**JS**

* Комменты:

//текст или /\*текст \*/

* Переменная (без пробелов, не начинается с цифры)

var a значение может меняться

const А переназначить нельзя, обозначение нельзя менять

let a можно переназначить

* Математические действия

const a = 11%3 %-остаток от деления

а++ добавить 1 а=+3 добавить 3

а-- отнять 1 а-= отнять 3

а \*\*2 возвести во 2 степень

* Функции

function (){

console.log (“Делаем что-то»)

}

var ourFirstFunction = function(){

console.log(“Привет, мир!»)

}

Вызов функции: ourFirstFunction()

Привет, мир!

function (argument){

Console.log(«Передан аргумент: »+ argument);

}

argument = name = “Аня»

function(argument1, argument2){

console.log(«Первый аргумент: » + argument1);

console.log(“Второй аргумент:» + argument2);

}

Оператор return - возвращает значение

var double = function (number){

return number;

}

double(3)

6

double(5)+double(6);

22

double(double(3));

12

var pickRandomWord = function (words){

return words[Math.floor(Math.random) \*Words.length];

}

var randomWords = [«я», «ты», «они»];

pickRandomWord(randomWords);

Я

* Объекты

1 Доступ через строки (=ключи/свойства)

2 Элемент-значение «ключ-значение» = 1+2

var cat={

“legs”: 3;

“name”: “Гармония»;

«colour”: “черепаховый»;

}

Можно без «»(leds=“legs”)

Литерал - когда значение записывается сразу и целиком

cat[“name”]; - доступ к значению (=cat.name - точечная нотация)

Object.keys(объект) - вывод ключей (выводит массивом)

Если + массив, то доступ -

массив[0]

Массив[0][ключ]

Массив[0].ключ - только для объектов

* Циклы

while(condition){

console.log(«Делаем что-то»)

I++;

}

For(настройка;условие; приращение){

console.log(«Делаем что-то»)

}

* Условия и циклы

Это управляющие конструкции

<script> -js код

Условные конструкции

1. If

2. 2 if … else

if(condition){

console.log(“Делаем что-то»)

}

if(condition){

console.log(“Делаем что-то»)

} else {

console.log(“Делаем что-то»)

}

if(condition1){

console.log(«сделай это, если условие 1 истинно»)

} else if (condition2){

console.log(сделай это, если условие 2 истинно»)

} else if(condition3){

console.log(«сделай это, если условие 3 истинно»)

} else{

console.log(“иначе сделай это»)

}

Ранний выход из функции

var fifthLetter = function (name){

if (name.length<5){

return;

}

return “пятая буква:»+ name(4);

}

var medalForScore = function (scope){

if(scope<3){

return «Бронза»

}

If(scope<7){

return “Серебро»

}

return “Золото»

}

function double(number){

return number \*2;

}

=

var double=function (number){

return number \*2;

}

* Массивы

Свойства - хранят сведения

Методы - изменяют его

Массив.length - длина массива

Массив[массив.length -1] - последний элемент массива

1массив.concat(2массив) - склеивает массивы (можно и 3 и 4)

Массив.indexOf(“элемент массива») - определяет индекс в массиве !!если в массиве нет элемента, вернет -1, а если он встречается несколько раз, вернет индекс, который ближе к началу массива

Массив.join() - объединяет все элементы массива в одну строку !! Что в скобках, такой и разделитель

Math.random() - рандомное число от 0 до 1

Math.random()\*10 - рандомное число от 0 до 10

Math.floor(3,6372) - округляет до целого

Math.floor(Math.random()\*5)- объединяет два предыдущих метода(массив->случайный элемент из массива)

* Висельница

prompt - диалоговое окно между <script>

var a= prompt (“…”) можно что-то написать

confirm-да/нет

var a = confirm (“?”)

if (a){

console.log();

} else {

console.log();

}

const user = {name: ”John”, age: 34}

const {name: userName, age: userAge} = yser

const user {

JohnDoe: {

age: 34,

email: “[John@jmail.com](mailto:John@jmail.com)”

}

}

Const {JohnDoe:{age, email}} = user

Const {JohnDoe:{age: userAge, email: userEmail}} = user

const[a,b] = [1, 2, 3, 4, 5, 6]

console.log(a, b) 1,2

const [a, b, , , c] = [1, 2, 3, 4, 5, 6]

console.log(a, b, c) 1, 2 5

Array.prototype.slice()

const[a, b, …arr] = [1, 2, 3, 4, 5, 6, 7]

console.log(a, b) 1,2

console.log(arr) 3, 4, 5, 6, 7

const profileUpdate = (profileData) =>{

const {name, age, nationality, location} = profileData:

}

=

const profileUpdate = ({name, age, nationality, location}) => { }

Object.freeze() – для того, чтобы данные не менялись

Const myF = function(){

Const a = “value”;

Return a;

}

=

Const myF = () => {

Const a =”value”;

Return a;

}

=

Const myF = () => “value”;

Const doubler = (item) => item \*2;

Doubler(4);

=

Const doubler = item => item \*2;

Const(multiplier) = (item, multi) => item \* multy

Const(greeting)=(name = “Ann”) => “Hi” + name

Function howMany(…args){

Return “You have passed” + arg.length + “arguments”

}

Var arr[6, 3, 89]

Var maximus = Math.max.apply(null, arr);

