JS

JS Core

1. https://jsfiddle.com

programming design patterns

SOLID principles

- программыне сущности: классы, модули, функции и т.п.
- Single Responsibility Principle
  - Принцип единственной обязанности
  - На каждый объект должна быть возложена одна единственная обязанность.
  - Разбивать объекты на более мелкие
- Open/Closed Principle
  - Принцип открытости/закрытости
  - Программные сущности должны быть открыты для расширения, но закрыты для изменения
- Liskov Substitution Principle
  - Принцип подстановки Барбары Лисков
  - Объекты в программе могут быть заменены их наследниками без изменения свойств программы
  - потомок должен дополнять, но не замещать методы родителя
- Interface Segregation Principle
  - Принцип разделения интерфейса
  - Клиенты не должны реализовывать ненужные методы, которые они не будут использовать
  - Разбивать интерфейсы на более мелкие
- The Dependency Inversion Principle DI
  - Критический взгляд на принцип инверсии зависимостей
  - Принцип инверсии зависимостей
  - Реализации на основе абстракций.
  - реализация зависит от абстракции, но не наоборот
  - модули должны зависеть от абстракций
  - модули верхнего уровня должны зависеть от модулей низкого

JS

- декларативный(.forEach) и императивный(for)
- ECMAScript — это язык программирования
  - объектно-ориентированный
  - с прототипной организацией
  - концепция объекта в качестве базовой абстракции
  - динамическая типизация

V8

- интерпретатор ES+WASM ignition/байткод
- компиляторturbofan JIT
- виртуальные машины
- однопоточный, а потому неблокирующий
- FIFO?
- динамическая типизация
  - примитивные типы присваивают значение, объекты - ссылки
  - https://codeburst.io/js-scope-static-dynamic-and-runtime-augmented-5abfee6223fe
- прототипное наследование
  - proto - sorax
  - https://dmitrysoshnikov.medium.com/oo-relationships-5020163ab162
  - почти у всех объектов есть прототип для хранения шаблона объекта, создания экземпляров через конструктор
  - класс - объект-шаблон для создания других объектов, экземпляров класса
  - Object.isPrototypeOf()
  - Object.instanceOf()
  - наследование
    - в конструкторе: `parentCLass.constructor.apply(this,arguments)`
    - в прототипе:
    -
  - `new` для определения нового `this` в новом экземпляре объекта/класса. Без new `this === undefined` в ES5, раньше - `window`

GC

- allocate,use,release memory
- задача точного выяснения того, нужен ли определенный участок памяти, алгоритмически неразрешима.
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management
- вместо алгоритма подсчёта ссылок сейчас браузеры применяют алгоритм проверки доступности(mark-compact). Например, циклические ссылки не считаются мусором в первом варианте, даже если на них нет ссылок извне.
- https://geekbrains.ru/posts/javascript_internals_part2
- активные функции(например, глобальные переменные) помечаются как корни, и от них проверяется доступность
- в отдельном потоке
- очищает не всю кучу, а последовательно по кускам через время
- use strict помогает избежать случайного объявления глобальной переменной(переменная без var или вызов функции с обращением к this.foo в глобальной области)
- глобальные переменные не очищаются, т.к. они корень
- забытые замыкания, со ссылками на глобальные переменные
- забытые таймеры, события, js ссылки на DOM - они не удаляются вместе с удалением узла

Event loop

- What the heck is the event loop anyway? | Philip Roberts | JSConf EU
- Jake Archibald: In The Loop - JSConf.Asia
- https://nodejs.dev/learn/the-nodejs-event-loop
- https://developer.mozilla.org/ru/docs/Web/JavaScript/EventLoop
- https://html.spec.whatwg.org/multipage/webappapis.html#event-loops
- Jake Archibald: все что я знаю про Event Loop в JavaScript (2018)
- https://medium.com/@olinations/the-javascript-runtime-environment-d58fa2e60dd0
- https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/
- приоритеты
    1. макрозадачи(без DOM, в тестах, вызов событий тоже в этой очереди)
    2. микрозадачи(после очистки макрозадач) - Promise.then()
    3. микрозадачи с задержкой - setTimeout(cb, time)
    4. отрисовка
- планирование задач
    - задачи про DOM --> RAF
        - https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame
        - https://developer.mozilla.org/en-US/docs/Web/API/Window/cancelAnimationFrame
        -

```
    let id = window.requestAnimationFrame(cb:(timestamp)=>number);
cancelAnimationFrame(id);
```

