is applied to specifier as part of the mapping process. A typical normalization process would include actions such as alphabetic case folding and expansion of relative and abbreviated path specifiers

Таким образом, даже если referencingScriptOrModule отличается, а specifier одинаков, может быть возвращен одинаковый экземпляр Module Records.

Однако этой унификации не будут подвержены импорты с дополнительными параметрами в specifier:

import moduleA from './moduleA.js?q=1111'

import \_moduleA from './moduleА.js?q=1234'

console.log(moduleA !== \_moduleA) // true

##### Циклические зависимости

При большой вложенности модулей друг в друга может возникнуть циклическая зависимость:

ModuleA -> ModuleB -> ModuleC -> ModuleD -> ModuleA

Для наглядности, эту цепочку зависимостей можно упростить до:

ModuleA <-> ModuleD

ES-модули нативно умеют работать с циклическими зависимостями и корректно их обрабатывать. Принцип работы подробно описан в спецификации. Однако, ESM редко используются без обработки. Обычно с помощью транспилятор (Babel) сборщик модулей (например, Webpack) преобразует их в CommonJS для запуска на Node.js, или в исполнямый скрипт (bundle) для браузера. Циклические зависимости не всегда могут быть источником явных ошибок и исключений, но могут стать причиной некорректного поведения кода, которое трудно будет отловить.

Есть несколько хаков, как можно обходить циклические зависимости для некоторые ситуаций, но лучше просто не допускать их возниковения.

https://habr.com/ru/companies/domclick/articles/532084/

#### Common modules features (export default, named exports, exports as, etc).

#### Dynamic imports.

### Advanced Functions

#### this in functions

Имеются различия при использовании this в строгом и нестрогом режиме.

В большинстве случаев значение this определяется тем, каким образом вызвана функция. Значение this не может быть установлено путём присваивания во время исполнения кода и может иметь разное значение при каждом вызове функции. В ES5 представлен метод bind(), который используется для привязки значения ключевого слова this независимо от того, как вызвана функция. Также в ES2015 представлены стрелочные функции, которые не создают собственные привязки к this (они сохраняют значение this лексического окружения, в котором были созданы).

##### Значение

Свойство контекста выполнения кода (global, function или eval), которое в нестрогом режиме всегда является ссылкой на объект, а в строгом режиме может иметь любое значение.

В глобальном контексте выполнения (за пределами каких-либо функций) this ссылается на глобальный объект вне зависимости от режима (строгий или нестрогий).

В пределах функции значение this зависит от того, каким образом вызвана функция.

В строгом режиме, если значение this не установлено в контексте выполнения, оно остаётся undefined.

Для того, чтобы при вызове функции установить this в определённое значение, используйте call() или apply().

В нестрогом режиме, если значение, переданное в call или apply как this, не является объектом, будет сделана попытка преобразовать его в объект с помощью внутренней операции ToObject. Таким образом, если переданное значение является примитивом, таким как 7 или 'foo', оно будет преобразовано в Object с использованием связанного конструктора, так что примитивное число 7 будет преобразовано в объект так, как будто с помощью new Number(7), а строка 'foo' - как будто с помощью new String('foo').

##### Метод bind

ECMAScript 5 представил Function.prototype.bind(). Вызов f.bind(someObject) создаёт новую функцию с тем же телом и областью действия, что и f, но там, где в исходной функции используется this, в новой функции оно постоянно будет связано с первым аргументом bind, независимо от того, как функция используется.

##### Стрелочные функции

В стрелочных функциях, this привязан к окружению, в котором была создана функция. В глобальной области видимости this будет указывать на глобальный объект.

##### В методе объекта

Когда функция вызывается как метод объекта, используемое в этой функции ключевое слово this принимает значение объекта, по отношению к которому вызван метод.

##### this в цепочке object's prototype

Это же представление справедливо и для методов, определённых где-либо в цепочке object's prototype. Если метод находится в цепочке прототипов, то this ссылается на объект, на котором был вызван метод, т.е. так, словно метод является методом самого объекта, а не прототипа.

##### this с геттерами/сеттерами

Все те же утверждения справедливы, если функция вызывается из геттера или сеттера. Для функции, которая используется как геттер или сеттер this привязан к объекту, свойство которого необходимо извлечь через геттер/сеттер.

##### В конструкторе

Когда функция используется как конструктор (с ключевым словом new ), this связано с создаваемым новым объектом.

##### call и apply

Когда в теле функции используется ключевое слово this, его значение может быть привязано к конкретному объекту в вызове при помощи методов call или apply, которые наследуются всеми функциями от Function.prototype.

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this>

This — это ссылка на контекст исполнения функции. Таким образом получается, что this тесно связан именно с функциями и рассматривается относительно них. Вне функции this будет ссылаться на глобальный контекст

Для функций, объявленных через function f( ) { }, this вычисляется в момент вызова и равен объекту перед точкой. Если такого объекта нет — тогда this будет указывать на глобальный контекст (window)

Для стрелочных функций this определяется в момент их создания и больше никогда не изменяется.

https://habr.com/ru/articles/515356/

#### Reference Type & losing this

Некоторые хитрые способы вызова метода приводят к потере значения this, например:

let user = {

name: "Джон",

hi() { alert(this.name); },

bye() { alert("Пока"); }

};

user.hi(); // Джон (простой вызов метода работает хорошо)

// теперь давайте попробуем вызывать user.hi или user.bye

// в зависимости от имени пользователя user.name

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

Присмотревшись поближе, в выражении obj.method() можно заметить две операции:

Сначала оператор точка '.' возвращает свойство объекта – его метод (obj.method).

Затем скобки () вызывают этот метод (исполняется код метода).

Итак, каким же образом информация о this передаётся из первой части во вторую?

Если мы поместим эти операции в отдельные строки, то значение this, естественно, будет потеряно:

// разделим получение метода объекта и его вызов в разных строках

let hi = user.hi;

hi(); // Ошибка, потому что значением this является undefined

Здесь hi = user.hi сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет this.

Для работы вызовов типа user.hi(), JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого Reference Type.

Этот ссылочный тип (Reference Type) является внутренним типом. Мы не можем явно использовать его, но он используется внутри языка.

Значение ссылочного типа – это «триплет»: комбинация из трёх значений (base, name, strict), где:

* base – это объект.
* name – это имя свойства объекта.
* strict – это режим исполнения. Является true, если действует строгий режим (use strict).