Или

Const arr = [6, 3, 89];

Const maximus = Math.max(…arr);

Math.max () – 2 аргумента в скобках. Возвращает наибольшее из нуля и более чисел

[arr1]=[…arr2] – добавляет значения массива, а не массив

Const user = {name: “John Doe”, age: 34};

=

Const {name,age} = user;

Const name = user.name

Const age = user.age

Do{

ourArray.push(i);

i++;

} while(i<5)

Const a=parseInt(“007”) 7

a?b:c a- условие, b – если да, с – если нет

function fruitsIs(val)

switch(fruit){

case “apple”:

console.log(“The fruit is an apple”);

break;

case “…”:

…

Break;

}

Switch (num){

Case value1:

Statement1;

Break;

Case value2:

Statemen2;

Break;

…

Default:

defaultStatement;

break;

}

\’ singlu quote

\” double quote

[\\ HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/" HYPERLINK "file://backslash/"backslash](file://\\backslash\)

\n new link

\t tag

\r carriage return

\b backspace

\f formfeed

«строка».slice(1,5)-откуда и до куда вырезать

«Эй, ты!”.toUpperCase()

«ЭЙ, ТЫ!»

«Эй, ты!”.toLowerCase()

«эй, ты!»

a = b.length a=1.length считает количество знаков

a = bcd[1] a = c …[] – говорит, какая буква под номером в [], в джава скрипт счет начинается с 0

const a = “abc”

const b = a[a.length – 1] – если с -, то счет начинается с конца

[“a”, “b”, “c”] [“a”, 1] – массив

[[“a”, 1], [“b”, “c”] – вложенный массив

Const a = [1, 2, 3]

a[0] = 4

замена значений в массиве

.push() – значение в скобках в конец массива

.pop() – забирает элемент из конца массива

,shift() – забирает первый элемент из массива

.unshift() – добавляет элемент в начало массива

Function a(){

}

функция

Function (a,b) {

Console.log(a+b);

}

Функция с переменными, которая считает сумму переменных

Function a (){

Const b = “cc”;

Console.log (b);

}

Выводит на экран

A();

Console.log (b) не выводит

=()=>

== приравнивает и строки и цифры

=== строгое равно (или то, или то)

Typeof – определят тип переменной

i= - не равно

i== - без преображения

>больше <меньше

>=больше или равно <=меньше или равно

a.hasOwnProprty(свойство) – проверяет наличие свойства

булевые операторы:

* && и
* || или
* ! не

Undefined – нет иного значения

Null – чтобы явно обозначить «тут пусто»

firstArray.concat(otherArray) – склеивает массив

.indexOf («элемент») – поиск индекса элемента в массиве

.join() – превращает массив в строку

.join(разделитель) – ставит вместо «разделитель» - разделитель(пробел, запятая или что угодно еще)

switch (fruit) {

case "apple":

console.log("The fruit is an apple");

break;

case "orange":

console.log("The fruit is an orange");

break;

}

Еще раз про свитч, работает как иф элс, всегда строгое равно===

Это все равно, что написать иф фрут === эпл конс лог () элс иф

Так короче и это удобнее

switch (num) {

case value1:

statement1;

break;

case value2:

statement2;

break;

...

default:

defaultStatement;

break;

}

Дефолт будет работать, если случаи не подошли

let result = "";

switch (val) {

case 1:

case 2:

case 3:

result = "1, 2, or 3";

break;

case 4:

result = "4 alone";

}

Чтобы объединить несколько случаев, нужно писать без брэйк, тоже вполне удобно, не нужно много раз писать одно и тоже

switch (val) {

case 1:

answer = "a";

break;

case 2:

answer = "b";

break;

default:

answer = "c";

}

Вот еще про синтаксис

function isEqual(a, b) {

if (a === b) {

return true;

} else {

return false;

}

}

Можно сократить до

function isEqual(a, b) {

return a === b;

}

const ourDog = {

"name": "Camper",

"legs": 4,

"tails": 1,

"friends": ["everything!"],

"bark": "bow-wow"

};

delete ourDog.bark;

позволяет удалить строку из объекта

const ourArray = [];

let i = 0;

do {

ourArray.push(i);

i++;

} while (i < 5);

А тут еще раз про ду вайл. Тут сначала выполняется цикл, а потом происходит проверка

alert -диалоговое окно alert(“…”)

break - выход из цикла

const person = {

name: "Zodiac Hasbro",

age: 56

};

const greeting = `Hello, my name is ${person.name}!

I am ${person.age} years old.`;

console.log(greeting);

это литерал шаблона, теперь не нужно писать через +, и использованы обратные кавычки. В литерале объема может быть несколько переменных

const getMousePosition = (x, y) => ({

x: x,

y: y

});

Равно

const getMousePosition = (x, y) => ({ x, y });

const person = {

name: "Taylor",

sayHello: function() {

return `Hello! My name is ${this.name}.`;

}

};

Равно

const person = {

name: "Taylor",

sayHello() {

return `Hello! My name is ${this.name}.`;

}

};

// Explicit constructor

class SpaceShuttle {

constructor(targetPlanet) {

this.targetPlanet = targetPlanet;

}

takeOff() {

console.log("To " + this.targetPlanet + "!");

}

}

// Implicit constructor

class Rocket {

launch() {

console.log("To the moon!");

}

}

const zeus = new SpaceShuttle('Jupiter');

// prints To Jupiter! in console

zeus.takeOff();

const atlas = new Rocket();

// prints To the moon! in console

atlas.launch();

class Book {

constructor(author) {

this.\_author = author;

}

// getter

get writer() {

return this.\_author;

}

// setter

set writer(updatedAuthor) {

this.\_author = updatedAuthor;

}

}

const novel = new Book('anonymous');

console.log(novel.writer);

novel.writer = 'newAuthor';

console.log(novel.writer);

The console would display the strings anonymous and newAuthor.

