**Разное**

**Proxy**

Вспомогательный объект Proxy позволяет создать оболочку над объектом и через эту оболочку контролировать его работу.

**let proxy = new Proxy (target, handler)**

Здесь target собственно сам объект, а handler – это оболочка или как еще называют ловушка.

let obj = {

label1: 1

}

let objProxy = new Proxy(obj, {})

objProxy.label2 = 2

console.log(objProxy)

console.log(obj)

Обратите внимание, objProxy без ловушек, поэтому это просто прозрачная оболочка.

**Ловушка get**

Ловушка get позволяет перехватить вызовы чтения свойств.

**get(target, property)**

Часто употребляется, когда надо возвращать значение по-умолчанию не undefined, а что-то другое:

let numberArr = [1,32,43,42543,535345]

numberArr = new Proxy(numberArr, {

get(target, property) {

return property in target ? target[property] : -1

}

})

console.log(numberArr[323])

**Ловушка set**

Ловушка set позволяет соотвественно перехватывать записи свойств. В отличии от get ловушка set должна возвращать true, если операция произошла успешно и false соотвественно.

**set(target, property, value)**

Можно продолжить пример сверху, но уже сделать массив, только их чисел:

let numberArr = [1,32,43,42543,535345]

numberArr = new Proxy(numberArr, {

get(target, property) {

return property in target ? target[property] : -1

},

set(target, property, value) {

if(typeof value === 'number') {

target[property] = value

return true

}

return false

}

})

console.log(numberArr[323])

numberArr.push('String') // Ошибка

console.log(numberArr)

**ownKeys**

Ловушка ownKeys позволяет перехватить перебор свойств объекта:

**ownKeys(target)**

Эта функция должна вернуть массив с ключа допустимых пар:

let user = {

name: 'John',

age: 33,

\_id: 1324234423432,

\_bank: 'Visa'

}

user = new Proxy(user, {

ownKeys(target) {

return Object.keys(target).filter(key => !key.startsWith('\_'))

}

})

for(let key in user) {

console.log(`${key} : ${user[key]}`)

}

**Ловушка deleteProperty**

Ловушка deleteProperty позволяет перехватить удаление свойств.

let user = {

name: 'John',

age: 33,

\_id: 1324234423432,

\_bank: 'Visa'

}

user = new Proxy(user, {

deleteProperty(target, property) {

if(!property.startsWith('\_')) {

delete target[property]

return true

}

throw new SyntaxError(`${property} is private`)

}

})

delete user.age

delete user.\_id // Ошибка

**has**

Ловушка has позволяет перехватить вызовы оператор in. Это ловушка позволяет реализовать гибкий диапазон:

let range = {

start: 1,

end: 10

}

range = new Proxy(range, {

has(target, property) {

return property >= target.start && property <= target.end

}

})

console.log(3 in range)

console.log(13 in range)

**BigInt**

Новый тип для представления целочисленных чисел, который не имеет ограничения на длину.

Благодаря BigInt в JS стало возможно легко реализовать вещи, требующие сложные математические операции: криптография, расчет пространства, блокчейн и т.д.

Можно создать через литер, написав в конце n или через BigInt:

let a = 1223134234321n

let b = BigInt('122323234234234234')

let c = BigInt(13325443523234)

В математических операциях нельзя смешить BigInt с другими типами, а также при делении результаты округляются:

console.log(1n + 2n)

console.log(13n / 2n)

**Функция eval**

Функция eval позволяет выполнить строчку js кода:

**let res = eval(code)**

Здесь за res будет принято последнее выражение:

let s = eval('2 + 2')

console.log(s)

В строгом режиме eval имеет собственное лексическое окружение:

eval('let x = 12; x++')

console.log(typeof x)

Использовать eval не рекомендуется, зачастую есть более гибкие решения, например new Function и т.д. Даже есть фразеологизм – «eval is evil»

Единственное место, где eval можно лаконично использовать – это арифметические операции и дебаг.

**Каррирование**

Каррирование – это техника, позволяющая преобразовать sum(a, b) в sum(a)(b).

function curry(fn) {

return function (a) {

return function (b) {

return fn(a, b)

}

}

}

function sum(a, b) {

return a + b

}

let curryFun = curry(sum)

let res = curryFun(12)(100)

console.log(res)

Эта реализация очень простая, но она работает с функцией длина, которая равна двум. Есть и более продвинутая функция:

function makeCurry(func) {

return function curried(...args) {

if (args.length >= func.length)

return func.apply(this, args)

return function (...args2) {

return curried.apply(this, args.concat(args2))

}

}

}

function minus(a, b, c) {

return a - b - c

}

console.log(makeCurry(minus)(12)(24)(9))

console.log(makeCurry(minus)(12)(24, 9))

console.log(makeCurry(minus)(12, 24, 9))

Работает она также очень просто:

1. создается вложенная функция curried, в которой замыкается func
2. в этой функции применяется рекурсия
3. в базовой ветке, когда все нужные аргументы собраны, то вызывается уже сама функция
4. в шаге рекурсии возвращается функция, которая вызовет curried еще раз, чтоб собрать все нужные аргументы

Здесь есть использование замыканий и рекурсии одновременно. Эти две концепции никак друг другу не противоречат.

Этот метод работает даже лучше, так как нет строго разделения по одному аргументу, но, увы, если есть какие-то остаточные параметры, то они будут не учтены.