Вопросы для собесов.

связанные списки (одно связные, двусвязные)
XSRF - атака
что такое nginx Apache
CI / CD
Какие виде тестов знаешь? (Особенности, плюсы, минусы)
Session id
JWT токен - что такое и как работает?
вопросы по гиту
агрегация, композиция
TDD плюсы и минусы
7 уровней модели osi
Что такое Граф

HTTP запросы, GET POST PUT PATCH и тд. Коды ответов сервера.

как хранятся данные в браузере

перегрузка методов

Hoisting (поднятие объявления функций и переменных)

Механизм, который позволяет вызывать функцию в коде до её объявления. То есть если у нас на 1 строке вызывается функция, которая объявлена на второй или ниже, то при запуске кода, интерпритатор JS поднимает все объявления функций в самый верх нашего кода, соответственно мы можем пользоваться функцией выше того места где она объявлена. С переменными (ЧЕРЕЗ VAR!!!!) дела обстоят чуть по другому. При запуске кода наверх поднимается только ОБЪЯВЛЕНИЕ переменной, без присваивания, соответсвенно, если мы попытаемся отконсолить её значение выше её объявления и присваивания в

коде, то получим undefined , то есть код видит эту переменную, но ей ничего не присвоено, тк поднялось вверх только объявление. Если бы попробовать отконсолить переменную, которой не существует, то мы получим ошибку.

Этот механизм не работает, если мы присвоим переменной функциональное выражение, то же самое не работает и со стрелочными функциями. Поэтому предпочтительнее использовать функции и переменные ПОСЛЕ (НИЖЕ) их объявления.

Семантические теги HTML

Такие теги позволяют доступно объяснить браузеру или другому разрабу о назначении определенной части кода, посредством использования определенного тега.

header, footer, nav, section, main, article, aside и менее популярные figure figcaption (для изображений и подписи изображений) , address, time, mark.

main можно использовать только 1 раз . нельзя вставлять тег header в footer

тег aside используется для вывода какой либо информации сбоку, реклама, баннеры, новости.

тег article имеет смысл использовать для онотипного контента. например текст в новостях.

section имеет смысл использовать для разделения не однозначного контента. то есть разные секции контента, секции, контент в которых несет различный смысл.

Использование семантических тегов позволяет пользоваться вашим сайтом людям с проблемами со зрением , которые используют скрин ридеры.

Контекст вызова

▼ Развернуть

```
const obj = {
a: 10,
b: 20,
  print(){
    console.log(this)
  }
```

```
}

obj.print()
// {a: 10, b:20, print: f}
```

Когда функция существует в рамках какого либо объекта, ключевое слово this всегда ссылается на этот объект. Если мы напишем такой код

```
const obj = {
a: 10,
b: 20,
  print(){
    console.log(this.a)
  }
}

obj.print()
// 10
```

то this.a в данном примере будет обращаться в свойству под именем ‘a’ у объекта this , то есть у нашего obj.

Рассмотрим такой пример, где перепишем метод print у нашего объекта obj

```
obj.print = function() {
setTimeout( function() {
 console.log(this)} , 1000)
}
```

В этом примере после вызова **obj.print()** мы получим в консоли весь объект window целиком, так как в нем находится функция (метод объекта window) **setTimeout** . И наша функция **print()** появился в глобальном объекте **window** как его метод.

Если же этот код переписать со **стрелочной функцией:**

```
obj.print = function() {
setTimeout( () => console.log(this) , 1000)
}
```

То контекст вызова не перейдет в **window** , и такая запись отобразит наш объект **obj** через 1 секунду в консоли.

```
const btn = document.querySelector('button')

function log() {
console.log(this)
}

log() // Выведет весь объект Window
btn.addEventListener('click' , log) // при нажатии на соотв. кнопку
// в консоли мы увидим <button>click me<button>
```

Это происходит из за разного контекста вызова одной и той же функции, первый раз мы просто вызвали её в файле js , а второй раз по нажатию на кнопку, так как запись

```
const btn = document.querySelector('button')
```

создает объект button со множеством методов и свойств. И при помощи this мы получаем доступ ко всем его свойствам.