рекурсия- это когда функция вызывает сама себя

function pow(x, n) {

let result = 1;

// умножаем result на x n раз в цикле

for (let i = 0; i < n; i++) {

result \*= x;

}

return result;

}

равно

function pow(x, n) {

if (n == 1) {

return x;

} else {

return x \* pow(x, n - 1);

}

}

const a = parseInt("007"); -возвращает целое число, в данном случае 7

a ? b : c

a- условие, b-результат,если тру, с- если фолс. эта запись может быть использована вместо иф элс

function findGreater(a, b) {

if(a > b) {

return "a is greater";

}

else {

return "b is greater or equal";

}

}

если сократить будет так:

function findGreater(a, b) {

return a > b ? "a is greater" : "b is greater or equal";

}

так же это работает при нескольких условиях:

function findGreaterOrEqual(a, b) {

if (a === b) {

return "a and b are equal";

}

else if (a > b) {

return "a is greater";

}

else {

return "b is greater";

}

}

а если сократить:

function findGreaterOrEqual(a, b) {

return (a === b) ? "a and b are equal"

: (a > b) ? "a is greater"

: "b is greater";

}

function findGreaterOrEqual(a, b) {

return (a === b) ? "a and b are equal" : (a > b) ? "a is greater" : "b is greater";

}

можно написать и в одну строчку, но в таком случае все не особо понятно

function countup(n) {

if (n < 1) {

return [];

} else {

const countArray = countup(n - 1);

countArray.push(n);

return countArray;

}

}

console.log(countup(5));

еще один прикол с рекурсивной функцией

теперь про стрелочные функции

const myFunc = function() {

const myVar = "value";

return myVar;

}

дано

const myFunc = () => {

const myVar = "value";

return myVar;

}

сократили

const myFunc = () => "value";

сократили еще

что мы имеем. саму функцию вроде можно будет использовать исключительно в другой функции. тут можно ничего не задавать никакой переменной, а просто сразу ввести в конец значение. можно использовать несколько аргументов (тогда аргументы нужно взять в скобки). То, что в {} - тело функции, там можно что-то написать. но если нужно что-то просто ?сохранить?,то берем их в скобки, чтобы получилось как-то так: ({а тут что-то})

const doubler = (item) => item \* 2;

doubler(4);

const doubler = item => item \* 2;

const multiplier = (item, multi) => item \* multi;

multiplier(4, 2);

можно задать дефолтный параметр, если никак не задать название в эти скобки (как в примере с конс лог (джон)), то на выходе будет дефолтный парамер (анонимус)

const greeting = (name = "Anonymous") => "Hello " + name;

console.log(greeting("John"));

console.log(greeting());

подробнее про ... Когда мы видим "..." в коде, это могут быть как остаточные параметры, так и оператор расширения.

Как отличить их друг от друга:

Если ... располагается в конце списка параметров функции, то это «остаточные параметры». Он собирает остальные неуказанные аргументы и делает из них массив.

Если ... встретился в вызове функции или где-либо ещё, то это «оператор расширения». Он извлекает элементы из массива.

Полезно запомнить:

Остаточные параметры используются, чтобы создавать новые функции с неопределённым числом аргументов.

С помощью оператора расширения можно вставить массив в функцию, которая по умолчанию работает с обычным списком аргументов.

Вместе эти конструкции помогают легко преобразовывать наборы значений в массивы и обратно.

К аргументам функции можно обращаться и по-старому — через псевдомассив arguments.

function howMany(...args) {

return "You have passed " + args.length + " arguments.";

}

console.log(howMany(0, 1, 2));

console.log(howMany("string", null, [1, 2, 3], { }));

в первом случае вернет 3, а во втором 4

не нужно задавать заранее все параметры

вот так можно ?расширить массив? чтобы не писать эпплай, нул и все такое

...арр вернет в скобки весь массив

var arr = [6, 89, 3, 45];

var maximus = Math.max.apply(null, arr);

const arr = [6, 89, 3, 45];

const maximus = Math.max(...arr);

...арр работает как литерал массива и в аргументе функции, так что чтобы передать значение нужно писать так:

const spreaded = [...arr];

а вот так не сработает:

const spreaded = ...arr;

Деструктуризация

Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.

Синтаксис:

let {prop : varName = default, ...} = object

Здесь двоеточие : задаёт отображение свойства prop в переменную varName, а равенство =default задаёт выражение, которое будет использовано, если значение отсутствует (не указано или undefined).

Для массивов имеет значение порядок, поэтому нельзя использовать :, но значение по умолчанию – можно:

let [var1 = default, var2, ...rest] = array

Объявление переменной в начале конструкции не обязательно. Можно использовать и существующие переменные. Однако при деструктуризации объекта может потребоваться обернуть выражение в скобки.

