**Урок 9**

**1. Контекст виконання функції**

Контекст виконання (execution context) — це, якщо говорити спрощено, концепція, що описує оточення, у якому виконується виконання коду JavaScript. Код завжди виконується всередині певного контексту.

Контекст виконання функції. Щоразу, коли викликається функція, нею створюється новий контекст. Кожна функція має власний контекст виконання. У програмі може одночасно бути безліч контекстів виконання функцій. Під час створення нового контексту виконання функції він проходить через певну послідовність кроків.

Стек виконання (execution stack), який ще називають стеком дзвінків (call stack), це стек, який використовується для зберігання контекстів виконання, створюваних під час роботи коду.

Коли JS-движок починає обробляти скрипт, движок створює глобальний контекст виконання і поміщає їх у поточний стек. При виявленні команди виклику функції движок створює новий контекст виконання цієї функції і поміщає їх у верхню частину стека.

Двигун виконує функцію, контекст виконання якої знаходиться у верхній частині стека. Коли робота функції завершується, її контекст витягується зі стека і керування передається тому контексту, який знаходиться в попередньому елементі стека.

Вивчимо цю ідею за допомогою такого прикладу:

 const a = 'Hello World!';

        function first() {

        console.log('Inside first function');

        second();

        console.log('Again inside first function');

        }

        function second() {

        console.log('Inside second function');

        }

        first();

        console.log('Inside Global Execution Context');

Коли даний код завантажується в браузер, JavaScript-движок створює глобальний контекст виконання і поміщає його в поточний стек викликів. При виконанні виклику функції first() двигун створює для цієї функції новий контекст і поміщає його у верхню частину стека.

При виклику функції second() з функції first() для цієї функції створюється новий контекст виконання і поміщається в стек. Після того, як функція second() завершує роботу, її контекст витягується зі стека і управління передається контексту виконання, що знаходиться в стеку під ним, тобто контекст функції first().

Коли функція first() завершує роботу, її контекст витягується зі стека і управління передається глобальному контексту. Після того, як весь код виявляється виконаним, двигун отримує глобальний контекст виконання з поточного стека.

**2. Правила визначення this**

1. Є this стрілочні і не стрілочні функції. Не стрілочні це звичайні function Expression і function declaration. This при виклику звичайної функції дорівнює обєкту, котрому функція належить в момент виклику. Залежно від того хто, коли, як викликає звичайну функцію, у неї може змінюватися господар тимчасово. і this дорівнює її господареві в момент виклику.

2. This – це завжди посилання на об'єкт.

3. Господар надається функції в момент створення, але може бути змінений в момент виклику (функція тимчасово позичається іншому об'єкту).

function calc(foot, inch){

            const result = (foot \* 12 + inch) \* 2.54;

            console.log(this);

            return result;

        }

        calc();

4.Функція, оголошена у загальному потоці, належить window. Функція calc була обявлена в загальному потоці.

Якщо ж функція створюється як метод об'єкта, то вона належить йому.

const user = {

            name: "Алекс",

            lastName: "Гийом-Василь",

            getFullName(){

                const fullName = `${this.name} ${this.lastName}`;

                // calc();

                console.log(this);

                return;

            }

        }

// user.getFullName();

У функції є метод getFullName() котрий повертає name та lastName з'єднані разом. Консоль лог this це об'єкт, тобто функція посилається на об'єкт. І зазвичай де б функція не викликалася в неї господар не змінюється. Якщо я в getFullName() викличу функцію calc в середині то вона, як і раніше, належить window.

Як приклад можна навести, що GO IT знімають офіс у бізнес центрі хвиля. У бізнес-центрі Хвиля наймають прибиральниць і контракт вони підписують з бізнес-центром Хвиля. Неважливо де вони прибирають у Go IT сусідньому офісі або десь ще. Вони все одно належать бізнес центру Хвиля. Тобто неважливо де функція викликається вона належить господарю. У момент виклику функції завжди THIS дорівнює її господареві. Створюється змінна THIS і вона дивиться кому належить функція під час виклику. У звичайному випадку вона належить тому, хто став її господарем у момент створення.