Результатом доступа к свойству user.hi является не функция, а значение ссылочного типа. Для user.hi в строгом режиме оно будет таким:

// значение ссылочного типа (Reference Type)

(user, "hi", true)

Когда скобки () 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, используемый, чтобы передать информацию от точки . до вызывающих скобок ().

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 ней. Поэтому дальнейший вызов происходит уже без this.

Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']() (они делают то же самое). Существуют различные способы решения этой проблемы: одним из таких является func.bind().

https://learn.javascript.ru/reference-type

#### Understand difference between function and method

A function is a set of instructions or procedures to perform a specific task, and a method is a set of instructions that are associated with an object.

* A function is used to split the code into easily understandable parts, which can be reused as well.
* A function doesn’t need any object and is independent, while the method is a function, which is linked with any object.
* We can directly call the function with its name, while the method is called by the object’s name.
* Function is used to pass or return the data, while the method operates the data in a class.
* Function is an independent functionality, while the method lies under object-oriented programming.
* In functions, we don’t need to declare the class, while to use methods we need to declare the class.
* Functions can only work with the provided data, while methods can access all the data provided in the given class.

<https://www.educative.io/answers/what-is-the-difference-between-a-method-and-a-function>

Другая важная концепция в кодировании — функции — позволяют хранить фрагмент кода, который выполняет одну задачу внутри определённого блока, а затем вызывать этот код всякий раз, когда вам это нужно, используя одну короткую команду, вместо того, чтобы вводить один и тот же код несколько раз.

Разница между методом и функцией лишь в том, что методы - это функции, определённые внутри объектов. Встроенные функции (методы) браузера и переменные (так называемые свойства) хранятся внутри структурированных объектов, чтобы сделать код более эффективным и более простым в использовании.

https://developer.mozilla.org/ru/docs/Learn/JavaScript/Building\_blocks/Functions

#### Understand how this works, realize this possible issues

#### Manage this

#### Be able to replace this value

#### Be able to use call and apply Function built-in methods

#### Know how to bind this scope to function

#### Binding, binding one function twice

### Functional Patterns

#### Callback (Function as argument)

Давайте передадим функцию callback вторым аргументом в loadScript, чтобы вызвать её, когда скрипт загрузится:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

для нескольких асинхронных действий, которые нужно выполнить друг за другом, код выглядит вот так:

loadScript('1.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('2.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('3.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...и так далее, пока все скрипты не будут загружены (\*)

}

});

}

})

}

});

https://learn.javascript.ru/callbacks

#### Know callback pattern

#### Know IIFE pattern (optional)

В прошлом, поскольку существовал только var, а он не имел блочной области видимости, программисты придумали способ её эмулировать. Этот способ получил название «Immediately-invoked function expressions» (сокращенно IIFE).

(function() {

var message = "Hello";

alert(message); // Hello

})();

#### Understand callback limitations (callback hell) (optional)

#### Carrying and partial functions

Каррирование – продвинутая техника для работы с функциями. Она используется не только в JavaScript, но и в других языках.

Каррирование – это трансформация функций таким образом, чтобы они принимали аргументы не как f(a, b, c), а как f(a)(b)(c).

Каррирование не вызывает функцию. Оно просто трансформирует её.

function curry(func) {

return function curried(...args) {

if (args.length >= func.length) {

return func.apply(this, args);

} else {

return function(...args2) {

return curried.apply(this, args.concat(args2));

}

}

};

}

https://learn.javascript.ru/currying-partials

### Object Oriented Programming

#### new keyword

Базовый синтаксис выглядит так:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Использование:

let user = new User("Иван");

user.sayHi();

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами.

При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

Когда вызывается new User("Иван"):

1. Создаётся новый объект.
2. constructor запускается с заданным аргументом и сохраняет его в this.name.

https://learn.javascript.ru/class

Оператор (операторная функция) new создаёт экземпляр объекта, встроенного или определённого пользователем, имеющего конструктор.

new constructor[([arguments])]

##### Параметры

constructor - функция, задающая тип объекта.

arguments - список параметров, с которыми будет вызван конструктор.

##### Описание

Создание объекта, определённого пользователем, требует два шага:

1. Написать функцию, которая задаст тип объекта.
2. Создать экземпляр объекта, используя new.

Чтобы определить новый тип объекта, создайте функцию, которая задаст его и имя и свойства. Свойство объекта также может быть объектом.

Когда исполняется new Foo(...) , происходит следующее:

1. Создаётся новый объект, наследующий Foo.prototype.
2. Вызывается конструктор — функция Foo с указанными аргументами и this, привязанным к только что созданному объекту. new Foo эквивалентно new Foo(), то есть если аргументы не указаны, Foo вызывается без аргументов.
3. Результатом выражения new становится объект, возвращённый конструктором. Если конструктор не возвращает объект явно, используется объект из п. 1. (Обычно конструкторы не возвращают значение, но они могут делать это, если нужно переопределить обычный процесс создания объектов.)

Всегда можно добавить свойство к уже созданному объекту. Например, car1.color = "black" добавляет свойство color к объекту car1, и присваивает ему значение "black". Это не затрагивает другие объекты. Чтобы добавить свойство ко всем объектам типа, нужно добавлять его в определение типа Car.

Добавить свойство к ранее определённому типу можно используя свойство Function.prototype (en-US). Это определит свойство для всех объектов, созданных этой функцией, а не только у какого-либо экземпляра.

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new

#### Understand how new keyword works

#### Function constructor

#### Know function constructor concept

#### Able to create constructor functions

#### Public, private, static members

Мы можем присвоить метод самому классу. Такие методы называются статическими.

В объявление класса они добавляются с помощью ключевого слова static.

class User {

static staticMethod() {

alert(this === User);

}

}

User.staticMethod(); // true

Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

Обычно статические методы используются для реализации функций, которые будут принадлежать классу в целом, но не какому-либо его конкретному объекту.

Статические свойства также возможны, они выглядят как свойства класса, но с static в начале.

Статические свойства и методы наследуются.

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.

Внешний интерфейс – методы и свойства, доступные снаружи класса.

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

1. Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
2. Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Защищённые свойства обычно начинаются с префикса \_.

Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению.

Приватные свойства и методы должны начинаться с #. Они доступны только внутри класса.

На уровне языка # является специальным символом, который означает, что поле приватное. Мы не можем получить к нему доступ извне или из наследуемых классов.

