**Паттерн «Модуль»**

Модуль — это самостоятельный фрагмент кода, который можно изменять, не затрагивая другой код проекта. Модули, кроме того, позволяют избегать такого явления, как загрязнение областей видимости, благодаря тому, что они создают отдельные области видимости для объявляемых в них переменных. Модули, написанные для одного проекта, можно повторно использовать в других проектах, в том случае, если их механизмы универсальны и не привязаны к особенностям конкретного проекта.

## Паттерн «Фабрика»

Паттерн «Фабрика» (Factory) использует для создания объектов так называемые «фабричные методы». При этом не требуется указывать классы или функции-конструкторы, которые применяются для создания объектов.

## Паттерн «Синглтон»

Паттерн «Синглтон» или «Одиночка» (Singleton) представляет собой объект, который может существовать лишь в единственном экземпляре. В рамках применения этого паттерна новый экземпляр некоего класса создаётся в том случае, если он пока не создан. Если же экземпляр класса уже существует, то, при попытке обращения к конструктору, возвращается ссылка на соответствующий объект. Последующие вызовы конструктора всегда будут возвращать тот же самый объект.

**Современный стандарт, "use strict»**

Очень долго язык JavaScript развивался без потери совместимости. Новые возможности добавлялись в язык, но старые – никогда не менялись, чтобы не «сломать» уже существующие HTML/JS-страницы с их использованием.

Однако, это привело к тому, что любая ошибка в дизайне языка становилась «вмороженной» в него навсегда.

Так было до появления стандарта ECMAScript 5 (ES5), который одновременно добавил новые возможности и внёс в язык ряд исправлений, которые могут привести к тому, что старый код, который был написан до его появления, перестанет работать.

Чтобы этого не случилось, решили, что по умолчанию эти опасные изменения будут выключены, и код будет работать по-старому. А для того, чтобы перевести код в режим полного соответствия современному стандарту, нужно указать специальную директиву use strict.

"Use strict» специальная деректива, которая переводит код в режим полного соответствия современному стандарту.

этот код будет работать по современному стандарту ES5

**Порядок исполнения скриптов. Async и defer подключение скриптов: для чего и какие отличия**

**Async**

<script src="1.js" async></script>

<script src="2.js" async></script>

Первым выполнится тот, который быстрее загрузится

**Defer**

<script src="1.js" async></script>

<script src="2.js" async></script>

Выполнятся последовательно. И defer  выполниться только тогда, когда весь документ закончится.

Оба методы асинхронны и не мешают отрисовки страницы.

**Знание о CRP (Critical Rendering Path) невероятно полезно для понимания того, как улучшить производительность сайта.**

Существует 6 этапов CRP:

построение DOM-дерева,

построение CSSOM-дерева,

запуск JavaScript,

создание Render-дерева,

генерация раскладки,

отрисовка

**Callback**

Любая функция, которая передается как аргумент, называется callback-функцией;

коллбэк – это функция, которая должна быть выполнена после того, как другая функция завершила выполнение (отсюда и название: callback – функция обратного вызова).

o = {}; литеральный способ.

// эквивалентно этому:

o = Object.create(Object.prototype);

**События (как повесить обработчик, делегирование, фазы)**

Обычно события становятся в очередь и обрабатываются в порядке поступления, асинхронно, независимо друг от друга.

Синхронными являются вложенные события, инициированные из кода.

Чтобы сделать событие гарантированно асинхронным, используется вызов через setTimeout(func, 0)

**Повесить обработчик:**

**1.Использование атрибута HTML**

<input value="Нажми меня" onclick="alert('Клик!')" type=«button">

**2. Использование свойства DOM-объекта**

<input id="elem" type="button" value="Нажми меня" />

<script>

elem.onclick = function() {

alert( 'Спасибо' );

};

</script>

**3. addEventListener и removeEventListener**

**StopPropagation**

Для остановки всплытия нужно вызвать метод event.stopPropagation().