Але є винятки. Перший виняток

Я створю кнопку btn- id.

<button id="btn">Нажми меня</button>

const button = document.getElementById("btn");

        // button.addEventListener("click", calc);

    // button.addEventListener("click", user.getFullName);

        button.addEventListener("click", user.getFullName.bind(user));

**Другим аргументом в** addEventListener **ми маємо передати функцію. Ми можемо її не створити а відразу вказати назву готової функції. У цьому випадку я вкажу функцію calc і перевіряйте. ми бачимо у консолі що THIS функції calc дорівнює будинку елемента до якого прив'язана подія.**

Тобто грубо кажучи

5. This функції-оброблювача події дорівнює DOM-елементу, до якого прив'язана подія.

(Функція тимчасово позичається DOM-елементу).

Припустимо, на підприємстві немає прибиральниці, а вона їм потрібна на даний момент. Вони йдуть до керівництва бізнес-центру Хвиля і просять тимчасово перевести їм до штату прибиральницю. Тобто коли до них приходить прибиральниця вона формально вступає до їхнього штату. Поки вона у них працює, вона належить їм. Вийшла за поріг уже знову співробітник бізнес-центру. Тобто якщо функція прив'язана як обробник події, то вона тимчасово позичається.

Абсолютно те саме з методом.

    // button.addEventListener("click", user.getFullName);

Ми можемо так написати тільки замість функції ми передали метод. Вона належала об'єкту, але в момент виклику стала належати методу.

GetFullName за замовчуванням this дорівнює обєкту якщо ми викликаємо цю функцію. // user.getFullName();

setTimeout – дозволяє нам викликати щось з затримкою. Першим аргуметом до сет таймауту передається функція котру потрібно викликати, другим проміжок в нас він 3000 мілі секунд це 3 секунди.

// setTimeout(user.getFullName, 3000);

Тобто я хочу викликати user.getFullName через 3 секунди. Чому дорювнює this. Він дорівнює window. Тому що setTimeout теж перевизначає.

6. This функції - перший аргумент setTimeout і setIterval дорівнює

window у момент виклику(Функція тимчасово позичається window).

Тобто щоб такому запобігти треба писати ось таку інструкцію:

setTimeout(function(){

console.log(this);

            user.getFullName()

        }, 3000)

У стрілочній функції буде віндоу, а тут вже нічого не торкнеться у фігурних дужках.

Є 2 випадки перевизначення THIS це присвоювання події та сет тайм.

**2.1.** **this у глобальній області видимості**

У глобальній області видимості, якщо скрипт виконується не в жорсткому режимі, this посилається на об'єкт window. У строгому режимі значення this, у глобальній області видимості, буде undefined.

Тобто ми можемо створити функції через Function expression та function declaration

//? Function expression

const fn = function () {

  console.log(`this ---->`, this);

};

fn();

//? Function declaration

function fn1() {

  console.log(`this ---->`, this);

}

Якщо ми зараз подивимось в консоль то побачимо що в обох випадках наш this зараз належить window.

Але якщо ми поставимо строгий режим тобто напишемо use strict, то наш this буде undefined.

**2.2. this в методі обєкта**

Якщо функція була викликана як метод об'єкта, то контекст буде посилатися на об'єкт, частиною якого є метод.

//TODO: Розглянемо як this поводиться в методах

const user = {

  name: 'Luis',

  age: 30,

  showThis() {

    // this = user;

    console.log('this ----> ', this);

  },

  showName() {

    // this = user;

    console.log(this.name);

  },

};

user.showThis();

user.showName();

const anotherUser = {

  name: 'Barry',

  age: 20,

  showThis() {

    // this = user;

    console.log('this ----> ', this);

  },

  showName() {

    // this = user;

    console.log(this.name);

  },

};

anotherUser.showThis();

anotherUser.showName();

**2.3. this в callback-функціях**

Передаючи методи об'єкта як колбек-функції, контекст не зберігається. Колбек - це посилання на метод, яке присвоюється як значення параметра, що викликається без об'єкта.