Приватные поля не конфликтуют с публичными. У нас может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

В отличие от защищённых, функциональность приватных полей обеспечивается самим языком.

https://learn.javascript.ru/static-properties-methods

#### Know how to create public/static/private members

#### Understand OOP emulation patterns and conventions (optional)

### ECMAScript Classes

#### Class declaration

Базовый синтаксис выглядит так:

class MyClass {

// методы класса

constructor() { ... }

method1() { ... }

method2() { ... }

method3() { ... }

...

}

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами.

При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

https://learn.javascript.ru/class

#### Know class declaration syntax

#### Understand difference between class and constructor function

В JavaScript класс – это разновидность функции.

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

}

// доказательство: User - это функция

alert(typeof User); // function

Вот что на самом деле делает конструкция class User {...}:

1. Создаёт функцию с именем User, которая становится результатом объявления класса. Код функции берётся из метода constructor (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как sayHi, в User.prototype.

При вызове метода объекта new User он будет взят из прототипа, как описано в главе F.prototype. Таким образом, объекты new User имеют доступ к методам класса.

https://learn.javascript.ru/class

#### Getter/setter

Как и в литеральных объектах, в классах можно объявлять вычисляемые свойства, геттеры/сеттеры и т.д.

Вот пример user.name, реализованного с использованием get/set:

class User {

constructor(name) {

// вызывает сеттер

this.name = name;

}

get name() {

return this.\_name;

}

set name(value) {

if (value.length < 4) {

alert("Имя слишком короткое.");

return;

}

this.\_name = value;

}

}

let user = new User("Иван");

alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.

При объявлении класса геттеры/сеттеры создаются на User.prototype

https://learn.javascript.ru/class

#### What does super() do and where we have to use it?

Наследование классов – это способ расширения одного класса другим классом.

Таким образом, мы можем добавить новый функционал к уже существующему.

##### Переопределение методов

По умолчанию все методы, не указанные в классе Rabbit, берутся непосредственно «как есть» из класса Animal.

Но если мы укажем в Rabbit собственный метод, например stop(), то он будет использован вместо него:

class Rabbit extends Animal {

stop() {

// ...теперь это будет использоваться для rabbit.stop()

// вместо stop() из класса Animal

}

}

Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово "super" для таких случаев.

* super.method(...) вызывает родительский метод.
* super(...) для вызова родительского конструктора (работает только внутри нашего конструктора).

У стрелочных функций нет super

Конструкторы в наследуемых классах должны обязательно вызывать super(...), и (!) делать это перед использованием this..

### Prototypal Inheritance Basics

#### \_\_proto\_\_ property

Обратите внимание, что \_\_proto\_\_ — не то же самое, что внутреннее свойство [[Prototype]]. Это геттер/сеттер для [[Prototype]]. Позже мы увидим ситуации, когда это имеет значение, а пока давайте просто будем иметь это в виду, поскольку мы строим наше понимание языка JavaScript.

Свойство \_\_proto\_\_ немного устарело, оно существует по историческим причинам. Современный JavaScript предполагает, что мы должны использовать функции Object.getPrototypeOf/Object.setPrototypeOf вместо того, чтобы получать/устанавливать прототип. Мы также рассмотрим эти функции позже.

По спецификации \_\_proto\_\_ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его. Так что мы вполне безопасно его используем.

#### Understand \_\_proto\_\_ object property

#### Able to use [Object.create] and define \_\_proto\_\_ explicitly

Свойство \_\_proto\_\_ считается устаревшим, и по стандарту оно должно поддерживаться только браузерами.

Современные же методы это:

Object.create(proto, [descriptors]) – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.

Object.getPrototypeOf(obj) – возвращает свойство [[Prototype]] объекта obj.

Object.setPrototypeOf(obj, proto) – устанавливает свойство [[Prototype]] объекта obj как proto.

Эти методы нужно использовать вместо \_\_proto\_\_.

У Object.create есть необязательный второй аргумент: дескрипторы свойств. Мы можем добавить дополнительное свойство новому объекту таким образом:

let animal = {

eats: true

};

let rabbit = Object.create(animal, {

jumps: {

value: true

}

});

alert(rabbit.jumps); // true

Мы также можем использовать Object.create для «продвинутого» клонирования объекта, более мощного, чем копирование свойств в цикле for..in:

// клон obj c тем же прототипом (с поверхностным копированием свойств)

let clone = Object.create(Object.getPrototypeOf(obj), Object.getOwnPropertyDescriptors(obj));

Такой вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и неперечисляемые, геттеры/сеттеры для свойств – и всё это с правильным свойством [[Prototype]].

Как было сказано в начале этой секции учебника, \_\_proto\_\_ – это способ доступа к свойству [[Prototype]], это не само свойство [[Prototype]].

Теперь, если мы хотим использовать объект как ассоциативный массив, мы можем сделать это с помощью небольшого трюка:

let obj = Object.create(null);

let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // "some value"

Object.create(null) создаёт пустой объект без прототипа ([[Prototype]] будет null):

Таким образом не будет унаследованного геттера/сеттера для \_\_proto\_\_. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно.

Мы можем назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще, чем обычные объекты {...}.

Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как toString. Но обычно это нормально для ассоциативных массивов.

https://learn.javascript.ru/prototype-methods

#### prototype property

Новые объекты могут быть созданы с помощью функции-конструктора new F().

Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

Обратите внимание, что F.prototype означает обычное свойство с именем "prototype" для F. Это ещё не «прототип объекта», а обычное свойство F с таким именем.

F.prototype используется только при вызове new F и присваивается в качестве свойства [[Prototype]] нового объекта.

Если после создания свойство F.prototype изменится (F.prototype = <другой объект>), то новые объекты, созданные с помощью new F, будут иметь в качестве [[Prototype]] другой объект, а уже существующие объекты сохранят старый.

По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор.

#### Know function prototype property

#### Understand dependency between function constructor prototype and instance \_\_proto\_\_

#### Able to create 'class' methods using function prototype property

Мы можем использовать свойство constructor существующего объекта для создания нового.

function Rabbit(name) {

this.name = name;

alert(name);

}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

Это удобно, когда у нас есть объект, но мы не знаем, какой конструктор использовался для его создания (например, он мог быть взят из сторонней библиотеки), а нам необходимо создать ещё один такой объект.

https://learn.javascript.ru/function-prototype

#### Able to set / get object prototype (optional)

Метод Object.setPrototypeOf() устанавливает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта в другой объект или null.

Object.setPrototypeOf(obj, prototype);

Метод Object.getPrototypeOf() возвращает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта.

Object.getPrototypeOf(obj)

### ECMAScript Advanced Data Types & Expressions

#### Set/Map data types

##### Map

Map – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа.