можно жестко задать определенный контекст для вызова функции с помощью bind()

Скажем функции myFn что она будет вызываться только с контекстом определенного объекта.

```
function myFn() {console.log(this)}

// вызов сразу после объявления
myFn()
// выведет объект Window

//определяем контекст вызова с помощью bind
const myObj = {x: 10, y: 20}
myFn = myFn.bind(myObj)

myFn()

// выведет в консоль наш объект, который ме передали в параметрах bind()
// {x: 10, y: 20}
```

Метод call

```
const person = {
name: 'Roman',
hi() { console.log(` Hello, my name is ${this.name}`)
}

вызов метода person.hi() выведет следующее
Hello, my name is Roman
```

Мы можем вызвать метод hi() из объекта person в контексте другого объекта.

```
const person2 = {
name: 'Valera'
}

person.hi.call(person2)

такой вызов выведет
Hello, my name is Valera
```

Можно добавить метод hi из person для объекта person2

```
person2.hi = person.hi.bind(person2)
```

теперь в объекте person2 у нас появится свой метод hi. Если мы вызовем его так

```
person2.hi()

в консоли получим
Hello, my name is Valera
```

Метод apply

Делает то же что и call. Только помимо this мы можем передать что то ещё.

```
const numbers = [1,2,3,4,5]
Math.max(numbers)
// NaN
```

метод max принимает параметры через запятую и не умеет работать с массивом
при помощи apply можно сделать так

```
Math.max.apply(null, numbers)
// 5
```

условно apply делает то же что и современный spread оператор

```
Math.max(...numbers)
// 5
```

при помощи call можно было бы сделать так

```
Math.max.call(null, 1,2,3,4,5)
// 5
```

Флаги и дескрипторы свойств. Геттеры и Сеттеры

▼ Флаги и дескрипторы

```
const user = {
name: 'Roman'
}

console.log(Object.getOwnPropertyDescriptor(user, 'name'))

value: 'Roman', writable: true, enumerable: true, configurable: true}
configurable: true // указывает можем ли мы удалить value и влиять каким либо обра
зом на другие флаги (enumerable и writable)
enumerable: true // указывает можем ли мы итерировать данное свойство
// например циклом for in и дальше его отобразить
value: "Roman"
writable: true // показывает можем ли мы изменить данное value либо будет
//readonly
```

при стандартном создании объекта, все флаги будут true

можно “настроить” одно свойство объекта таким образом **(defineProperty)**

3 параметра. имя объекта, свойство, объект с настройками для свойства.

```
Object.defineProperty(user, 'name', {
writable: false, // запрещаем изменять
enumerable: false, // запрещаем перечислять в циклах
configurable: false, // запрещаем удалять
})
```

либо же работать сразу с несколькими свойствами **(defineProperties)**

2 параметра. Имя объекта, объект с подобъектами и настройками каждого из свойств.

```
const user = {
  name: 'Roman',
  age: 29
}

Object.defineProperties(user, {
    name : {
      writable: false,
      enumerable: false,
      configurable: false,
    },
    age: {
      writable: false,
      enumerable: false,
      configurable: false,
    }
})
```

Мы можем легко добавлять новые свойства в объект

```
const user = {
  name: 'Roman',
  age: 29
}

user.city = 'Minsk'

console.log(user) //
{name: 'Roman', age: 29, city: 'Minsk'}
```

Мы можем запретить добавлять новые свойства в объект **(preventExtensions)**

этот метод никак не влияет на удаление свойств из объекта

```
Object.preventExtensions(user)
```

следующий метод запрещает удалять и добавлять свойства в объект

но редактирование свойств объекта открыто

```
Object.seal(user)
```

следующее свойство замораживает объект полностью. Не доступен для удаления, добавления и изменения

```
Object.freeze(user)
```