//TODO: This в callback функціях

const user = {

  name: 'Luis',

  age: 30,

  showThis() {

    console.log(`this ---->`, this);

  },

  showName() {

    console.log(`this name is: ${this.name}`);

  },

};

user.showThis();

user.showName();

const user2 = {

  name: 'mango',

};

const someFunction = function (callback) {

  console.log(callback === user.showThis);

  callback();

};

someFunction(user.showThis);

**2.4. this у стрілочних функціях**

**This стрілочної функція визначається в момент створення і є незмінним у процесі роботи, тобто стрілочну функцію не можна нікому позичити.**

**this стрілочної функції дорівнює this функції-батька (функції, в якій вона була створена).**

**1. Ця стрілочна функція, створена в загальному потоці, дорівнює window.**

**2. Ця стрілочна функція, створена в методі об'єкта, дорівнює цьому об'єкту.**

const calc = (foot, inch) => {

            const result = (foot \* 12 + inch) \* 2.54;

            console.log(this);

            return result;

        }

        calc();

        const button = document.getElementById("btn");

        button.addEventListener("click", calc);

Це не звичайна це стрілочна функція. Просто THIS у ній це WINDOW. А тепер ми цю функцію присвоюємо також як обробник події і THIS теж дорівнює WINDOW. Ця стрілочна функція не має свого THIS тому вона запозичує його у свого батька.

Ти завжди знатимеш чому дорівнює THIS, але ти ніколи не зможеш його поміняти.

Другий важливий момент:

        const user = {

            name: "Алекс",

            lastName: "Гийом-Василь",

            getFullName(){

                const fullName = `${this.name} ${this.lastName}`;

                // console.log(this);

                const func = ()=> {

                    console.log(this);

                }

                return func;

            }

        }

**Ми створюємо стрілочну функцію всередині геть фул нейма. Та пишемо**

const newFunc = user.getFullName();

**Тобто ми викликаємо геть фулл нейм і нам повертає стрілочну функцію, яка була тут створена.** const func = ()=> {

                    console.log(this);

}

**Та ми її викликаємо:**

// console.log(newFunc)

// newFunc()

**Тобто чому дорівнює зис стрілочної функції? THIS функції всередині якої вона була створена. Тобто THIS getFullName дорівнює об'єкту тесть (USER у нашому випадку) і у цієї стрілочної функції THIS при виклику теж буде дорівнює (USER). Відповідно якщо ми як обробник повісимо цю функцію. THIS у нас не перевизначиться.**

// button.addEventListener("click", newFunc);

**Навіть якщо ми зробимо те саме в сет таймауті. THIS дорівнюватиме об'єкту.**

// setTimeout(newFunc, 2000)

const user = {

            name: "Алекс",

            lastName: "Гийом-Василь",

            getAge: ()=> {

                console.log(this) --- ВОТ ТУТ WINDOW

            },

getFullName(){

                const fullName = `${this.name} ${this.lastName}`;

                const func = ()=> { --- ТУТ ФУНКЦІЯ БАТЬКО getFullName ДЛЯ func

                    console.log(this);

                }

                return func;

            }

        }

**Тільки не плутайте якщо ми створимо тут стрілочну функцію як тут батьком стрілочної функції у нас буде функція WINDOW.**

**3. Методи функцій**

Три методи функцій : call(), apply(), bind().

За допомогою цих методів ми можемо явно вказати значення this. Ці три методи схожі між собою, але тим не менш мають суттєві відмінності у роботі.

call та apply дуже схожі, вони обидва приймають значення this та аргументи, і викликають функцію з цими даними. Різниця лише в тому, що call приймає аргументи окремо, а apply – у масиві.