Методы и свойства:

new Map() – создаёт коллекцию.

map.set(key, value) – записывает по ключу key значение value.

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 (пару «ключ/значение») по ключу key.

map.clear() – очищает коллекцию от всех элементов.

map.size – возвращает текущее количество элементов.

Map может использовать объекты в качестве ключей.

Чтобы сравнивать ключи, объект Map использует алгоритм SameValueZero. Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа.

Этот алгоритм не может быть заменён или модифицирован.

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.

Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array.

Если у нас уже есть обычный объект, и мы хотели бы создать Map из него, то поможет встроенный метод Object.entries(obj), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате.

Так что мы можем создать Map из обычного объекта следующим образом:

let obj = {

name: "John",

age: 30

};

let map = new Map(Object.entries(obj));

##### Set

Объект Set – это особый вид коллекции: «множество» значений (без ключей), где каждое значение может появляться только один раз.

Его основные методы:

new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.

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.

Основная «изюминка» – это то, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

Мы можем перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach

https://learn.javascript.ru/map-set

#### WeakSet/WeakMap data types

##### WeakMap

WeakMap не предотвращает удаление объектов сборщиком мусора, когда эти объекты выступают в качестве ключей.

Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными значениями

WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё.

В WeakMap присутствуют только следующие методы:

weakMap.get(key)

weakMap.set(key, value)

weakMap.delete(key)

weakMap.has(key)

##### WeakSet

Коллекция WeakSet ведёт себя похоже:

Она аналогична Set, но мы можем добавлять в WeakSet только объекты (не примитивные значения).

Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.

Как и Set, она поддерживает add, has и delete, но не size, keys() и не является перебираемой.

https://learn.javascript.ru/weakmap-weakset

### JavaScript Errors

#### JavaScript Errors (throw, Error class)

Оператор throw генерирует ошибку.

Синтаксис:

throw <объект ошибки>

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенными ошибками).

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);

// ...

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство name – это в точности имя конструктора. А свойство message берётся из аргумента.

1. Блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.
2. Техника «проброс исключения» выглядит так:
3. Блок catch получает все ошибки.

В блоке catch(err) {...} мы анализируем объект ошибки err.

Если мы не знаем как её обработать, тогда делаем throw err.

В коде ниже мы используем проброс исключения, catch обрабатывает только SyntaxError:

let json = '{ "age": 30 }'; // данные неполны

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени");

}

blabla(); // неожиданная ошибка

alert( user.name );

} catch(e) {

if (e.name == "SyntaxError") {

alert( "JSON Error: " + e.message );

} else {

throw e; // проброс (\*)

}

}

Ошибка в строке (\*) из блока catch «выпадает наружу» и может быть поймана другой внешней конструкцией try..catch (если есть), или «убьёт» скрипт.

Таким образом, блок catch фактически обрабатывает только те ошибки, с которыми он знает, как справляться, и пропускает остальные.

#### try..catch statement

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

try {

// код...

} catch (err) {

// обработка ошибки

}

Работает она так:

Сначала выполняется код внутри блока try {...}.

Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.

Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.

Конструкция try..catch может содержать ещё одну секцию: finally.

Если секция есть, то она выполняется в любом случае:

* после try, если не было ошибок,
* после catch, если ошибки были.

Расширенный синтаксис выглядит следующим образом:

try {

... пробуем выполнить код...

} catch(e) {

... обрабатываем ошибки ...

} finally {

... выполняем всегда ...

}

https://learn.javascript.ru/try-catch

#### Error handling

#### Error class

#### error logging

#### async error events

try..catch работает синхронно

Чтобы поймать исключение внутри запланированной функции, try..catch должен находиться внутри самой этой функции:

setTimeout(function() {

try {

noSuchVariable; // try..catch обрабатывает ошибку!

} catch {

alert( "ошибка поймана!" );

}

}, 1000);

#### Custom errors (optional)

Класс Error встроенный, вот его примерный код, просто чтобы мы понимали, что расширяем:

// "Псевдокод" встроенного класса Error, определённого самим JavaScript

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

}

}

Теперь давайте унаследуем от него ValidationError и попробуем новый класс в действии:

class ValidationError extends Error {

constructor(message) {

super(message); // (1)

this.name = "ValidationError"; // (2)

}

}

function test() {

throw new ValidationError("Упс!");

}

try {

test();

} catch(err) {

alert(err.message); // Упс!

alert(err.name); // ValidationError

alert(err.stack); // список вложенных вызовов с номерами строк для каждого

}

Обратите внимание: в строке (1) вызываем родительский конструктор. JavaScript требует от нас вызова super в дочернем конструкторе, так что это обязательно. Родительский конструктор устанавливает свойство message.

Родительский конструктор также устанавливает свойство name для "Error", поэтому в строке (2) мы сбрасываем его на правильное значение.

Попробуем использовать его в readUser(json):

class ValidationError extends Error {

constructor(message) {

super(message);

this.name = "ValidationError";

}

}

// Использование

function readUser(json) {

let user = JSON.parse(json);

if (!user.age) {

throw new ValidationError("Нет поля: age");

}

if (!user.name) {

throw new ValidationError("Нет поля: name");

}

return user;

}

// Рабочий пример с try..catch

try {

let user = readUser('{ "age": 25 }');

} catch (err) {

if (err instanceof ValidationError) {

alert("Некорректные данные: " + err.message); // Некорректные данные: Нет поля: name

} else if (err instanceof SyntaxError) { // (\*)

alert("JSON Ошибка Синтаксиса: " + err.message);

} else {

throw err; // неизвестная ошибка, пробросить исключение (\*\*)

}

}

Блок try..catch в коде выше обрабатывает и нашу ValidationError, и встроенную SyntaxError из JSON.parse.

Обратите внимание, как мы используем instanceof для проверки конкретного типа ошибки в строке (\*).

https://learn.javascript.ru/custom-errors

### ECMAScript Advanced

#### Promises

Синтаксис создания Promise:

let promise = new Promise(function(resolve, reject) {

// функция-исполнитель (executor)

// "певец"

});

Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: исполнитель – это «певец».

Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

* resolve(value) — если работа завершилась успешно, с результатом value.
* reject(error) — если произошла ошибка, error – объект ошибки.

Итак, исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать resolve или reject.

https://learn.javascript.ru/promise-basics

#### Promise states

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

* state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.
* result («результат») — вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

https://learn.javascript.ru/promise-basics

#### Promise chaining

Объект Promise служит связующим звеном между исполнителем и функциями-потребителями, которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы с помощью методов .then и .catch.