Следующие методы указывают есть ли доступ к соотв. функционалу объекта

```
Object.isExtensible(user) // вернет булево значение (можем ли мы добавлять)
Object.isSealed(user) // вернет булево (можем ли добавить или удалить)
Object.isFrozen(user) // вернет булево (можем ли удалить, добавить или изменить)
```

▼ Геттеры и сеттеры

Геттеры и Сеттеры являются функциями

Внутри объекта два типа свойств - данные (обычные свойства) и аксессеры (геттеры и сеттеры)

```
const user = {
  name: 'Roman',
  secondName: 'Chuchval',
```

```
  age: 29,
  get fullName() {
    return this.name + this. secondName
  }
}

console.log(user.fullName)
// Roman Chuchval
```

геттеры доступны только для чтения

```
const user = {
  name: 'Roman',
  secondName: 'Chuchval',
  age: 29,
  set setName(value) {
    this.name = value
  }
}
user.setName = 'Abrakadabra'
// сеттеры и геттеры не запускаются как функция, в сеттер мы просто
// переприсваиваем значение

console.log(user)
{name: 'Abrakadabra', ..... }
```

Сеттер принимает один параметр (value) но мы можем передать в него массив

```
user.setName = ['Vasya', 'Pupkin']

// соответственно в объекте запишем так

set setname(value) {
  [name, secondName] = value
  this.name = name
  this.secondName = secondName
}

// либо

set setname(value) {
  [this.name, this.secondName] = value
}
```

Геттеры и сеттеры могут иметь одинаковое название. по записи JS понимает, что в сеттер передаются аргументы, а геттер просто выводит значение.

Жизненный цикл события

▼

Когда мы применяем к кнопке слушателя событий, мы указываем какую то функцию, которая будет выполнена при клике. Как параметр в эту функцию мы можем передать объект Event , который формируется на основании самого события. Если это клик, то это будет объект MouseEvent , в котором хранится множество свойст, многие из них указывают на параметры определенного клика, например координаты на экране, зажата ли была кнопка альт или контрл в момент клика и тд. Мы можем доставать из этого объекта различные свойства и как то использовать их в нашей функции.

если мы не хотим передавать событие клика выше (отменить всплытие), то можно в функции которая “висит” на кнопке использовать метод объекта event.stopPropagation()

и событие не будет всплывать выше. для отлова события при погружении, addEventListener принимает опциональный параметр capture: true. по умолчанию он false. Если передать true , то события будут срабатывать сразу после клика на кнопку, но на элементах, которые находятся “выше” кнопки по дереву дом элементов. например body, div, button, div , body - это пример, когда на трех элементах висят обработчики событий и сначала происходит погружение, потом после достижения целевого элемента, по которому был сделан клик, событие начинает всплывать.

event.target указывает на элемент, ИМЕННО ПО КОТОРОМУ был сделан клик, а

event.currentTarget указывает на все остальные элементы, по которым “всплыл” клик выше по дереву html элементов.

Если повесить обработчик клика на window и вывести в консоль event.target, то в консоли будет тот элемент, именно по которому был сделан клик, где то внизу дерева.

Жизненный цикл события проходит 3 фазы

Фаза погружения - событие сначала идёт сверху вниз по элементам

Фаза цели - событие достигло целевого исходного элемента

Фаза всплытия - событие начинает всплывать снизу вверх

При клике по элементу событие проходит по цепочке родителей сверху вниз, достигает свой целевой элемент, по которому был сделан клик, а потом идет наверх вызывая по пути обработчик.

DOM

В браузерном окружении существует объект Window.

В нем находятся подобъекты трех категорий JS, BOM и DOM

BOM - browser object model, доп объекты предоставляемые браузером, для работы со всем кроме документа. можно получать различные данные о браузере, об операционной системе и тд. Там так же находятся методы alert, confirm, prompt.