**3.1. Метод Call()**

Метод call() викликає функцію із зазначеним значенням this та індивідуально наданими аргументами. Перший аргумент служить контекстом виклику і стає значенням ключового слова цієї функції в тілі. Усі аргументи call() – це значення, що передаються викликаної функції. Завдяки методу call() будь-який об'єкт може використовувати методи, що належать іншому об'єкту.

var x = 10;

        var obj = { x: 15 };

        function fun() {

        alert(this.x);

        alert(this);

        }

        fun();          // 10, [object Window]

        fun.call(obj);  // 15, [object Object]

Перший виклик fun() відобразить значення 10, так як це посилається на глобальний об'єкт Window. Другий виклик (через метод call) відобразить значення 15. 15 – це значення властивості x всередині об'єкта obj. Тут метод call() використовується виклику fun() (методу) від імені об'єкта obj.

Після контексту в методі call можна передати аргументи для функції, що викликається:

const person = {

        firstName: "John",

        lastName: "Konor",

        specialty: "юрист-консультант"

        };

        function fullName(param1, param2) {

        alert( this[param1] + " " + this[param2] );

        }

        // func.call(контекст, аргумент1, аргумент2, ...)

        fullName.call(person, 'firstName', 'lastName') // John Konor

        fullName.call(person, 'firstName', 'specialty') // John юрист-консультант

В даному прикладі ми двічі передаємо аргументи до нашої функції що викликається.

Також за допомогою call ви можете використовувати метод, що належить одному об'єкту, а викликати його в контексті іншого:

        const person = {

        firstName:"John",

        lastName: "Konor",

        fullName: function() {

        return this.firstName + " " + this.lastName;

         }

        }

        const user = {

        firstName:"Max",

        lastName: "White",

        }

        alert(person.fullName.call(user));  // "Max White"

В данному прикладі ми використали метод який належить обєкту user в в контексті обєкта person.

8. .call(object) позичає функцію object у момент виклику(this === object).

**3.2. Метод apply()**

Метод apply() ідентичний call(), за винятком того, що apply() вимагає, щоб як другий параметр був обраний масив (або масивоподібний об'єкт). Масив наводить аргументи для цільового методу.

Метод apply() корисний, якщо у вас є масив і ви хочете використовувати його значення як аргументи функції, для яких інакше довелося б писати цикл масиву значень. Класичний приклад – пошук мінімальної чи максимальної кількості в масиві. Допоміжним функціям Math.max()/Math.min() можна передати будь-яку кількість аргументів, а вони повертають мінімальне або максимальне значення відповідно. Ми можемо використовувати метод apply(), щоб викликати ці функції з наявним масивом:

        var arr = [7, 8, 4, 5, 9];

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

        var min = Math.min.apply(null, arr);

        alert( max );   // 9

        alert( min );   // 4

        /\* сравним с циклом по массиву значений \*/

        max = -10, min = +10;

        for (var i = 0; i < arr.length; i++) {

        if (arr[i] > max) {

        max = arr[i];

        }

        if (arr[i] < min) {

        min = arr[i];

        }

        }

        alert( max );   // 9

        alert( min );   // 4

Зверніть увагу, що замість this значення ми просто передаємо null. В даному випадку як контекст можна передавати що завгодно, оскільки у своїй внутрішній реалізації методи Math.max і Math.min не використовує this взагалі.

**3.3. Метод bind()**

Метод bind прив'язати контекст до функції. Як перший параметр слід передавати контекст, а наступними параметрами - параметри функції. Метод повертає нову функцію, всередині якої this буде рівним переданому контексту.

var obj = { num: 2 };

        function add(a, b){

        return this.num + a + b;

        }

        const func = add.bind(obj, 3, 5);

        func(); // Возвращает 10

Отже, виклик add.bind(obj, 3, 5) повертає функцію. У цьому випадку ви надаєте її константі func і потім виконуєте.

Тут виклик func() означає виклик функції add() об'єкта obj з аргументами 3, 5.

7. .bind(object) прибиває значення this цвяхами - воно стає рівним об'єкту, вказаному в аргументі bind.

**3.4. bind() і методи обєкта**