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены. То есть, stopPropagation препятствует продвижению события дальше, но на текущем элементе все обработчики отработают. Для того, чтобы полностью остановить обработку, современные браузеры поддерживают метод **event.stopImmediatePropagation()**. Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Методы addEventListener и removeEventListener являются современным способом назначить или удалить обработчик, и при этом позволяют использовать сколько угодно любых обработчиков.

Назначение обработчика осуществляется вызовом addEventListener с тремя аргументами:

element.addEventListener(event, handler[, phase]);

event

Имя события, например click

handler

Ссылка на функцию, которую надо поставить обработчиком.

phase

Необязательный аргумент, «фаза», на которой обработчик должен сработать. True - cрабатывает на погружение, False- на всплытие.

Удаление обработчика осуществляется вызовом removeEventListener:

// передать те же аргументы, что были у addEventListener

element.removeEventListener(event, handler[, phase]);

**Делегирование**

Всплытие событий позволяет реализовать один из самых важных приёмов разработки – делегирование.

Он заключается в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому – мы ставим один обработчик на их общего предка. Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

Пожалуй, это один из самых полезных приёмов для работы с DOM. Он отлично подходит, если есть много элементов, обработка которых очень схожа.

Алгоритм:

Вешаем обработчик на контейнер.

В обработчике: получаем event.target.

В обработчике: если event.target или один из его родителей в контейнере (this) – интересующий нас элемент – обработать его.

event.target.closest

**Фазы**

**Погружение - захват- всплытие.**

Алгоритм:

При наступлении события – элемент, на котором оно произошло, помечается как «целевой» (event.target).

Далее событие сначала двигается вниз от корня документа к event.target, по пути вызывая обработчики, поставленные через addEventListener(...., true).

Далее событие двигается от event.target вверх к корню документа, по пути вызывая обработчики, поставленные через on\* и addEventListener(...., false).

Каждый обработчик имеет доступ к свойствам события:

event.target – самый глубокий элемент, на котором произошло событие.

event.currentTarget (=this) – элемент, на котором в данный момент сработал обработчик (до которого «доплыло» событие).

event.eventPhase – на какой фазе он сработал (погружение =1, всплытие = 3).

Любой обработчик может остановить событие вызовом event.stopPropagation(), но делать это не рекомендуется, так как в дальнейшем это событие может понадобиться, иногда для самых неожиданных вещей.

**Event loop, таски, микротаски**

Задачи исполняются по порядку и браузер может рендерить в промежутках между ними

Микрозадачи исполняются по порядку и исполняются:

после каждого колбека, если только это не часть выполнения какого-то другого сценария

в конце каждой задачи

С начало выполнятся микротаски, затем таски.

Promise - микротаски т.к имеют свои таски

console.log('script start');

setTimeout(function() {

console.log('setTimeout');

}, 0);

Promise.resolve().then(function() {

console.log('promise1');

}).then(function() {

console.log('promise2');

});

console.log('script end’);

script start

script end

promise1

promise2

setTimeout

**AJAX**

AJAX (аббревиатура от «Asynchronous Javascript And Xml») – технология обращения к серверу без перезагрузки страницы.

COMET – общий термин, описывающий различные техники получения данных по инициативе сервера.

Можно сказать, что AJAX – это «отправил запрос – получил результат», а COMET – это «непрерывный канал, по которому приходят данные».

**Fetch -** нельзя отменить

Совершает AJAX запросы, возвращает промис

let promise = fetch(url[, options]);

url – URL, на который сделать запрос,

options – необязательный объект с настройками запроса.