Вложенные объекты и массивы тоже работают, при деструктуризации нужно лишь сохранить ту же структуру, что и исходный объект/массив.

<https://learn.javascript.ru/destructuring> вот тут вроде понятно написано

const user = { name: 'John Doe', age: 34 };

const name = user.name;

const age = user.age;

равно

const { name, age } = user;

в общем это помогает скипнуть писанину с конст нэйм = юзер.нэйм

const user = { name: 'John Doe', age: 34 };

const { name: userName, age: userAge } = user;

так можно одновременно извлечь значение из юзер.нэйм и задать переменную юзерНэйм со значением юзер.нейм (с эйдж работает аналогично)

это же можно провернуть и со вложенными объектами :

const user = {

johnDoe: {

age: 34,

email: 'johnDoe@freeCodeCamp.com'

}

};

const { johnDoe: { age, email }} = user;

const { johnDoe: { age: userAge, email: userEmail }} = user;

а теперь про массивы и их деструктуризацию

const [a, b] = [1, 2, 3, 4, 5, 6];

console.log(a, b);

мы получим 1, 2

const [a, b,,, c] = [1, 2, 3, 4, 5, 6];

console.log(a, b, c);

а если поставим запятых, то получим 1,2,5

const [a, b, ...arr] = [1, 2, 3, 4, 5, 7];

console.log(a, b);

console.log(arr);

такая штука будет работать как а,б,с, где а-первое число в массиве, б - второе,а все остальное приходится на арр, что-то вроде slice

так же можно использовать деструктуризацию в качестве аргументов функции (или что-то такое)

const profileUpdate = (profileData) => {

const { name, age, nationality, location } = profileData;

}

тоже самое, только сокращенно:

const profileUpdate = ({ name, age, nationality, location }) => {

}

тут,чтобы передать, что это объект,а не тело функции нужно взять передаваемый объект в ()

const stats = {

max: 56.78,

standard\_deviation: 4.34,

median: 34.54,

mode: 23.87,

min: -0.75,

average: 35.85

};

const half = (stats) => (stats.max + stats.min) / 2.0;

более "элегантно":

const half = ({ max, min }) => (max + min) / 2.0;

литерал шаблона

объявляется при помощи обратных кавычек (`), тут не нужно экранировать двойные или одинарные кавычки, но обратные, естественно, нужно.

он позволяет написать строку в несколько строк, используя энтр, а не /н, что удобно.

const method = 'concatenation'

const dynamicString = 'This string is using ' + method + '.'

вот это можно заменить на

const method = 'interpolation'

const dynamicString = `This string is using ${method}.`

Интерполировать можно целые выражения, а не только переменные. Например — как здесь, где в строку встраивается результат сложения двух чисел:

const sum = (x, y) => x + y

const x = 5

const y = 100

const string = `The sum of ${x} and ${y} is ${sum(x, y)}.`

console.log(string)

Здесь объявлена функция sum() и константы x и y. После этого в строке используется и функция, и эти константы. Вот как выглядит константа string, выведенная в консоль:

The sum of 5 and 100 is 105.

Этот механизм может оказаться особенно полезным при использовании тернарного оператора, который позволяет проверять условия при формировании строки:

const age = 19

const message = `You can ${age < 21 ? 'not' : ''} view this page`

console.log(message)

const getMousePosition = (x, y) => ({

x: x,

y: y

});

равно

const getMousePosition = (x, y) => ({ x, y });

про this

This — это ключевое слово, используемое в JavaScript, которое имеет особое значение, зависящее от контекста в котором оно применяется.

Когда this используется внутри объекта, это ключевое слово ссылается на сам объект. Рассмотрим пример. Предположим, вы создали объект dog с методами и обратились в одном из его методов к this. Когда this используется внутри этого метода, это ключевое слово олицетворяет объект dog.

var dog = {

name: 'Chester',

breed: 'beagle',

intro: function(){

console.log(this);

}

};

dog.intro();

// в консоль выводится представление объекта dog со всеми его свойствами и методами

// {name: "Chester", breed: "beagle", intro: ƒ}

// breed:"beagle"

// intro:ƒ ()

// name:"Chester"

// \_\_proto\_\_:Object

Применение this во вложенных объектах может создать некоторую путаницу. В подобных ситуациях стоит помнить о том, что ключевое слово this относиться к тому объекту, в методе которого оно используется. Рассмотрим пример.

var obj1 = {

hello: function() {

console.log('Hello world');

return this;

},

obj2: {

breed: 'dog',

speak: function(){

console.log('woof!');

return this;

}

}

};

console.log(obj1);

console.log(obj1.hello()); // выводит 'Hello world' и возвращает obj1

console.log(obj1.obj2);

console.log(obj1.obj2.speak()); // выводит 'woof!' и возвращает obj2

Стрелочные функции ведут себя не так, как обычные функции. Вспомните: при обращении к this в методе объекта, этому ключевому слову соответствует объект, которому принадлежит метод. Однако это не относится к стрелочным функциям. Вместо этого, this в таких функциях относится к глобальному контексту (к объекту window). Рассмотрим следующий код, который можно запустить в консоли браузера.

var objReg = {

hello: function() {

return this;

}

};

var objArrow = {

hello: () => this

};

objReg.hello(); // возвращает, как и ожидается, объект objReg

objArrow.hello(); // возвращает объект Window!

