

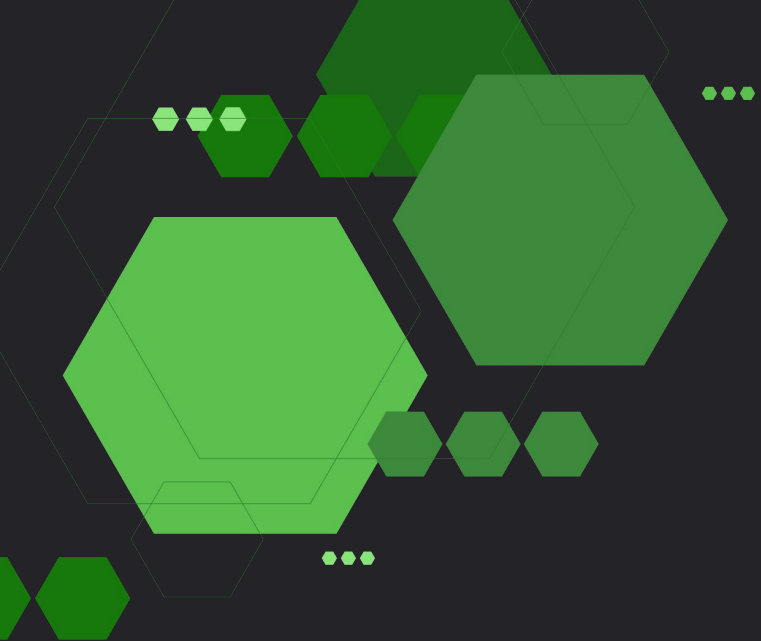
SKILLFACTORY



Продвинутая работа с функциями

Токаревская Юлия

Frontend-разработчик в Socialbakers



Замыкания

Что такое замыкания?

Мы знаем, что функция может получить доступ к переменным из внешнего окружения. Но что произойдёт, если внешние переменные изменятся? Функция получит последнее значение или то, которое существовало на момент создания функции?

```
let string = "Hello";

function printString() {
  console.log(string);
}
```

```
string = "World";
```

```
printString();
```

```
// "World"
```

```
function makePrintFunc() {
  let string = "Hello";
  return function() {
    console.log(string);
  }
}
```

```
let string = "World";
```

```
let print = makePrintFunc();
print();
```

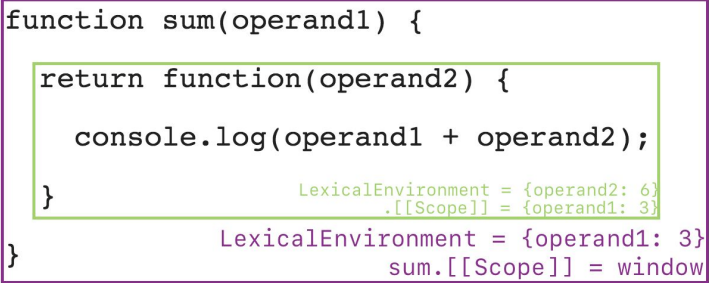
```
// "Hello"
```

Лексическое окружение

Лексическое окружение - это статическая область в JavaScript, которая определяет, к каким переменным, функциям и объектам мы можем иметь доступ, основываясь на их расположении в коде.

При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова.

```
01. function sum(operand1) {  
02.     return function(operand2) {  
03.         console.log(operand1 + operand2);  
04.     }  
05. }  
06.  
07. let plus3 = sum(3);  
08. plus3(6); // 9
```