Наиболее важный и фундаментальный метод – .then.

promise.then(

function(result) { /\* обработает успешное выполнение \*/ },

function(error) { /\* обработает ошибку \*/ }

);

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает то же самое:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Ошибка!")), 1000);

});

// .catch(f) это то же самое, что promise.then(null, f)

promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду

По аналогии с блоком finally из обычного try {...} catch {...}, у промисов также есть метод finally.

Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

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();

});

Здесь каждый вызов loadScript возвращает промис, и следующий обработчик в .then срабатывает, только когда этот промис завершается. Затем инициируется загрузка следующего скрипта и так далее. Таким образом, скрипты загружаются один за другим.

#### Promise static methods

##### Promise.all(iterable)

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.

##### Promise.allSettled(iterable)

Ожидает завершения всех полученных промисов (как исполнения так и отклонения).

Возвращает промис, который исполняется когда все полученные промисы завершены (исполнены или отклонены), содержащий массив результатов исполнения полученных промисов.

##### Promise.race(iterable)

Ожидает исполнения или отклонения любого из полученных промисов.

Возвращает промис, который будет исполнен или отклонён с результатом исполнения первого исполненного или отклонённого промиса из iterable.

##### Promise.reject(reason)

Возвращает промис, отклонённый из-за reason.

##### Promise.resolve(value)

Возвращает промис, исполненный с результатом value.

##### Promise.any

Метод очень похож на Promise.race, но ждёт только первый успешно выполненный промис, из которого берёт результат.

#### Be able to compare promise and callback patterns (optional)

Например, у нас есть loadScript(src, callback).

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) => {...})

Давайте промисифицируем её. Новая функция loadScriptPromise(src) будет делать то же самое, но будет принимать только src (не callback) и возвращать промис.

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(...)

#### Be able to handle errors in promises

Цепочки промисов отлично подходят для перехвата ошибок. Если промис завершается с ошибкой, то управление переходит в ближайший обработчик ошибок.

Самый лёгкий путь перехватить все ошибки – это добавить .catch в конец цепочки:

fetch('/article/promise-chaining/user.json')

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

.then(user => fetch(`https://api.github.com/users/${user.name}`))

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

.then(githubUser => new Promise((resolve, reject) => {

let img = document.createElement('img');

img.src = githubUser.avatar\_url;

img.className = "promise-avatar-example";

document.body.append(img);

setTimeout(() => {

img.remove();

resolve(githubUser);

}, 3000);

}))

.catch(error => alert(error.message));

Если мы пробросим (throw) ошибку внутри блока .catch, то управление перейдёт к следующему ближайшему обработчику ошибок. А если мы обработаем ошибку и завершим работу обработчика нормально, то продолжит работу ближайший успешный обработчик .then.

В примере ниже .catch успешно обрабатывает ошибку:

// the execution: catch -> then

new Promise((resolve, reject) => {

throw new Error("Ошибка!");

}).catch(function(error) {

alert("Ошибка обработана, продолжить работу");

}).then(() => alert("Управление перейдёт в следующий then"));

Что происходит, когда обычная ошибка не перехвачена try..catch? Скрипт умирает с сообщением в консоли. Похожее происходит и в случае необработанной ошибки промиса.

https://learn.javascript.ru/promise-error-handling

#### async/await

Существует специальный синтаксис для работы с промисами, который называется «async/await».

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("готово!"), 1000)

});

let result = await promise; // будет ждать, пока промис не выполнится (\*)

alert(result); // "готово!"

}

f();

В данном примере выполнение функции остановится на строке (\*) до тех пор, пока промис не выполнится.

Ошибки можно ловить, используя try..catch, как с обычным throw:

async function f() {

try {

let response = await fetch('http://no-such-url');

} catch(err) {

alert(err); // TypeError: failed to fetch

}

}

f();

Если у нас нет try..catch, асинхронная функция будет возвращать завершившийся с ошибкой промис (в состоянии rejected). В этом случае мы можем использовать метод .catch промиса, чтобы обработать ошибку:

async function f() {

let response = await fetch('http://no-such-url');