DOM - Document object model представляет собой всё содержимое HTML страницы, которые можно менять. Каждый тег HTML является объектом. все вложенные теги являются потомками своего родителя, то есть дочерними, текст и комментарии в коде тоже являются объектами. все эти объекты доступны в JS и можем на них влиять.

Основная точка входа в ДОМ это **объект document** из него получаем доступ к любому узлу. и самые верхние элементы дерева доступны как свойства этого объекта. такие как **html head body**

у объекта document есть свойства, с помощью которых в JS мы можем находить нужные нам элементы из html

```
const elements = document.querySelectorAll('.input')
в переменной elements появится коллекция, в которой будет список всех найденных объект
```

```
ов
```

Коллекция это псевдомассив, для перебора коллекции мы можем использовать цикл for of , но методы массива не будут работать.

Так же в объекте document мы можем найти нужный нам объект html через метод

```
const element = document.getElementById('elementId')
```

Можно найти все элементы по имени тега и получить коллекцию

```
const elements = document.getElementsByTagName('li')
```

поиск по имени класса и получаем коллекцию найденных элементов

```
const elements = document.getElementsByClassName('li')
```

TRY CATCH FINALLY

конструкция позволяет отлавливать ошибки и принимать решение, блокировать последующее выполнение кода, либо каким то образом обработать ошибку и продолжить выполнять программу.

в блоке try помещается код, который потенциально может вызвать ошибку которую мы хотим отловить, если же ошибка не произошла, мы просто продолжаем программу, игнорируя блок catch. Если же ошибка произошла, она залетает в блок catch как параметр, и мы можем его обработать.

Блок finally выполнится в любом случае, была ли ошибка отловлена, либо ошибки при выполнении кода в блоке try не было вовсе.

связка блоков try - catch обязательна. блок finally опционален.

```
try {
..... some code
}
```

```
catch(error){
console.log(error)
}
finally {
... выполнится в любом случае, не важно была ли ошибка, или нет
}
```

eval()

Выполнение кода JavaScript с текстовой строки - это невероятный риск для безопасности. Злоумышленнику слишком легко запустить какой угодно код, когда вы используете `eval()`

Метод `eval()`
выполняет JavaScript-код, представленный строкой.

eval() это функция глобального объекта. параметром принимает строку, которая может быть представлена JS выражением, оператором или последовательностью операторов.

Возвращает значение выполненного кода, либо undefined если код не возвращает ничего.

Object.defineProperty()

метод глобального объекта Object , определяющий новое свойство или изменяющий существующее свойство непосредственно на объекте, возвращая этот объект. и позволяющее произвести настройку свойств.

Object.getOwnPropertyDescriptors()

Метод `Object.getOwnPropertyDescriptors()`
возвращает все собственные дескрипторы свойств данного объекта.

```
const object1 = {
property1: 42
};
const descriptors1 = Object.getOwnPropertyDescriptors(object1);
```

```
console.log(descriptors1.property1.writable);
// Expected output: true

console.log(descriptors1.property1.value);
// Expected output: 42
```

Наследование в классах

Например создав конструктор на базе класса, мы можем определить какие то методы и свойства, которые будут у каждого объекта, который будет создан через конструктор данного класса.

Но если мы решим, что методов и свойств, которые нам нужны в объекте, не достаточно и мы решим добавить что нибудь ещё. Мы можем создать новый класс, расширив уже имеющийся родительский.

например

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(this.name + 'Hello')
  }
}

const alex = new User('Alex')
// User {name: 'Alex'}

class Person extends User {
  constructor(name, email) {
    super(name)
    this.email = email;
  }
}

const roman = new Person('Roman', 'roman@gmail.com')