        - запускает обратный вызов перед рендерингом, возвращает номер для отмены обещания
        - стили могут не применяться, если перетирают друг друга
        - getComputedStyle() заставит применить стили после назначения свойств и отрисовать их на GPU
        - до 2019 safari выполнял RAF после рендеринга https://bugs.webkit.org/show_bug.cgi?id=177484
    - фоновые --> requestIdleCallback
        - https://developer.mozilla.org/en-US/docs/Web/API/Window/requestIdleCallback
        -

```
    let id = window.requestIdleCallback(cb:()=>number,options?:
{timeout:number})
```

-
  - тяжёлая --> webworker
    - общение с DOM через postMessage
      - https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage
      - общение через адрес страницы
    - нет DOM, событий
    - есть XHR, promise
    - свой контекст, вне CSP документа

```
    // worker.js
    onmessage = function(e) {
        console.log('Worker: Message received from main script');
        const result = e.data[0] * e.data[1];
        if (isNaN(result)) {
            postMessage('Please write two numbers');
        } else {
            const workerResult = 'Result: ' + result;
            console.log('Worker: Posting message back to main
script');
            postMessage(workerResult);
        }
    }

    // index.js
    if (window.Worker) {
        const myWorker = new Worker("worker.js");

        first.onchange = function() {
            myWorker.postMessage([first.value, second.value]);
            console.log('Message posted to worker');
        }

        myWorker.onmessage = function(e) {
            result.textContent = e.data;
            console.log('Message received from worker');
        }
    }

    myWorker.terminate();
```

  - приоритетная --> setTimeout(cb, 0)
    - понижает приоритет отрисовки
    - добавляет событие в очередь

- очередь задач queue task
  - привязаны источники событий

    - мышь
    - клава
    - XHR

- serviceworker
    - работа по сети, кэширование
    - являются web worker'ами под капотом

```
if ('serviceWorker' in navigator) {
window.addEventListener('load', function() {
    navigator.serviceWorker.register('/sw.js');
});
}
```

- DOM
    - Critical Rendering Path: DOM, CSSOM, Render tree, Layout, Paint
    - https://developer.mozilla.org/ru/docs/Web/Performance/Critical_rendering_path
    - Построение DOM инкрементально. Ответ в виде HTML превращается в токены, которые превращаются в узлы (nodes). Узлы (nodes) связаны с Render Tree с помощью иерархии токенов.
    - Чем больше количество узлов (node) имеет приложение, тем дольше происходит формирование DOM tree
    - Если формирование DOM инкрементально, CSSOM - нет. CSS блокирует рендер: браузер блокирует рендеринг страницы до тех пор, пока не получит и не обработает все CSS-правила. CSS блокирует рендеринг, потому что правила могут быть перезаписаны
    - CSS-правила ниспадают каскадом, вложенные узлы наследуют стили от родительских.
    - наименее специфичные селекторы срабатывают быстрее.
    - Дерево рендера охватывает сразу и содержимое страницы, и стили: это место, где DOM и CSSOM деревья комбинируются в одно дерево. Для построения дерева рендера браузер проверяет каждый узел (node) DOM, начиная от корневого (root) и определяет, какие CSS-правила нужно присоединить к этому узлу.
    - Дерево рендера охватывает только видимое содержимое. Например, секция head может не включаться в дерево. display: none так же не включается в дерево (как и потомки этого узла).
    - В тот момент, когда дерево рендера (render tree) построено, становится возможным этап компоновки (layout). Компоновка зависит от размеров экрана.
    - отрисовка (paint) пикселей на экране