Свойства options:

method – метод запроса,

headers – заголовки запроса (объект),

body – тело запроса: FormData, Blob, строка и т.п.

mode – одно из: «same-origin», «no-cors», «cors», указывает, в каком режиме кросс-доменности предполагается делать запрос.

credentials – одно из: «omit», «same-origin», «include», указывает, пересылать ли куки и заголовки авторизации вместе с запросом.

cache – одно из «default», «no-store», «reload», «no-cache», «force-cache», «only-if-cached», указывает, как кешировать запрос.

redirect – можно поставить «follow» для обычного поведения при коде 30x (следовать редиректу) или «error» для интерпретации редиректа как ошибки.

**JSON**

fetch('/users.json')

.then(function(response) {

return response.json()

}).then(function(json) {

console.log('parsed json', json)

}).catch(function(ex) {

console.log('parsing failed', ex)

})

**Post JSON**

fetch('/users', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

name: 'Hubot',

login: 'hubot',

})

})

**call, bind, apply (что делают и чем отличаются)**

Методы явного указания this.

Вызов func.call(context, a, b...) – то же, что обычный вызов func(a, b...), но с явно указанным this(=context).

func.apply(context, args) // this = context (явная передача)

func.call(context, arg1, arg2, …)

**bind не похож на call/apply**

Методы bind и call/apply близки по синтаксису, но есть важнейшее отличие.

Методы call/apply вызывают функцию с заданным контекстом и аргументами.

А bind не вызывает функцию. Он только возвращает «обёртку», которую мы можем вызвать позже, и которая передаст вызов в исходную функцию, с привязанным контекстом.

**Каррирование** – создание новой функции путём фиксирования аргументов существующей.

function mul(a, b) {

return a \* b;

};

var double = mul.bind(null, 2)

alert( double(3) ); // = mul(2, 3) = 6

Вызов bind часто используют для привязки функции к контексту, чтобы затем присвоить её в обычную переменную и вызывать уже без явного указания объекта.

Вызов bind также позволяет фиксировать первые аргументы функции («каррировать» её), и таким образом из общей функции получить её «частные» варианты – чтобы использовать их многократно без повтора одних и тех же аргументов каждый раз.

Чтобы новым объектам автоматически ставить прототип, конструктору ставится свойство prototype.

Метод **Object.create()** создаёт новый объект с указанными объектом прототипа и свойствами. Object.create(proto[, propertiesObject])

Object.create = function create(protoObj){

function F() {};

F.prototype = protoObj;

return new F();

}

**Прототипы. Придумать и реализовать примеры: наследования, вызов родительского конструктора в дочернем. Где хранится прототип у объекта(proto)? В чем отличие prototype от \_\_proto\_\_?**

Прототип хранится в свойстве \_\_proto\_\_;

**Prototype просто свойство указывающие на прототип**

При создании объекта через new, в его прототип \_\_proto\_\_ записывается ссылка из prototype функции-конструктора.

Свойство prototype указывает на прототип, а \_\_proto\_\_ указывает на объект прототипа.

Проще говоря: "вся хрень лежит в объекте \_\_proto\_\_, который создан из prototype". Пнятненько?

Для произвольной функции – назовём её Person, верно следующее:

Прототип \_\_proto\_\_ новых объектов, создаваемых через new Person, можно задавать при помощи свойства Person.prototype.

Значением Person.prototype по умолчанию является объект с единственным свойством constructor, содержащим ссылку на Person. Его можно использовать, чтобы из самого объекта получить функцию, которая его создала. Однако, JavaScript никак не поддерживает корректность этого свойства, поэтому программист может его изменить или удалить.

Современный метод Object.create(proto) можно эмулировать при помощи prototype, если хочется, чтобы он работал в IE8-.

function Node(name, attributes = {}) {

this.name = name;

this.attributes = attributes;

}

Node.prototype.getAttributesAsLine = function () {

return Object.keys(this.attributes).reduce((acc, key) => `${acc} ${key}="${this.attributes[key]}"`, '');

}

function PairedTag(name, attributes, body = '', children = []) {

Node.apply(this, [name, attributes]);

this.body = body;

this.children = children;

}

PairedTag.prototype = Object.create(Node.prototype);