// Person {name: 'Roman', email: 'roman@gmail.com'}

roman.sayHello() // методы объявленные в родителе доступны в дочерних классах.
```

что такое super - при вызове super() попадает в конструктор родителя и забирает от туда все свойства, которые определены в конструкторе родителя и добавляет их в конструктор дочернего конструктора. через super мы можем

обратится к методом родительского класса из дочернего. Если цепочка классов длинная, то super будет по очереди вверх выпрыгивать и искать, либо метод, либо данные из конструтора, в зависимости от того, где мы вызовем его,

вызов супера в конструктора дочернего класса будет искать данные в конструкторах родительского класса, а если вызываем супер в методе, и вызываем у супера какой то метод, которые есть в родителях, то супер будет искать этот метод в методах у родителей.

SCRUM

Это фреймворк Agile.

В SCRUM есть 3 роли

Product Owner - голос клиента. общее видение. понимает бизнес и значимость системы.

Scrum master - мост между owner и командой. его задача наладка стабильного процесса и улучшение процесса на основании получаемых данных.

Team - команда разработчиков.

Events:

Спринт - отрезок от недели до месяца в течении которого должен быть выполнено опр скоуп задач. его длительность фиксированна и его длительность определяется на старте проекта.

Spring planning - формируется скоуп задач на спринт.

daily scrum - дэйли митинг, на котором все делятся, кто что сделал за вчера. нужен для выявление всех проблем, и быстрого их решения и что планируют делать сегодня.

grooming - выставление абстрактных баллов сложности определенных задач.

sprint review - показ результатов спринта

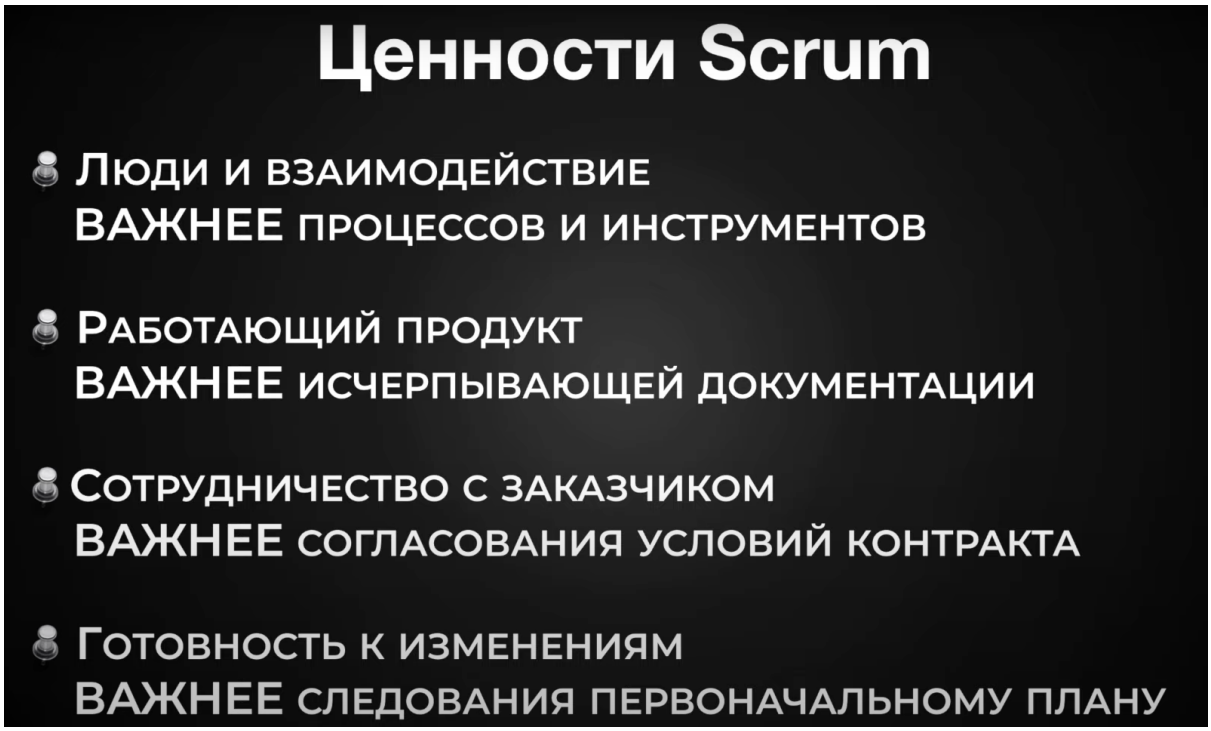
retrospective - обсуждение результатов спринта, что можно улучшить и тд. инфа передается скрам мастеру для работы нам этим

Артефакты

product backlog - большой список задач всего продукта. Приоритизация и тд. этой работой занимается продукт оунер

sprint backlog - список задач на спринт

product increment - график показывает как выполняется работа. т.к как задачи переходят из состоянии “нужно сделать” в “сделано”



Set, Map

в ES6 добавилось 2 коллекции Set и Map . weak set и weak map . по своей природе они похожи на объекты.

в отличии от объекта где ключем могут быть только строки в map ключами могут быть разные значения.