Рассмотрим пример с уже известным нам объектом dog. В качестве метода этого объекта можно назначить функцию chase, объявленную за его пределами. Тут в объекте dog никаких методов не было, до тех пор, пока мы не создали метод foo, которому назначена функция chase. Если теперь вызвать метод dog.foo, то будет вызвана функция chase. При этом ключевое слово this, к которому обращаются в этой функции, указывает на объект dog. А функция chase, при попытке её вызова как самостоятельной функции, будет вести себя неправильно, так как при таком подходе this будет указывать на глобальный объект, в котором нет тех свойств, к которым мы, в этой функции, обращаемся через this.

var dog = {

breed: 'Beagles',

lovesToChase: 'rabbits'

};

function chase() {

console.log(this.breed + ' loves chasing ' + this.lovesToChase + '.');

}

dog.foo = chase;

dog.foo(); // в консоль попадёт Beagles loves chasing rabbits.

chase(); //так эту функцию лучше не вызывать

Ключевое слово this находит применение в функциях-конструкторах, используемых для создания объектов, так как оно позволяет, универсальным образом, работать со множеством объектов, создаваемых с помощью такой функции. В JavaScript есть и стандартные функции-конструкторы, с помощью которых, например, можно создавать объекты типа Number или String. Подобные функции, определяемые программистом самостоятельно, позволяют ему создавать объекты, состав свойств и методов которых задаётся им самим.

Как вы уже поняли, мне нравятся собаки, поэтому опишем функцию-конструктор для создания объектов типа Dog, содержащих некоторые свойства и методы.

function Dog(breed, name, friends){

this.breed = breed;

this.name = name;

this.friends = friends;

this.intro = function() {

console.log(`Hi, my name is ${this.name} and I’m a ${this.breed}`);

return this;

};

Когда функцию-конструктор вызывают с использованием ключевого слова new, this в ней указывает на новый объект, который, с помощью конструктора, снабжают свойствами и методами.

Вот как можно работать со стандартными конструкторами JavaScript.

var str = new String('Hello world');

/\*\*\*\*\*\*\*

Строки можно создавать так, но лучше этого не делать, используя подход, применённый при объявлении переменной str2 ниже. Одна из причин подобной рекомендации заключается в том, что в JavaScript строки удобно создавать, пользуясь строковыми литералами, когда строкой считается всё, включённое в двойные или одинарные кавычки. То же самое касается и других примитивных значений. Стоит отметить, что мне, на практике, не встречалась ситуация, когда надо было бы использовать конструкторы для создания значений примитивных типов.

\*\*\*\*\*\*\*/

var str2 = 'Hello world';

// когда строка объявлена так, система, всё равно, позволяет работать с ней как с объектом

Теперь поработаем с только что созданной функцией-конструктором Dog.

// Создадим новый экземпляр объекта типа Dog

var chester = new Dog('beagle', 'Chester', ['Gracie', 'Josey', 'Barkley']);

chester.intro(); // выводит Hi, my name is Chester and I'm a beagle

console.log(chester); // выводит Dog {breed: "beagle", name: "Chester", friends: Array(3), intro: ƒ}

Вот ещё один пример использования функций-конструкторов.

var City = function(city, state) {

this.city = city || "Phoenix";

this.state = state || "AZ";

this.sentence = function() {

console.log(`I live in ${this.city}, ${this.state}.`);

};

};

var phoenix = new City(); // используем параметры по умолчанию

console.log(phoenix); // выводит в консоль строковое представление объекта

phoenix.sentence(); // выводит I live in Phoenix, AZ.

var spokane = new City('Spokane', 'WA');

console.log(spokane); // выводит сам объект

spokane.sentence(); // выводит I live in Spokane, WA.

При вызове функции-конструктора с использованием ключевого слова new ключевое слово this указывает на новый объект, который, после некоторой работы над ним, будет возвращён из этой функции. Ключевое слово this в данной ситуации весьма важно. Почему? Всё дело в том, что с его помощью можно, используя единственную функцию-конструктор, создавать множество однотипных объектов.

Это позволяет нам масштабировать приложение и сокращать дублирование кода. Для того чтобы понять важность этого механизма, подумайте о том, как устроены учётные записи в социальных сетях. Каждая учётная запись может представлять собой экземпляр объекта, создаваемый с помощью функции-конструктора Friend. Каждый такой объект можно заполнять уникальными данными о пользователе. Рассмотрим следующий код.

// Функция-конструктор

var Friend = function(name, password, interests, job){

this.fullName = name;

this.password = password;

this.interests = interests;

this.job = job;

};

function sayHello(){

// раскомментируйте следующую строчку, чтобы узнать, на что указывает this

// console.log(this);

return `Hi, my name is ${this.fullName} and I'm a ${this.job}. Let's be friends!`;

}

// Мы можем создать один или несколько экземпляров объекта типа Friend, используя ключевое слово new

var john = new Friend('John Smith', 'badpassword', ['hiking', 'biking', 'skiing'], 'teacher');

console.log(john);

// Назначим функцию ключу greeting объекта john

john.greeting = sayHello;

// Вызовем новый метод объекта

console.log( john.greeting() );

// Помните о том, что sayHello() не стоит вызывать как обычную функцию

console.log( sayHello() ) ;

классы

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.

