Занятие 5 (13). Ключевое слово THIS, NEW,

методы функций call(), apply(), bind()

THIS

**function declaration:**

f() {…}

**arrow function:**

let f = () => {}

|  |  |
| --- | --- |
| **Declaration f() {…}** | **Arrow let func = () => {}** |
| 1. Число аргументов не фиксировано, можно использовать Объект **arguments** — это подобный массиву объект, который содержит аргументы, переданные в функцию. 2. Значение **this** зависит от того от чьего имени была **вызвана** функция:   - **обычное выполнение:** this - глобальный объект window в браузере (*объект слева от точки)*  - **вызов как метода объекта:** this - это объект, которому принадлежит метод  - **использование call() и apply():** this - это первый аргумент  3. **Нужно явно задавать return** (либо не задавать его, вернется undefined)  4. **Есть методы функций**: call(), apply(), bind()  5. **Есть Prototype**  6. **Есть super()** | 1. Фиксированное число аргументов. Как и со значение this массив arguments для стрелочных функций будет браться из внешней функции. 2. Не имеет значения, как она была вызвана, значение **this** внутри стрелочной функции **всегда эквивалентно значения this внешней функции в момент ее объявления**. Функция не создает собственный контекст исполнения, она использует внешний. 3. Если стрелочная функция содержит в теле одну инструкцию, и ты опустил фигурные скобки, тогда выражение будет **возвращено автоматически**. 4. **Нет методов функций** call(), apply(), bind() 5. **Нет Prototype** 6. **Нет Super ()** |

**Совет по использованию**: в основном надо пользоваться **declaration**, **стрелочной** функцией только тогда, **когда важно сохранение контекста**, чтобы не привязывать каждый раз функцию к новому объекту.

В **setTimeout** и **setInterval** this привязывается к window, т.к. они оба являются браузерными API, но если браузерные API вызваны в методе объекта, то declaration примет this от window, а стрелочная примет от метода(this будет объект метода)

Объявление функции происходит до срабатывания setTimeout!

**this** **стрелочной** функции там, где она **написана**!!! Если стрелочная функция - метод объекта, что this будет window, т.к. объект не создает область видимости.

**this** **обычной** функции там, где она **вызвана**!!!

В **Typescript** this надо выносить в аргументы и там типизировать:

grunt.registerMultiTask('clean', function(**this: {files:any[]}**) {

this.files.forEach(function(f) { Delete(f); });

});

Метод bind() (возвращает новую привязанную функцию)

Допустим, у объекта есть метод. Как можно применить этот метод к другому объекту?

*let* obj1 = {

    name: 'Eugene',

    age: 32,

    sayName() {

        console.log(this.name);

    }

}

*let* obj2 = {

    name: 'Eugene',

    age: 32

}

Есть два варианта решения:

1. Напрямую передать ссылку: obj2.sayName = obj1.sayName. Но в этом случае мы расширяем объект! Мутируем его.
2. Использовать методы функций.

Метод **bind(контекст, аргументы)** ***создаёт новую******привязанную*** *функцию*, которая при вызове устанавливает в качестве контекста выполнения this предоставленное значение.

Например:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

let obj1 = { // объект1

name: 'Eugene',

age: 32,

sayName(arg1, arg2) {

console.log(this.name, arg1, arg2);

}

}

let obj2 = { // объект1

name: 'Eugene',

age: 32

}

let nameFunc = obj1.sayName.bind(obj1, 10, ‘hello’) // байндим с obj1 в качестве this и аргументы. В этом случае жестко привязывается контекст и аргументы.

nameFunc() // вызов функции даст ‘Eugene, 10, hello’

//или

let nameFunc1 = obj1.sayName.bind(obj1) // байндим с obj1 в качестве this, без аргументов. В этом случае аргументы передаем при вызове функции

nameFunc1(15, 'Mike')

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Методы функций call(), apply() - делают вызов функции на месте

**Метод call() вызывает функцию с указанным значением this и индивидуально предоставленными аргументами через запятую.**

**Метод apply() вызывает функцию с указанным значением this и индивидуально предоставленными аргументами в виде массива**

**Больше разницы нет!!**

Например:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

let obj1 = { // объект1

name: 'Eugene',

age: 32,

sayName(arg1, arg2) {

console.log(this.name, arg1, arg2);

}

}

let obj2 = { // объект1

name: 'Eugene',

age: 32

}

**// CALL**

obj1.sayName.call(obj2) // в консоль выведется ‘Eugene, undefined, undefined’

obj1.sayName.call(obj2, 20, 'Stan') // в консоль выведется ‘Eugene, 20, ‘Stan’’

**// APPLY**

obj1.sayName.apply(obj2) // в консоль выведется ‘Eugene, undefined, undefined’

obj1.sayName.apply(obj2, [20, 'Stan']) // в консоль выведется ‘Eugene, 20, ‘Stan’’

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**В реальных проектах на React в классовых компонентах контекст выполнения лучше привязывать стрелочными функциями. Т.е. в качестве метода пишем стрелочные функции и все.**

Если в качестве методов используется function declaration, то надо прописывать bind намеренно, чтобы сохранился контекст.

Как еще можно использовать call и apply ? Например, можно украсть метод у объекта и применить его к той сущности, которая не обладает таким методом. Например, псевдомассив из querySelectorAll или arguments:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

[].filter.call(arr) - функция это тоже объект, от нее можно привязать this

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_