Практическая работа

Курс: Разработка интерфейса на JavaScript

Дисциплина: Основы JavaScript

Тема занятия№28: Прототипы. Контекст. Виды функций. Замыкания

Создайте объект person с тремя полями.

```
const person = {
  name: 'Maxim',
  age: 25,
  greet: function() {
    console.log('Greet!')
  }
}
```

Если мы выведем их в консоли, то увидим следующее.

```
> person
<- ▶ {name: "Maxim", age: 25, greet: f}
```

То есть у нас есть с вами объект, и мы можем получать доступ к любому ключу.
Теперь попробуем вызвать функцию, которой нет в объекте, то мы получим ошибку (такой функции нет в объекте).

```
person.sayHello()
▶ Uncaught TypeError:
person.sayHello is not a function
    at <anonymous>:1:8
```

А теперь если я попробую вызвать, например функцию toString у объекта person, то никакой ошибки нет, хотя функции тоже нет в объекте.

```
person.toString()
"[object Object]"
```

Почему же так? Так работают прототипы! У прототипа есть функция toString. Если мы выведем в консоли объект person, то увидим, что там есть прототип.

```
person
▼{name: "Maxim", age: 25, greet: f}
    age: 25
  ▶greet: f ()
    name: "Maxim"
  ▶__proto__: Object
```

Это специально свойство, которое является ссылкой на глобальный Объект. Если мы его раскроем, то увидим там метод toString. Когда мы вызывали этот метод мы сначала смотрели на верхнем уровне, если там нет, то уже на нижнем.
Почему у объекта есть свойство прототип? Потому что на сам деле объект создаётся так.

```
const person = new Object()
```

То есть образец глобального Объекта, а та запись — это сокращение.
Что мы можем сделать с прототипами? Например, обратиться к глобальному объекту и у него через прототип создать новую функцию

```
Object.prototype.sayHello = function() {
  console.log('Hello!')
}
```

Теперь этот метод можно вызвать через наш объект person.

```
person.sayHello()
```

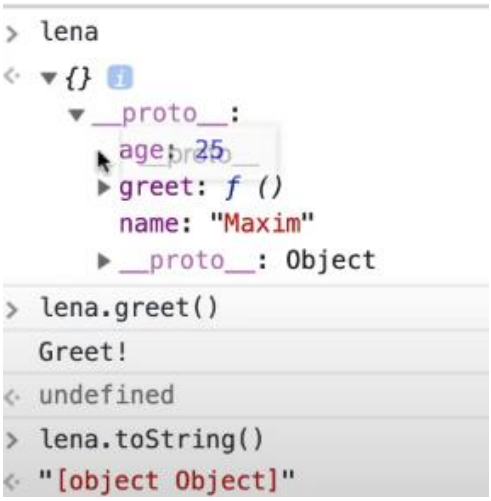
```
Hello!
```

То есть мы расширили глобальный объект. Прототипы так и работают. То есть благодаря им мы можем получать доступ к более расширенным функциям и наследовать от них.

Рассмотрим следующий пример. Создадим объект lena, который в качестве прототипа возьмёт объект person.

```
const lena = Object.create(person)
```

Это объект пустой на верхнем уровне. Но на нижним у него будут значения от объекта person, а ещё ниже от глобального объекта.



Если мы у объекта lena создадим ключ name с другим значением, то при обращении к этому ключу будет брать самое верхнее значение.

Задание: Создайте объект с несколькими ключами и попытайтесь вызвать у него функцию, которой нет у этого объекта, которая выполнит сложение двух чисел.

```
const my_object = {
  name: 'Alex',
  phone: '+432423',
  age: 24
}

Object.prototype.get_sum = function (a, b) {
  return a + b;
}
console.log(my_object.get_sum(5, 4));
```

Вопрос. Сколько типов данных? На самом деле один. Всё в JS является объектами. Сможете доказать это? Просто нужно создать строку и вывести её результат.

```
const str = new String('I am string')
```

Как работает Контекст? Всё очень просто давайте преобразуем вызов функций следующим образом.

```
window.hello()
Hello                                   index.js:2
▶ Window {postMessage: f, blur: f, focus: f
  , close: f, parent: Window, …}
```

Ничего не изменилось. Грубо говоря ключевое слово this, указывает на тот объект, в контексте которого оно было вызвано. То, что находится слева от вызываемой функций или свойства. Поэтому ключевое свойство this всегда динамичное.

Задача. Предположим, что мы хотим создать ещё одну функцию в объекте person, которая будет также ссылать на функцию hello, но контекст будет объект window. (window == this)

```
function hello() {
  console.log('Hello', this)
}

const person = {
  name: 'Vladilen',
  age: 25,
  sayHello: hello,
  sayHelloWindow: hello.bind(window)
}
```

Теперь модернизируем наш объект добавив функцию для вывода имени и возраста. Но вызовем эту функцию для другого объекта.

```
const person = {
  name: 'Vladilen',
  age: 25,
  sayHello: hello,
  sayHelloWindow: hello.bind(document),
  logInfo: function() {
    console.log(`Name is ${this.name}`)
    console.log(`Age is ${this.age}`)
  }
}

const lena = {
  name: 'Elena',
  age: 23
}

person.logInfo.bind(lena)()
```

Модернизируйте функцию logInfo, чтобы туда можно было ещё передать параметры.

```
logInfo: function(job, phone) {
  console.group(`${this.name} info:`)
  console.log(`Name is ${this.name}`)
  console.log(`Age is ${this.age}`)
  console.log(`Job is ${job}`)
  console.log(`Phone is ${phone}`)
  console.groupEnd()
```

```
person.logInfo.bind(lena, 'Frontend', '8-999-123-12-23')()
```

Если использовать метод call, то он вызовет функцию сразу же. Если же использовать функцию apply, то идёт всего два параметра. Второй это массив с параметрами для функции. Модернизируйте функцию добавив третий параметр, написав через apply.

Задача.

Комбинирования прототипа и контекста. Написать функцию, которая умножит каждое число массива на какое-то число.

```
const array = [1, 2, 3, 4, 5]

function multBy(arr, n) {
  return arr.map(function(i) {
    return i * n
  })
}

console.log(multBy(array, 15))
```

```
Array.prototype.multBy = function(n) {
  return this.map(function(i) {
    return i * n
  })
}

console.log(array.multBy(20))
```

Замыкания.

Рассмотрим пример функции внутри другой функции.

```
function createCalcFunction(n) {
  return function() {
    console.log(1000 * n)
  }
}

createCalcFunction(42)
```

Но ничего не выведет. Потому что первая функция возвращает новую функцию. Поэтому можно занести результат в переменную и вызвать эту переменную как функцию.

```
function createCalcFunction(n) {
  return function() {
    console.log(1000 * n)
  }
}

const calc = createCalcFunction(42)
calc()
```

Когда вызывали 1 функцию и передали туда число 42. Она отработала и вернула новую функцию. А так как 2 функцию в контексте 1 функции, то переменная N как бы замкнута там. Значение N уже известно для 2 функции.

Задача. Написать через замыкания функцию, которая всегда прибавляет определённое число.

```
function createIncrementor(n) {
  return function(num) {
    return n + num
  }
}

const addOne = createIncrementor(1)
const addTen = createIncrementor(10)

console.log(addOne(10))
console.log(addOne(41))

console.log(addTen(10))
console.log(addTen(41))
```