Например:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

// Использование:

let user = new User("Иван");

user.sayHi();

Когда вызывается new User("Иван"):

Создаётся новый объект.

constructor запускается с заданным аргументом и сохраняет его в this.name.

Затем можно вызывать на объекте методы, такие как user.sayHi().

Методы в классе не разделяются запятой

Частая ошибка начинающих разработчиков – ставить запятую между методами класса, что приводит к синтаксической ошибке.

Синтаксис классов отличается от литералов объектов, не путайте их. Внутри классов запятые не требуются

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.

Базовый синтаксис для классов выглядит так:

class MyClass {

prop = value; // свойство

constructor(...) { // конструктор

// ...

}

method(...) {} // метод

get something(...) {} // геттер

set something(...) {} // сеттер

[Symbol.iterator]() {} // метод с вычисляемым именем (здесь - символом)

// ...

}

MyClass технически является функцией (той, которую мы определяем как constructor), в то время как методы, геттеры и сеттеры записываются в MyClass.prototype.

// Explicit constructor

class SpaceShuttle {

constructor(targetPlanet) {

this.targetPlanet = targetPlanet;

}

takeOff() {

console.log("To " + this.targetPlanet + "!");

}

}

// Implicit constructor

class Rocket {

launch() {

console.log("To the moon!");

}

}

const zeus = new SpaceShuttle('Jupiter');

// prints To Jupiter! in console

zeus.takeOff();

const atlas = new Rocket();

// prints To the moon! in console

atlas.launch();

какая-то бяка(

class Book {

constructor(author) {

this.\_author = author;

}

// getter

get writer() {

return this.\_author;

}

// setter

set writer(updatedAuthor) {

this.\_author = updatedAuthor;

}

}

const novel = new Book('anonymous');

console.log(novel.writer);

novel.writer = 'newAuthor';

console.log(novel.writer);

The console would display the strings anonymous and newAuthor.

class Thermostat {

constructor(fahrenheit) {

this.\_fahrenheit = fahrenheit;

}

get temperature() {

return (5 / 9) \* (this.\_fahrenheit - 32);

}

set temperature(celsius) {

return this.\_fahrenheit = (celsius \* 9.0) / 5 + 32;

}

}

скрипты

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

записывается так. в типе что-то пишется, а src - ссылка, вроде

экспорт части кода

export const add = (x, y) => {

return x + y;

}

const add = (x, y) => {

return x + y;

}

export { add };

export { add, subtract }; - а это эксорт двух функций

а чтобы что-то импортировать нужно написать так:

import { add } from './math\_functions.js';

где указано что, откуда. из этого файла копируется только одна функция, а все остальное игнорируется

./показывает, что файл, который указан,необходимо искать в той же папке, где и этот файл хранится

При таком использовании импорта требуются относительный путь к файлу (./) и расширение файла (.js).

import { add, subtract } from './math\_functions.js'; - можно импортировать несолько файлов

import \* as myMathModule from "./math\_functions.js" - чтобы импортировать все из фукции, при этом myMathModule рандомное название,а все функции, что были написаны там, перенесутся сюда

Геттеры и сеттеры

Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:

let obj = {

get propName() {

// геттер, срабатывает при чтении obj.propName

},

set propName(value) {

// сеттер, срабатывает при записи obj.propName = value

}

};

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение присваивается.

Например, у нас есть объект user со свойствами name и surname:

let user = {

name: "John",

surname: "Smith"

};

А теперь добавим свойство объекта fullName для полного имени, которое в нашем случае "John Smith". Само собой, мы не хотим дублировать уже имеющуюся информацию, так что реализуем его при помощи аксессора:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

}

};

alert(user.fullName); // John Smith

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. Мы не вызываем user.fullName как функцию, а читаем как обычное свойство: геттер выполнит всю работу за кулисами.

На данный момент у fullName есть только геттер. Если мы попытаемся назначить user.fullName=, произойдёт ошибка:

let user = {

get fullName() {

return `...`;

}

};

user.fullName = "Тест"; // Ошибка (у свойства есть только геттер)

Давайте исправим это, добавив сеттер для user.fullName:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

},

set fullName(value) {

[this.name, this.surname] = value.split(" ");

}

};

// set fullName запустится с данным значением

user.fullName = "Alice Cooper";

alert(user.name); // Alice

alert(user.surname); // Cooper

В итоге мы получили «виртуальное» свойство fullName. Его можно прочитать и измени

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними.

Например, если мы хотим запретить устанавливать короткое имя для user, мы можем использовать сеттер name для проверки, а само значение хранить в отдельном свойстве \_name:

let user = {

get name() {

return this.\_name;

},

set name(value) {

if (value.length < 4) {

alert("Имя слишком короткое, должно быть более 4 символов");

return;

}

this.\_name = value;

}

};

user.name = "Pete";

alert(user.name); // Pete

user.name = ""; // Имя слишком короткое...

Таким образом, само имя хранится в \_name, доступ к которому производится через геттер и сеттер.

Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью user.\_name, но существует широко известное соглашение о том, что свойства, которые начинаются с символа "\_", являются внутренними, и к ним не следует обращаться из-за пределов объекта

Экспорт по дефолту:

export default function add(x, y) {

return x + y;

} - это именованая функция

export default function(x, y) {

return x + y;

} - анонимная функция

для того, чтобы импортировать дефолтное значение нужно написать так:

import add from "./math\_functions.js"; - тут не нужно что-либо брать в{}

обещния в js

Обещание в JavaScript — это именно то, на что оно похоже — вы используете его, чтобы пообещать что-то сделать, обычно асинхронно. Когда задача будет выполнена, вы либо выполните свое обещание, либо не сделаете этого. Promise — это функция-конструктор, поэтому для его создания необходимо использовать ключевое слово new. В качестве аргумента он принимает функцию с двумя параметрами: решить и отвергнуть. Это методы, используемые для определения результата обещания. Синтаксис выглядит следующим образом:

const myPromise = new Promise((resolve, reject) => {

});

Обещание имеет три состояния: pending(ожидание), fulfilled,(выполнение) и rejected(отклонение). Обещание, которое вы создали в последнем задании, навсегда застряло в состоянии ожидания, поскольку вы не добавили способ выполнения обещания. Для этого используются параметры разрешения и отклонения, заданные аргументу обещания. «Решимость» используется, когда вы хотите, чтобы ваше обещание было успешным, и reject используется, когда вы хотите, чтобы он потерпел неудачу. Это методы, которые принимают аргумент, как показано ниже.

const myPromise = new Promise((resolve, reject) => {

if(condition here) {

resolve("Promise was fulfilled");

} else {

reject("Promise was rejected");

}

});

Промисы наиболее полезны, когда в вашем коде есть процесс, который занимает неизвестное количество времени (т. е. что-то асинхронное), часто это запрос к серверу. Когда вы делаете запрос к серверу, это занимает некоторое время, и после его завершения вы обычно хотите что-то сделать с ответом сервера.

Этого можно добиться, используя метод then. Метод then выполняется сразу после того, как ваше обещание будет выполнено с разрешением. Вот пример:

myPromise.then(result => {

});

catch — это метод, используемый, когда ваше обещание было отклонено. Он выполняется сразу после вызова метода отклонения обещания. Вот синтаксис:

myPromise.catch(error => {

});

error (ошибка) — это аргумент, передаваемый методу отклонения.

**CSS**

<style> - расположен в пределах <head>, позволяет контролировать стиль создаваемого элемента

element {

property: value;

}

Расположено в нем

 h1{

        text-align:center

      }

Теперь все, что в h1 расположено в центре

можно написать h1,h2,p, если все в одном стиле, чтобы не писать одно и тоже для каждого отдельно

так же для удобства можно создать отдельный файл со стилями, чтобы не перегружать код, а в самом коде оставить ссылку

в отдельном файле не нужно писать <style>

  <link rel="stylesheet" href="styles.css">

Ссылка, которая расположена в <head>!

rel="stylesheet"  - определяет, что ссылка содержит информацию, как стилизовать страницу

<meta name="viewport" content="width=device-width, initial-scale=1.0" /> расположено в <head>, чтобы на телефоне и на компе выглядело одинаково

body{

  background-color: brown;

} создает коричневую заливку, а burlywood кофейную, нежную

<div> - тег для дизайна

<div id=”menu”> - создает ссылку

#example-id {

width: 250px;

}

#menu{

  width: 300px;

}

Для того, чтобы тег диф не занимал всю страницы, а определенную часть страницы – в ширину

Можно использовать вместо пикселей проценты, аля 80% вместо 300 рх

/\* comment here \*/ для того, чтобы оставлять комментарии

 margin-left: auto;

  margin-right: auto;

меняет по горизонтали, чтобы было по центру все

.class-name {

styles

}

.menu {

  width: 80%;

  background-color: burlywood;

  margin-left: auto;

  margin-right: auto;

}

Классовый-селектор, теперь не через айди (<div class="menu">) Id меняем на class

body {

  background-image: url(https://cdn.freecodecamp.org/curriculum/css-cafe/beans.jpg);

}

Чтобы добавить фото на фон

<article> - еще один вариант элементов, обычно содержат несколько элементов, имеющих связанную информацию (<p> может быть внутри)

.flavor{

  text-align: left;

}

Текст влево

  <h2>Coffee</h2>

          <article class="item">

            <p class="flavor">French Vanilla</p>

            <p class="price">3.00</p>

          </article>

Чтобы было в 1 строчку, а не разбросано

.item p{

  display: inline-block;

}

Все элементы вложенные в р ведут себя как встроенные элементы (что-то я запуталась)

.flavor {

  text-align: left;

  width: 49%;

}

.price {

  text-align: right;

  width: 49%;

}

Так можно сделать, чтобы цена и кофе были с краев

.menu {

  width: 80%;

  background-color: burlywood;

  margin-left: auto;

  margin-right: auto;

  padding-left: 20px;

  padding-right: 20px;

}

 padding-left: 20px;

  padding-right: 20px;

тут создает пространство молочным коричневым и началом текста

 padding-top: 20px;- отступ сверху

padding-bottom: 20px; - отступ снизу

если все стороны одинаковые, то чтобы не писать

padding-left: 20px;

  padding-right: 20px;

 padding-left: 20px;

  padding-right: 20px;

можно просто написать padding: 20px;

 max-width:500px;– для того, чтобы на очень большом экране не было так, что цена и кофе слишком далеко

body {

  background-image: url(https://cdn.freecodecamp.org/curriculum/css-cafe/beans.jpg);

  font-family: sans-serif;