- микрозадачи
    - resolved promise
    - mutation observer(нельзя добавить второй наблюдатель, пока не пустая очередь микрозадач)
    - intersection observer
    - стартуют после очистки очереди макрозадач

    - выполняются до очистки очереди микрозадач
    - могут порождать новые микрозадачи

- ```
  console.log(1);
  const promise = new Promise(resolve => {  resolve() });
  promise.then(() => { setTimeout(() => {  console.log(2); })});
  setTimeout(() => {  console.log(3); }, 500);
  setImmediate(() => { console.log(4);});
  setTimeout(() => { console.log(5); }, 0);
  promise.then(() => { console.log(6);});
  console.log(7);


  // 1 7 6 undefined 4 5 2 3
  ```

Новое в ES6 https://tc39.es/ecma262/

- ES6 по-человечески
- let, const
    - let, const - блочная область видимости(scope), не всплывают, не добавляются в this, не существуют до своего объявления
    - const - неизменяемое значение(примитив), ссылка(объект). Object.freeze() позволяет защитить первый уровень вложенных объектов.
    - var - функциональная область видимости, требуют изоляции в замыканиях
- Стрелочные функции
    - не имеют своего this, arguments
- Параметры функций по умолчанию
- Spread/Rest оператор
- Расширение возможностей литералов объекта
- Восьмеричный и двоичный литералы
- Деструктуризация массивов и объектов
- Ключевое слово super для объектов
- Строковые шаблоны и разделители
- for...of , for...in
    - for of
        - нельзя пользовать аннотации
        - по итерируемым свойствам
        - Array, nodeList
        -
    - for in
        - по перечисляемым свойствам, игнорируя неитерируемые
        - нет строгого порядка обхода
        - нежелательно менять/добавлять свойства до прохода итератора
        - нельзя деструктурирующие присваивания
        - object
- Map/set
  ```

    -
- WeakMap/WeakSet
    - Weak поощряет сборку мусора, т.к. содержит меньше ссылок
    - нельзя итерировать, т.к. нет итератора
- Классы в ES6
- Тип данных Symbol
    - https://www.programiz.com/javascript/symbol
    - https://www.javascripttutorial.net/es6/symbol/
    - https://medium.com/intrinsic/javascript-symbols-but-why-6b02768f4a5c
    - https://javascript.info/symbol
    - не перечисляемый(enumerable)
- Итераторы
    - ? ссылки на все элементы
- Генераторы
- Classes
- Promises
- Symbol
- String.includes()
- String.startsWith()
- String.endsWith()
- Array.from()
- Array keys()
- Array find()
- Array findIndex()
- New Math Methods
- New Number Properties
- New Number Methods
- New Global Methods
- Object entries
- JavaScript Modules

ES2021(12)

```
* https://levelup.gitconnected.com/top-5-javascript-es12-features-you-should-start-using-now-b16a8b5353b1
    * promise.any()
    * ||= Logical OR assignment operator
        * if false
        * `a || (a = b)`
    * &&= Logical AND assignment operator
        * is true
        * `a && (a = b)`
    * ??= Nullish coalescing assignment operator
        * is null or undefined
        * `a ?? (a = b)`
```

ES modules

````
* https://v8.dev/features/modules#mjs
* https://hacks.mozilla.org/2015/08/es6-in-depth-modules/
    * автоматически "use strict"
    * можно делать import/export
*

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

    // Aggregating modules
    export * from 'nested1.js'
    export { name } from 'nested2.js'

    export {a,b,c}
    const a = false, b = 0, c = '';

    // Dynamic module loading
    import('./modules/myModule.js')
        .then((module) => {
            // Do something with the module.
        });

    // Top level await
    const colors = fetch('../some.json')
        .then(response => response.json());

    export default await colors;
    // import colors from './modules/getColors.js';

```

* в браузере пока лучше использовать .js вместо .mjs - нужен `Content-Type text/javascript`. Иначе будет strict MIME type checking error: `"The server responded with a non-JavaScript MIME type"`
* локально через `file://` не работает - CORS
* по-умолчанию strict mode
* по-умолчанию используют defer script attribute
* модули исполняются только один раз, даже в нескольких <script>
* модули не видны в глобальной области видимости, только локально
````