PairedTag.prototype.toString = function () {

const value = this.children.length > 0 ? this.children.map(child => child.toString()).join('') : this.body;

return `<${this.name}${this.getAttributesAsLine()}>${value}</${this.name}>`;

};

function SingleTag(name, attributes = {}) {

Node.apply(this, [name, attributes]);

}

SingleTag.prototype = Object.create(Node.prototype);

SingleTag.prototype.toString = function () {

return `<${this.name}${this.getAttributesAsLine()}>`

}

const signleTag = new SingleTag('img', {class: 'img', src: './picture.img'});

console.log(signleTag.toString());

const pairedTag = new PairedTag('div', { class: 'container'}, 'container');

console.log(pairedTag.toString());

**Промисы**

Promis.all([p1, p2, p3]);

Promise.resolve(value);

Promise.reject(err);

Promise.**finally**(() => {}) - используется если нужно произвести какие то действия вне зависимости от результата

Promise.race([promise1, promise2]) - выполняет первое выполненное или откл событие

httpGet("/article/promise/user.json")

.then(

response => alert(`Fulfilled: ${response}`),

error => alert(`Rejected: ${error}`)

);

**Асинхронность**

Возможность выполнять отложенные задачи.

Для лучшего понимания асинхронности неплохо разобраться с тем, как устроен рантайм (браузер или Node.js) JavaScript. JavaScript изначально появился в браузерах, и к нему предъявлялись особые требования, из-за которых он кардинально отличается от остальных языков программирования. Браузер работает по так называемой событийной модели. Он загружает страницу и ждёт действий от пользователя: клики, набор текста или движение мышкой. А код, загруженный на страницу, реагирует на эти события.

Такая организация взаимодействия невозможна в синхронном коде, у которого есть понятия "запуск" и "завершение" работы. Код в браузере не может завершиться совсем, он проходит стадию инициализации, а затем ждёт событий для реакции на них. Технически это выглядит, как колбек, который соединён с определённым типом события. Когда событие срабатывает, то колбек вызывается.

<!-- Если нажать на кнопку, то выведется модальное окно с текстом "Бум!" -->

<button onclick="alert('Бум!')"></button>

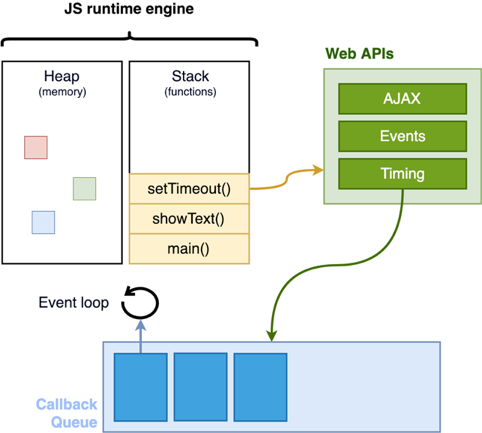
Организация асинхронного взаимодействия требует наличия событийного цикла (Event Loop). Он может быть реализован как на уровне языка в виде библиотеки, так и на уровне рантайма, как в случае с JavaScript. Упрощённо цикл событий можно представить себе так:

while (queue.waitForMessage()) {

queue.processNextMessage();

}

Как только появляется новое событие, оно начинает обрабатываться (тем самым обработчиком, который мы повесили на это событие). Время обработки события может быть довольно большим, но это не значит, что в этот момент браузер зависает и не даёт пользователю работать. Нет, напротив, благодаря тому, что код асинхронный, события могут (и будут) накапливаться в очереди (queue). Именно поэтому в коде выше сообщение проверяется в переменной с именем queue.

****

**Функциональное программирование**

**Какие основные понятия у ФП?**

Неизменяемость, чистые функции, функции высшего порядка, каррирование

Что такое чистая функция? Будет ли она чистой если в ней будет вызов console.log или обращение к пакету Math?(нет)

**Чистая функция** :

Функция, вызываемая от одних и тех же аргументов, всегда возвращает одинаковое значение.