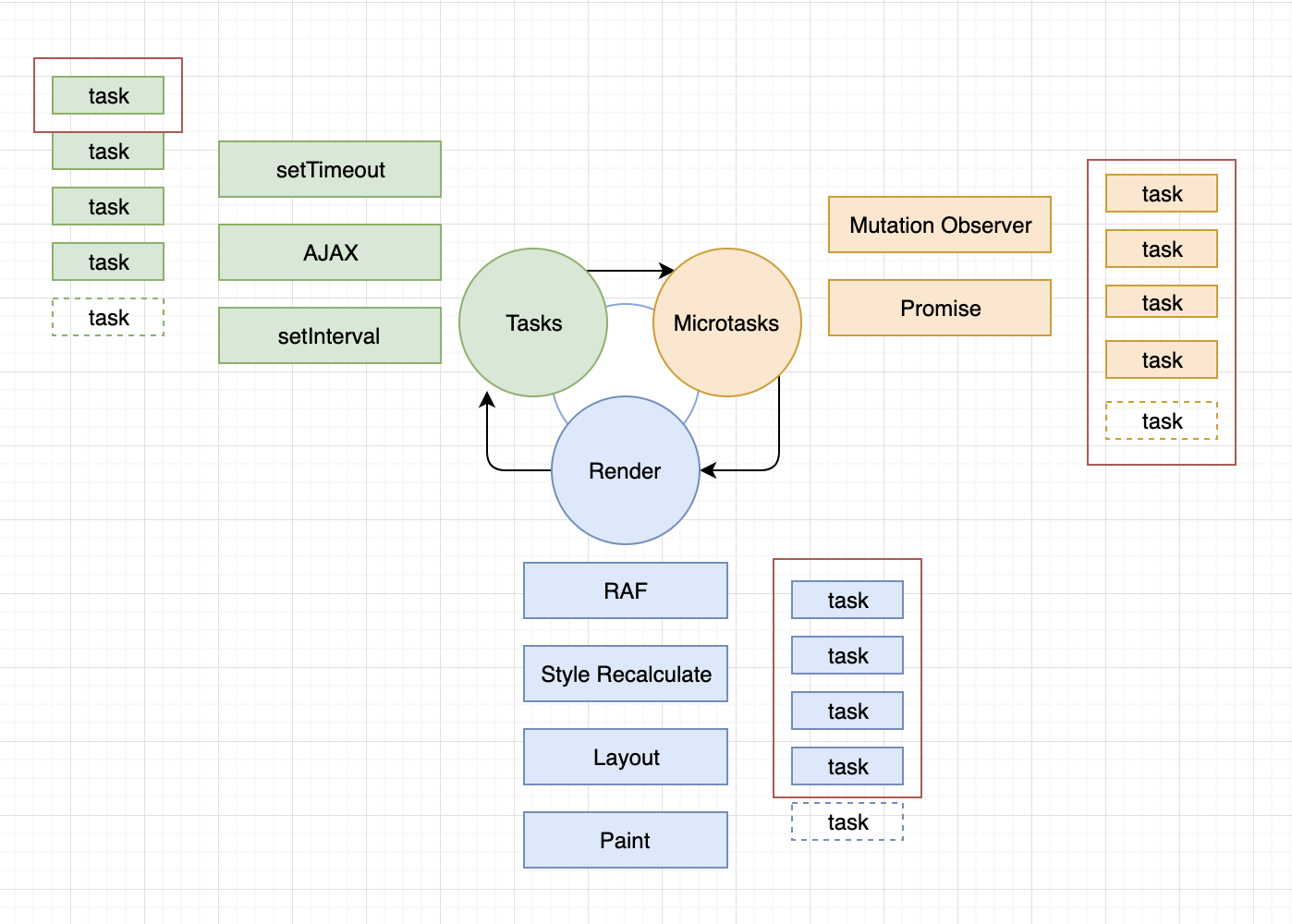
}

// f() вернёт промис в состоянии rejected

f().catch(alert); // TypeError: failed to fetch // (\*)

https://learn.javascript.ru/async-await

#### event loop



После того как мы начали выполнять какой-либо script, в очередь Tasks ставится задача с выполнением этого скрипта. По мере выполнения этого кода, нам встречаются задачи от разных заказчиков, которые ставятся в соответствующие очереди. После того как завершается задача по выполнению скрипта (задача от Tasks), Event Loop идет к Microtasks (после задачи от Tasks Event Loop берет задачи от Microtasks). У него Event Loop берет задачи до тех пор, пока они не закончатся. Это значит, что если время их добавления равно времени их выполнения, то Event Loop будет бесконечно их разгребать.

Далее он идет к Render и выполняет задачи от него. Задачи от Render оптимизируются браузером и, если он посчитает, что в этом цикле не нужно ничего перерисовывать, то Event Loop просто пойдет дальше. Далее Event Loop снова берет задачи от Tasks и просит у него только одну, первую в очереди задачу, передает ее в CallStack и идет дальше по циклу.

https://habr.com/ru/articles/461401/

#### Garbage collector (concept) (optional)

Основной концепцией управления памятью в JavaScript является принцип достижимости.

Если упростить, то «достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

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

Например:

* Выполняемая в данный момент функция, её локальные переменные и параметры.
* Другие функции в текущей цепочке вложенных вызовов, их локальные переменные и параметры.
* Глобальные переменные.
* (некоторые другие внутренние значения)

Эти значения мы будем называть корнями.

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

Например, если в глобальной 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.

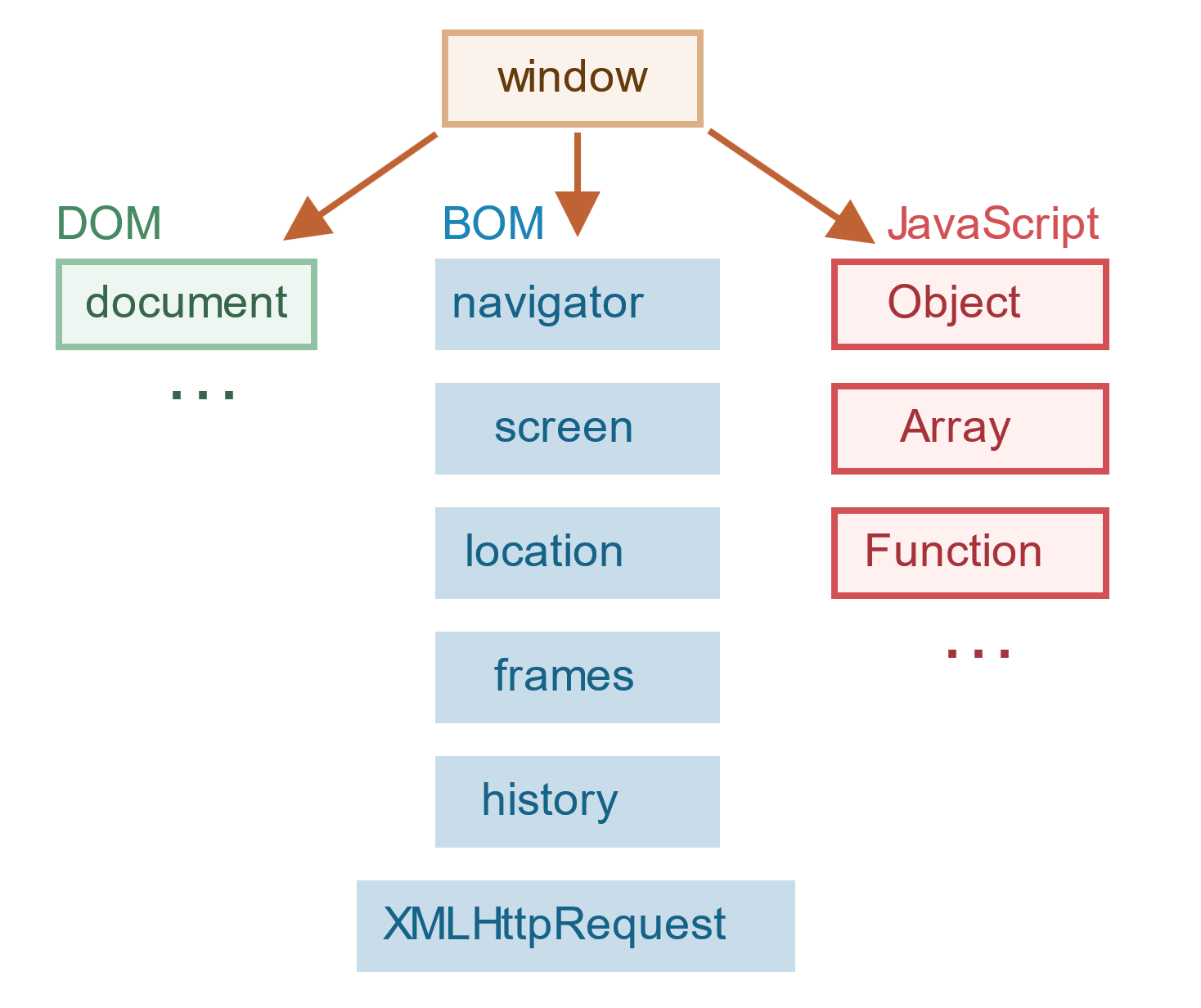
В движке JavaScript есть фоновый процесс, который называется сборщиком мусора. Он отслеживает все объекты и удаляет те, которые стали недоступными.