}

Тут font-family: sans-serif; задает шрифт для всего что в боди

h1, h2 {

  font-family: Impact, serif;

}

Так можно задать запасной шрифт(на всякий случай, мало ле)

h1{

  font-size: 40px;

}

Задает размер шрифта

 <hr> - для разделения !!самозакрывающийся

hr{

  height: 3px;

}

Изменяет толщину линии

hr {

  height: 3px;

  background-color: brown;

  border-color: brown

}

А так можно изменить цвет и границу заливки этой линии. По умолчанию толщина границы 1 пиксель

.item p {

  display: inline-block;

  margin-top: 5px;

  margin-bottom: 5px;

}

Настроить отступы между позициями в меню (межстрочный интервал что ли)

a{

  color: black;

}

Так можно изменить цвет ссылки, чтобы она не меняла цвет вне зависимости нажимали на нее или нет

a:visited{

  color: grey;

}

А вот так можно поменять цвет ссылки, если на нее нажали

a:hover{

  color: brown;

}

Теперь, если навести мышкой на ссылку, она станет коричневой, а не серой

a:active{

  color:white;

}

Теперь после того, как на ссылку нажали, она будет белого цвета, а не серого

**HTML**

 <!DOCTYPE html>- тэг в самом начале

<html> - 2 фраза, тут можно написать <html lang="en">

чтобы обозначить язык(английский)

<head> - это голова, тут можно дать название веб-страницы (будет написано на вкладке в браузере) при помощи <title>Cafe Menu </title>

<meta charset="utf-8"> расположено в <head> дает информацию браузеру, как кодировать элементы на этой странице !!самозакрывающийся

<body> идет после <head>, тут основной код

<main> расположен в <body>, несет основную информацию из страницы

<h1> тег, который делает большие буквы, 2 3 4 5 6 все меньше и меньше

<p> параграф, дает возможность писать текст буквами поменьше с новой строки, перед ним при написании кода нужно делать 2 пробела

<img> - для картинок, !!самозакрывающийся

src – атрибут, который указывает адрес, например для фотки: <img src-“https…”

alt- атрибут, который отображает текст, если, например, картинка не привязывается: <img src=”cat.jpg” alt=”A cat”>”

<a href=”ссылка”>тут может быть текст ссылки</a> - переход на другую страницу, а чтобы открыть ссылку в новой вкладке, нужно написать <a target=\_blank href=”ссылка”></a>

<section> - отделяет, создает автономный раздел

<ul> <ol>

<li>milk</li>

<li>cheese</li>

</ul> </ol>

Ul – создает список, когда перед пунктом будет стоять точка, а ol – список, когда перед пунктом цифра (1,2,3)

<figure> - объединяет надпись с картинкой

<figcaption> - создает текст, который описывает картинку (под картинкой)

<figure>

<figcaption>

</figcaption>

</figure>

<em>- подчеркивает текст <strong>-делает текст жирным

<form> - веб-форма

<action> - куда отправить данные формы, например: <form action="ссылка”> Этот атрибут определяет, куда отправляются данные. Его значение должно быть действительным URL. Если этот атрибут не указан, данные будут отправлены на URL-адрес страницы, содержащей форму.

<input> - собирает данные !!самозакрывающийся

<type> - тип сбора, например <input type=”text”>

Чтобы добавить комментарий:<!--комментарий -->

<name> - чтобы получить доступ к данным из месторасположения: <input type=”text” name=”email”>

Placeholder – подсказка, что ввести

Required – проверяет, заполнено ли поле (можно не задавать значение)

<button> - чтобы кликать=) если не ввести значение между <>, отправиться само собой

Синтаксис type: <input type=”text” name=”username”>

<input type=”submit” value=”отправить”>

<button type=”submit”>Submit<button> пример из фриКодКэмп

Radio – переключатель

Label – связывает с input- элементами

<label><input type=”radio”>cat</label> - позволяет щелкнуть на cat

Id – атрибут идентификации

<fieldset> - группирует связанные входные данные + делает новую строку, создает рамку

<legend>- элемент, действует как подпись к содержимому элемента fieldset – создает заголовок для этой группы, текст вписан в рамке

Type=”checkbox”- флажок, если есть несколько вариантов

Привязка к флажку через for:

<input id=”loving” type=”checkbox”>

<label for=”loving”>Loving</label>

Chtckbox – можно добать input, тогда будет вариант по умолчанию

<footer> - нижний колонтикулум

Основы html

Гипертекстовый - фрагменты текста свойственны гиперссылкам (ссылки на другие объекты)

Язык разметки - способ встраивать в текст дополнительную информацию

HTML - страница состоит из элементов, который начинается с открывания тела, заканчивается закрыванием

Блоковые элементы:

<h1> - заголовок 1 уровня

<p> - создает параграф

Строчные элементы:

<em> - курсив

<strong> - жирный шрифт

<!DOCTYPE html> - это “HTML-документ»

<html>- закрывается к конце, должен быть в каждом коде

<head> инфа про док

<title> текст на закладке в браузере

<body> - то, что будет в браузере

<a> - ссылка на интернет-адрес

<p><a href=“ссылка»>Нажмите сюда</p>

Можно добавить атрибут title

<a htef=“ссылка» title=“fhj:комиксы!»>Нажмите сюда</a>

Теперь при наведении курсора на ссылку будет виден текст (комиксы)