```
let map = new Map()

map.set('str', 'string')
map.set(1, 'number')
map.set(true, 'boolean')

.set добавляет элемент в коллекцию
.get позволяет получить элемент из коллекции
.size показывает количество элементов в коллекции
```

```
.has(принимаеь значение ключа) возвращает true или false (есть ли элемент
с таким ключем в коллекции)
map.delete(1) удаляет элемент

для итерации можно использовать
keys() -> итерируемый объект для ключей
values() -> итерируемый объект для значений
entries() -> итерируемый объект для пар ключ-значение (для цикла for of)
```

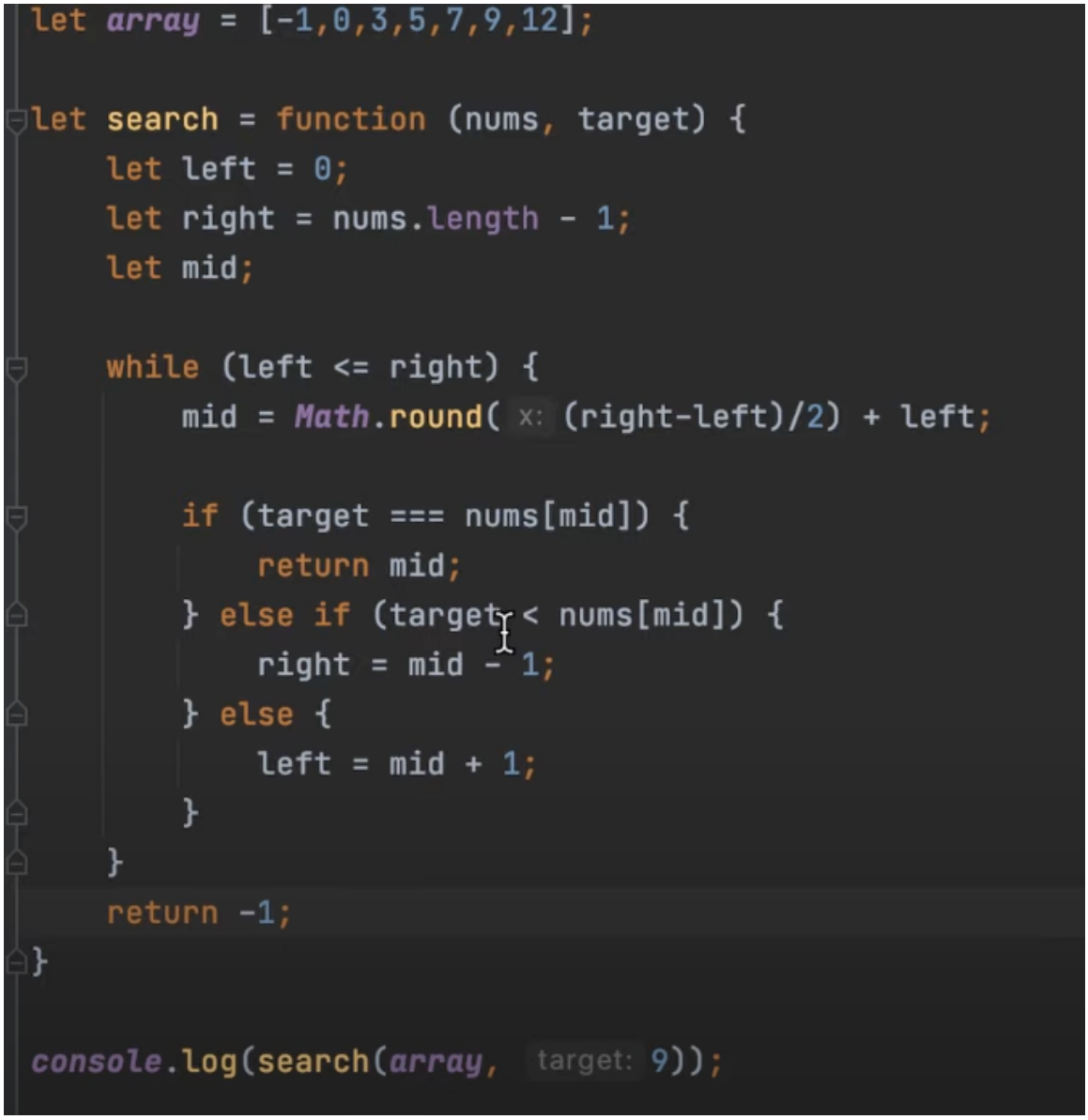
set - коллекция для хранения множества значений, где каждое значение уникально.