Промисы

- https://dev.to/lydiahallie/javascript-visualized-promises-async-await-5gke
- отменяемые обещания
- async/await

```
// async - это promise.resolve
async function foo() { return 1; }
// It is similar to:
function foo() { return Promise.resolve(1); }

// async меняет ссылку
const p = new Promise((res, rej) => { res(1); });
async function asyncReturn() { return p; }
function basicReturn() { return Promise.resolve(p); }
console.log(p === basicReturn()); // true
console.log(p === asyncReturn()); // false
```

bind, call, apply

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply
- bind(this, obj, ...obj) - без вызова
- apply(this, arr)
- call(this, obj, ...obj)

deep clone клонирование объектов

- structuredClone(value)
- https://developer.mozilla.org/en-US/docs/Glossary/Deep_copy
- только через JSON, остальное - shallow copy

this

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this
- для глобального контекста this - это объект Window

Use strict в ES6

- делает контекст this по-умолчанию undefined вместо window
- переменные без объявления выводят ошибку, защита от очепяток
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode
- включает ошибки для плохого кода, делает код более безопасным
- меняет разрешение ссылок на переменные
- меняет eval и arguments

Разница между == и ===

- присваивание с приведением типов

функции

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions
- вызываемый callable объект

- первого класса - можно использовать как текст, присваивать
- функциональные выражения
  - не всплывают
  - могут быть именованными и безымянными
- super

```
super([arguments]); // calls the parent constructor.
super.functionOnParent([arguments]);
```

- yield
  - поддерживает итерируемые for-of и итерирующие `next():(next:any)|(done:boolean)` протоколы
  - вместо return yield

```
function* counter(value) {
    let step;

    while (true) {
        step = yield ++value;

        if (step) {
            value += step;
        }
    }
}

const generatorFunc = counter(0);
console.log(generatorFunc.next().value);   // 1
console.log(generatorFunc.next().value);   // 2
console.log(generatorFunc.next().value);   // 3
console.log(generatorFunc.next(10).value); // 14
console.log(generatorFunc.next().value);   // 15
console.log(generatorFunc.next(10).value); // 26
```

- yield*

```
function* g1() {
    yield 2;
    yield 3;
    yield 4;
}

function* g2() {
```

```
        yield 1;
        yield* g1();
        yield 5;
    }

    const iterator = g2();

    console.log(iterator.next()); // {value: 1, done: false}
    console.log(iterator.next()); // {value: 2, done: false}
    console.log(iterator.next()); // {value: 3, done: false}
    console.log(iterator.next()); // {value: 4, done: false}
    console.log(iterator.next()); // {value: 5, done: false}
    console.log(iterator.next()); // {value: undefined, done: true}
```

- функциональные выражения

Разница между TS и JS

- чего нет в JS, что есть в TS: properties(в классе), decorators
- новые типы данных
  - enum
  - mixin `T extends ...`
  - decorator - объявление @функции перед классом, геттером, параметром, функцией. Внутрь декоратора передаётся последующий операнд или результат следующего декоратора. У классов передаётся конструктор, он его подменяет.

В чем отличие throttle от debounce

- debounce - адаптивный/динамический, отсеивает слишком частые, таймер между значениями
- throttle - выбирает через интервал

PWA - webwokers, manifest, best practices

JSON.stringify

- сохраняет только названия свойств объектов, примитивы, null
- игнорирует свойства со значениями symbol, function, undefined

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

- немедленный вызов в момент определения
- изоляция в замыкании

замыкание

- у функций есть контекст(scope) и лексическое окружение
- Контекст исполнения — это абстрактное понятие, используемое спецификацией ECMAScript для типизации и разграничения исполняемого кода.
  - Стек контекстов исполнения: Стек контекстов исполнения — это LIFO структура, используемая для контроля и очередности исполнения кода.