## JavaScript in Browser:

### Global object window

#### Location

На картинке ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:



Как мы видим, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода (Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения.).
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Document Object Model, сокращённо DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

Объект document – основная «входная точка». С его помощью мы можем что-то создавать или менять на странице.

https://learn.javascript.ru/browser-environment

Глобальный объект - это объект, который всегда существует в глобальной области видимости (en-US).

В JavaScript всегда определён глобальный объект. В веб-браузере, когда скрипты создают глобальные переменные, они создаются как свойства глобального объекта. (В Node.js это не так.) Interface глобального объекта зависит от контекста, в котором выполняется скрипт.К примеру:

* В веб-браузере любой код, который не запускается скриптом явно как фоновую задачу, имеет Window в качестве своего глобального объекта. Это покрывает большую часть JavaScript-кода в сети.
* Код, работающий в Worker имеет WorkerGlobalScope (en-US) объект в качестве своего глобального объекта.
* Скрипты, работающие в Node.js имеют объект, который называется global в качестве своего глобального объекта.

Объект window - Глобальный Объект в браузере. Доступ к любым Глобальным Переменным или функциям может быть получен как к свойствам объекта window.

var foo = "foobar";

foo === window.foo; // Возвращает: true

После определения Глобальной Переменной foo, мы можем получить доступ к его значению прямо с объекта window, использую имя переменной foo в качестве имени свойства Глобального Объекта window.foo.

#### Know browser location structure

Интерфейс Location представляет собой адрес (URL) объекта, с которым он связан. Его модификации отражаются на родительском объекте. Интерфейсы Document и Window имеют подобный Location, к которому можно получить доступ через Document.location и Window.location соответственно.

##### Свойства

Интерфейс Location не имеет унаследованных свойств, но реализует свойства URLUtils.

Location.href (en-US)

DOMString, содержащий URL целиком. При изменении, соответствующий документ переходит на новую страницу.

Location.protocol (en-US)

DOMString, содержащий протокол текущего URL, включая ':'.

Location.host (en-US)

DOMString, содержащий хост, а именно имя хоста, ':' и порт.

Location.hostname (en-US)

DOMString, содержащий домен текущего URL.

Location.port (en-US)

DOMString, содержащий номер порта текущего URL.

Location.pathname (en-US)

DOMString, содержащий первый '/' после хоста с последующим текстом URL.

Location.search (en-US)

DOMString, содержащий '?' с последующими параметрами URL.

Location.hash (en-US)

DOMString, содержащий '#' с последующим идентификатором.

Location.username (en-US)

DOMString, содержащий имя пользователя, указанное перед именем домена.

Location.password (en-US)

DOMString, содержащий пароль, указанный перед именем домена.

Location.origin (en-US) Только для чтения

Возвращает DOMString, содержащий протокол, хост и порт текущего URL.

##### Методы

Интерфейс Location не имеет унаследованных методов, но реализует методы URLUtils.

Location.assign()

Загружает ресурс по URL, указанному в качестве параметра.

Location.reload()

Перезагружает ресурс по текущему URL. Единственный опциональный параметр Boolean (en-US) при значении true указывает, что страница должна быть заново загружена с сервера, при значении false страница может быть загружена из кеша.

Location.replace()

Заменяет текущий ресурс на новый по URL, указанному в качестве параметра. Отличие от assign() в том, что при использовании replace() текущая страница не будет сохранена в History, и пользователь не сможет использовать кнопку назад, чтобы вернуться к ней.

Location.toString() (en-US)

Возвращает DOMString, содержащий URL целиком. Это синоним URLUtils.href, однако он не может использоваться для изменения значения.

https://developer.mozilla.org/ru/docs/Web/API/Location

#### History API (Global object window)

DOM-объект Window предоставляет доступ к истории текущей сессии браузера (не путать с историей браузерных расширений (en-US)) через объект history. Он предоставляет полезные методы и свойства, которые позволяют переходить назад и вперёд по истории пользователя и манипулировать её содержимым.

Понятия и использование

Перемещение назад и вперёд по истории пользователя выполняется с помощью методов back() (en-US), forward() (en-US), и go().

https://developer.mozilla.org/ru/docs/Web/API/History\_API

History интерфейс позволяет манипулировать историей браузера в пределах сессии, а именно историей о посещённых страницах в пределах вкладки или фрейма загруженного внутри страницы.

##### Свойства

History интерфейс не наследует никакие родительские свойства.

History.length Только для чтения

Возвращает целочисленное значение типа Integer, которое характеризует собой количество записей в истории сессии, включая текущую загруженную страницу. Для примера, история новой вкладки с загруженной страницей равна 1.

History.current Только для чтения Non-standard

Возвращает DOMString, представляющий собой активный URL элемент в истории сессии. Это свойство никогда не было доступно для веб-контента и более не поддерживается никакими браузерами. Используйте location.href (en-US) вместо него.

History.next Только для чтения Non-standard

Возвращает DOMString, представляющий собой следующий URL элемент в истории сессии. Это свойство никогда не было доступно для веб-контента и более не поддерживается другими браузерами.

History.previous Только для чтения Non-standard

Возвращает DOMString, представляющий собой предыдущий URL элемент в истории сессии. Это свойство никогда не было доступно для веб-контента и более не поддерживается другими браузерами.

History.state Только для чтения

Возвращает какое-либо значение, представляющее собой состояние в вершине истории стека. Это способ посмотреть на состояние без ожидания popstate (en-US) события.

##### Методы

History интерфейс не наследует никаких родительских методов.

History.back() (en-US)

Делает вызов предыдущей страницы из истории, если она существует. Эквивалент метода history.go(-1).

Примечание: Вызов этого метода, чтобы вернуться к первой странице в истории сессии не имеет никакого эффекта и не вызывает исключений.

History.forward() (en-US)

Переход к следующей странице в истории сессии, то же самое действие, как и при нажатии пользователем кнопки Forward в браузере. Эквивалентно history.go(1).