```
let jack = {name: 'Jack'}
let roma = {name: 'Roma'}
let leo = {name: 'Leo'}

let users - new Set()

users
  .add(jack)
  .add(roma)
  .add(leo)
  .add(jack)
  .add(roma)

users.size // 3. так как могут быть только уникальные объекты. Они не должны
//повторяться

у элементов коллекции set нет ключей.

.add добавляет item
.delete(item) -> true false
.has(item) -> true false
.clear() очищает коллекцию
```

Бинарный поиск

Это эффективный алгоритм поиска. Например если нам нужно найти имя в массиве, мы должны отсортировать массив по алфавиту, далее разделить его по полам и определить, в какой части находится наше имя, далее делим часть, в которой будет искомое имя ещё раз по полам и делаем тоже самое.

Сложность такого алгоритма O (log n) так как на каждой итерации массив уменьшается в 2 раза.



```
let array = [-1,0,3,5,7,9,12];

let search = function (nums, target) {
    let left = 0;
    let right = nums.length - 1;
    let mid;

    while (left <= right) {
        mid = Math.round( x: (right-left)/2) + left;

        if (target === nums[mid]) {
            return mid;
        } else if (target < nums[mid]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return -1;
}

console.log(search(array,  target: 9));
```

загрузка скрипта через defer и async

Решает проблему, когда при загрузке скрипта сверху файла еще не построено дерево ДОМ элементов и скрипт не видит и не может с ними взаимодействовать. Можно поместить скрипт в конец файла, после загрузки

всего дом дерева, это не лучшее решение, так как при медленном соединении может произойти заметная задержка.

При использовании defer мы сообщаем браузеру, что нужно дальше обрабатывать страницу и грузить скрипт в фоновом режиме, а запустить скрипт, когда ДОМ дерево будет достроено.

Атрибут `defer` предназначен только для внешних скриптов

Атрибут `defer` будет проигнорирован, если в теге `<script>` нет `src`.

При использовании async . Этот атрибут говорит о том, что скрипт полностью независим. Страница не ждет загрузки скриптов, содержимое обрабатывается и отображается. Если есть несколько скриптов с async они будут отрабаывать одновременно и то, что загрузится первее, первым и запустится.

Асинхронные скрипты очень полезны для добавления на страницу сторонних скриптов: счётчиков, рекламы и т.д. Они не зависят от наших скриптов, и мы тоже не должны ждать их