Во время выполнения функции не возникают побочные эффекты.

Примеры побочных эффектов:

Видоизменение входных параметров

console.log

HTTP вызовы (AJAX/fetch)

Изменение в файловой системе

Запросы DOM

К**аррирование** – создание новой функции путём фиксирования аргументов существующей.

**Объе́ктно-ориенти́рованное программи́рование (ООП) —** методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

**Абстракция** означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.

**Инкапсуляция** означает ограничение доступа к данным и возможностям их изменения.(сокрытие реализации от пользовательского кода, пользователю важно что делает метод, а не почему**).**

**Наследование —** механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.

**Полиморфизм —** это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат.

В общем смысле, концепцией полиморфизма является идея "один интерфейс, множество форм».

**Solid**

**Принцип единственной ответственности (The Single Responsibility Principle)**

Каждый класс должен иметь одну ответственность (выполняет лишь одну задачу).

**Принцип открытости/закрытости (The Open Closed Principle)**

«программные сущности … должны быть открыты для расширения, но закрыты для модификации.»

**Принцип подстановки Барбары Лисков (The Liskov Substitution Principle)**

Методы, использующие некий тип, должны иметь возможность использовать его подтипы, не зная об этом.

**Принцип разделения интерфейса (The Interface Segregation Principle)**

«много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.»

**Принцип инверсии зависимостей (The Dependency Inversion Principle)**

«зависимости должны строится относительно абстракций, а не деталей».

**Big O. Сложность алгоритмов.**

По сути, это удобный способ быстрой и точной оценки характеристик работы алгоритма. Будь то, как быстро в худшем случае или в среднем случае выполняется алгоритм. Сколько места в худшем или среднем случае оно использует.

O(ƒ(n)) – (Big-O) – верхняя граница, «не хуже чем»

o(ƒ(n)) – (Little-o) – верхняя граница, «лучше чем»

Ω(ƒ(n)) – (Omega) – нижняя граница, «не лучше чем»

Θ(ƒ(n)) – (Theta) – точная оценка.

Простые решения можно оценивать по количеству вложенных циклов

Существуют «стандартные» оценочные значения, на которые принято ориентироваться (от лучшего к худшему):

**O(1), O(log n), O(n), O(n log n), O(n^2), O(2^n), O(n!)**

То есть, приблизительно прикинув сложность нашего алгоритма, мы можем говорить про Big O, что он, допустим, O(n log n). То есть, ни при каких обстоятельствах не хуже. Но в то же время, недостаточно хорош, чтобы оцениваться как O(n).

**Задачка**

for (var i = 0; i < 10; i++){

setTimeout(function(){

console.log(i);

}, 1000);

}

К моменту срабатывания setTimeout, цикл уже полностью выполнится и переменная примет значение равное десяти. А раз на момент вызова переменная уже равна десяти, то и на вывод пойдет десять.

Для решения, Вам надо убрать зависимость и создать локальное замыкание. Поскольку в JS область видимости у функций, Вам надо создать функцию и передать ей значение на текущий момент. Тогда оно будет скопировано, а не использовано как ссылка.

for (var i = 0; i < 10; i++) {

((i) => {

setTimeout( function() { console.log(i); } , 1000);

})(i)

}

Заменить var на let каждую итерацию новая переменная будет создаваться

for (let i = 0; i < 10; i++){

setTimeout(function(){

console.log(i);

}, 1000);

}

**Передавать i в setTimeout**

for (var i = 0; i < 10; i++) {

setTimeout(console.log, 1000, i);

}

**Увеличивать, уменьшать i**

for (var i=0; i<10; i++) {

setTimeout(function() { console.log(i++ - 10); }, 1000);

**Using bind**

for(var i=0; i<10;i++) {

setTimeout(console.log.bind(null,i), i\*100);

}