# CoreJS Interview #2

## 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

#### Reference Type & losing this

#### Understand difference between function and method

#### 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)

#### Know callback pattern

#### Know IIFE pattern (optional)

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

#### Carrying and partial functions

### Object Oriented Programming

#### new keyword

#### Understand how new keyword works

#### Function constructor

#### Know function constructor concept

#### Able to create constructor functions

#### Public, private, static members

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

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

### ECMAScript Classes

#### Class declaration

#### Know class declaration syntax

#### Understand difference between class and constructor function

#### Getter/setter

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

### Prototypal Inheritance Basics

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

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

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

#### prototype property

#### Know function prototype property

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

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

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

#### ECMAScript Advanced Data Types & Expressions

#### Set/Map data types

#### WeakSet/WeakMap data types

### JavaScript Errors

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

#### try..catch statement

#### Error handling

#### Error class

#### error logging

#### async error events

#### Custom errors (optional)

### ECMAScript Advanced

#### Promises

#### Promise states

#### Promise chaining

#### Promise static methods

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

#### Be able to handle errors in promises

#### async/await

#### event loop

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

## JavaScript in Browser:

### Global object window

#### Location

#### Know browser location structure

#### History API (Global object window)

#### 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

#### 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?

#### 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)