Примечание: Вызов этого метода, чтобы перейти к последней странице в истории сессии не имеет никакого эффекта и не вызывает исключений.

History.go()

Загружает страницу из истории сессии, определяя её положение относительно текущей страницы, например: -1 для предыдущей страницы или 1 для следующей страницы. Когда integerDelta выходит за предел (например, -1, когда нет ранее посещённых страниц в истории сессии), метод не делает ничего, и не вызывает исключение. Вызов Go () без параметров или с не целочисленным аргументом не имеет никакого эффекта (в отличие от Internet Explorer, который поддерживает строки URL в качестве аргумента).

History.pushState()

Помещает полученные данные в стек истории сессии с определённым заголовком и, при наличии , URL. Данные рассматриваются DOM как непрозрачные; вы можете задать любой объект JavaScript, который может быть сериализован. Обратите внимание, что в настоящее время Firefox игнорирует параметр заголовка; для получения дополнительной информации см. управление историей браузера.

Примечание: В Gecko 2.0 до Gecko 5.0, передаваемый объект сериарилизовался в JSON. Начиная с Gecko 6.0, объект обрабатывается по алгоритму структурированного клонирования. Он позволяет передавать более широкое разнообразие объектов.

History.replaceState()

Обновляет последнюю запись в стеке истории содержащий определённые данные, заголовок и, при наличии, URL. Данные рассматриваются DOM как непрозрачные; вы можете задать любой объект JavaScript, который может быть сериализован. Обратите внимание, что в настоящее время Firefox игнорирует параметр заголовка; для получения дополнительной информации см.управление историей браузера.

https://developer.mozilla.org/ru/docs/Web/API/History

#### Know browser History APIconcept

#### Be able to navigate within browser history

#### Be able to use history state (optional)

#### Navigator (optional)

#### Know how to parse user agent (optional)

#### Know how to discover client platform, browser

#### Cookies

### Page Lifecycle

#### Parsing

#### Reflow

#### Repaint

#### Critical rendering path (CRP) (optional)

#### Events Basics (optional)

#### Custom events (optional)

### Web components (optional)

#### Web components, shadow DOM (concept) (optional)

### Network requests

#### Fetch (with usage)

#### XMLHTTPRequest (concept) (optional)

#### WebSocket (concept) (optional)

### Timers (optional)

#### requestAnimationFrame (optional)

#### Be able to explain difference between setTimeout and requestAnimationFrame (optional)

### Web Storage API & cookies

#### Cookies

#### Difference between localStorage, sessionStorage and cookies

## Typescript:

### Ability to write concise TypeScript code using its constructs

#### basic types

#### enums

#### type / interface, differences between them

#### using interfaces with optional properties, read-only properties, etc...

#### function types

#### utitily types (optional)

#### typeguards (optional)

#### creating custom types

#### generic types (concept)

#### understanding TS (ES6) module system

## Design patterns:

### Creational Design Patterns

### Structural Design Patterns

### Behavioral Design Patterns

### MVC (optional)

### Intermediate knowledge of patterns and best practices:

#### SOLID principles

#### design patterns used on a student's project, and able to compare these patterns (optional)

### Software Development Methodologies (optional)

#### Agile

#### Scrum / Kanban / Waterfall

#### Estimation

### Testing (optional)

#### Testing Types

##### Integration Testing

##### E2E

##### Security Testing

##### Perforamance Testing

#### Test Pyramid

#### Testing approaches (optional)

#### FIRST

#### TDD и BDD

#### Frameworks (optional)

## Web Communication Protocols: (optional)

### HTTP vs HTTPS

### HTTP 1.x, 2.x, 3.x

### HTTP methods, headers, responses, body

### HTTP status codes groups (1xx, 2xx, 3xx, 4xx, 5xx)

### RESTful API

## Common web-security knowledge (optional)

### Basic understanding of most common security terms (CORS, XSS) (optional)

#### XSS

#### CORS

#### OWASP Top 10

#### Auth (JWT, OAuth, Basic, etc.)

## Coding tasks:

#### Function.prototype.bind implement polyfill

let obj = {

name: 'Jack',

};

let myFunc = function (id, city) {

console.log(`${this.name}, ${id}, ${city}`); // id will be undefined

};

// Accepting any number of arguments passed to myBind

Function.prototype.myBind = function (obj, ...args) {

let func = this;

// Accepting arguments passed to newFunc

return function (...newArgs) {

func.apply(obj, [...args, ...newArgs]);

};

};

let newFunc = myFunc.myBind(obj, 'a\_random\_id')

newFunc('New York') // Jack, a\_random\_id, New York

#### Object.create implement polyfill

#### Array.flat implement polyfill

#### Array.reduce implement polyfill

#### 'hello world'.repeating(3) -> 'hello world hello world hello world'. How to implement?

#### myFunc('!', 4, -10, 34, 0) -> '4!-10!34!0`. How to implement?

#### five(plus(seven(minus(three())))) -> 9. How to implement?

function makeNum(num, func) {

if (func === undefined) {

return num;

} else {

return func(num);

}

}

function zero(func) {

return makeNum(0,func);

}

function one(func) {

return makeNum(1,func);

}

function two(func) {

return makeNum(2,func);

}

function three(func) {

return makeNum(3,func);

}

function four(func) {

return makeNum(4,func);

}

function five(func) {

return makeNum(5,func);

}

function six(func) {

return makeNum(6,func);

}

function seven(func) {

return makeNum(7,func);

}

function eight(func) {

return makeNum(8,func);

}

function nine(func) {

return makeNum(9,func);

}

function plus(right) {

return function(left) { return left + right; };

}

function minus(right) {

return function(left) {

return left - right;

};

}

function times(right) {

return function(left) { return left \* right; };

}

function dividedBy(right) {

return function(left) { return left / right; };

}

eight(minus(three())); // return 5

six(dividedBy(two())); // return 3

two(plus(five())); // return 7

five(plus(two())); // return 7

seven(times(five())); // return 35

console.log(four(plus(nine()))); // must return 13

#### add(5)(9)(-4)(1) -> 11. How to implement?

#### periodOutput(period) method should output in the console once per every period how mach time has passed since the first function call. Example: periodOutput(100) -> 100(after 100 ms), 200(after 100 ms), 300(after 100 ms), ...

#### extendedPeriodOutput(period) method should output in the console once per period how mach time has passed since the first function call and then increase the period. Example: // extendedPeriodOutput(100) -> 100(after 100 ms), 200(after 200 ms), 300(after 300 ms)