    - Каждый контекст исполнения имеет ассоциированное с ним лексическое окружение (lexical environment).
    - по сути это ссылка на родительское лексическое окружение
    - появляется в момент создания функции
- Лексическое окружение — это структура, используемая для ассоциации идентификаторов, появляющихся в контексте, с их значениями. Каждое лексическое окружение также может иметь ссылку на родительское окружение.
    - по сути объект со всеми переменными и методами вызванной функции
    - появляется в момент запуска функции
- Чистая функция: без побочных эффектов, возвращает одни и те же данные для одного набора входных параметров
- Функция первого класса: функция, которая может быть использована в качестве обычных данных: т.е. сохранена в переменную, передана в качестве аргумента, или возвращена в качестве значения из другой функции. Функции в ECMAScript являются объектами первого класса.
- Свободная переменная: переменная, не являющаяся ни параметром, ни локальной переменной данной функции.
- Статическая/лексическая область видимости: язык программирования использует статическую область видимости, если только по анализу исходного кода, можно определить, в каком лексическом окружении будут разрешены свободные переменные.
- Замыкание — это функция, захватывающая лексическое окружение того контекста, где она создана. В дальнейшем это окружение используется для разрешения идентификаторов.
    - простейший пример замыкания - функция, возвращающая функцию
    - у новой функции есть ссылка на лексическое окружение родительской, поэтому оно не уничтожается

new

- создаёт новый объект и вызывает в контексте этого объекта переданную функцию, в случае передачи объекта - функцию-конструктор
- без new контекст - this
- Object.create(прототип)

класс

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
- по сути функция
- свойства в контексте, методы в прототипе
- три типа наследования
    - классическое - extends, конструкторы
        - Javascript-джедай #21 - Конструкторы и классы
        - прямая подмена прототипа и вызов new
        - любая функция может быть конструктором
        - любую функцию можно вызвать с new
        - любая функция имеет прототип
    - прототипное - модификация proto через Object.create()
    - функциональное - взять свойства родителя в конструкторе через `parent.call(this)`

Шаблоны проектирования

- GOF/GRASP

Циклы

- for() {}
- while(){}
- do {} while()
- for (of) - итерируемые
  - `for (const [key,value] of Object.entries(obj) {}`
- for (in) - перечисляемые
  - `for (const key in Object.entries(obj) { obj[key as keyof typeof obj] }`

Практика

- итерирование по свойствам объекта: for in, Object.keys, ...obj,
- итерирование по свойствам DOM узла

поднятие hoisting

- https://developer.mozilla.org/en-US/docs/Glossary/Hoisting
- функций, классов, переменных

```
    catName("Tiger");

    function catName(name) {
    console.log("My cat's name is " + name);
    }

    // hoists declarations, not initializations!
    console.log(num); // Returns 'undefined' from hoisted var declaration
(not 6)
    var num; // Declaration
    num = 6; // Initialization
    console.log(num); // Returns 6 after the line with initialization is
executed.

    console.log(num); // Throws ReferenceError exception - the interpreter
doesn't know about `num`.
    num = 6; // Initialization

    // функциональные выражения не всплывают
```

- вызов переменной перед определением
  - var - без ошибок
  - let - referenceError

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz

```
{ // TDZ starts at beginning of scope
    console.log(bar); // undefined
    console.log(foo); // ReferenceError
    var bar = 1;
    let foo = 2; // End of TDZ (for foo)
}
{
    // TDZ starts at beginning of scope
    const func = () => console.log(letVar); // OK

    // Within the TDZ letVar access throws `ReferenceError`

    let letVar = 3; // End of TDZ (for letVar)
    func(); // Called outside TDZ!
}
// results in a 'ReferenceError'
console.log(typeof i);
let i = 10;

function test() {
    var foo = 33;
    if (foo) {
        let foo = foo + 55; // ReferenceError
    }
}
test();

let x = 1;

{
var x = 2; // SyntaxError for re-declaration
}
```