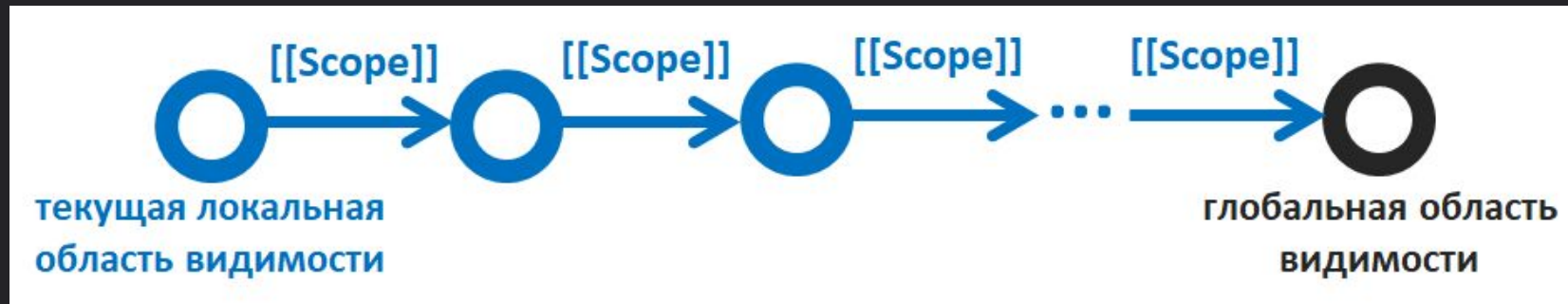
Примечание

Лексическое окружение есть не только у функций, но и у любых блоков кода, ограниченных {}, например, блоки условия, блоки циклов и т.д.

Лексическое окружение

Объект лексического окружения состоит **из двух частей**:

1. **Environment Record (запись окружения)** – объект, в котором хранятся все локальные переменные
2. **Ссылка на внешнее лексическое окружение** – то есть то, что находится снаружи от текущих фигурных скобок



Практические задания на замыкания

Задача 1:



Написать функцию, которая считает и выводит в консоль количество своих вызовов

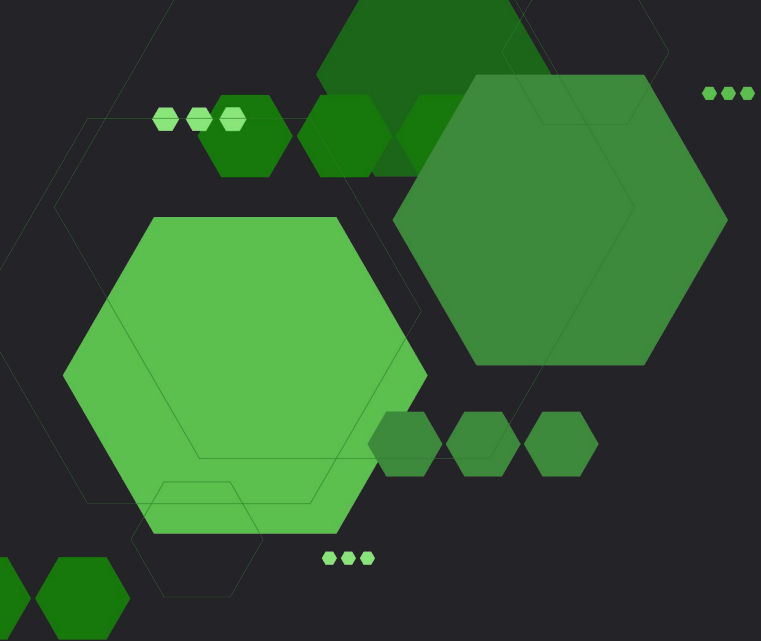
Задача 2:



Найти и исправить ошибку в коде:

```
let fruits = ['Banan', 'Apple', 'Mango', 'Kiwi'];

function printFruits(fruits) {
  let i = 0;
  while (i < fruits.length) {
    setTimeout(() => {
      console.log(fruits[i]);
    }, 1000);
    i++;
  }
}
```



Рекурсия

Что такое рекурсия?

Рекурсия - это функция, которая сама вызывает себя (как правило, с другими аргументами).

Любая задача, которую можно реализовать с применением рекурсии, также можно реализовать и итеративно (циклами). Нет никакого выигрыша в производительности от использования рекурсии.

Порой итеративный подход с циклами будет работать даже быстрее, но простота и наглядность, с которой могут быть выполнены вычисления с помощью рекурсии, иногда превосходит недостатки, связанные с перерасходом ресурсов при повторных вызовах функции.

Что нужно помнить при использовании рекурсии?

- Наилучшее применение рекурсии – это решение задач, для которых свойственна следующая черта: решение задачи в целом сводится к решению **подобной же задачи**, но меньшей размерности.
- Общее количество вложенных вызовов называют **глубиной рекурсии**. Максимальная глубина рекурсии в браузерах ограничена 10 000 рекурсивных вызовов.
- Значение, на котором рекурсия заканчивается, называют **базовый случай** или **терминальный сценарий**.
- Большое количество рекурсивных вызовов в функции может привести к **переполнению стека**. Поскольку местом для хранения параметров и локальных переменных функции является стек, а каждый новый вызов создает новую копию переменных, пространство стека может исчерпаться.

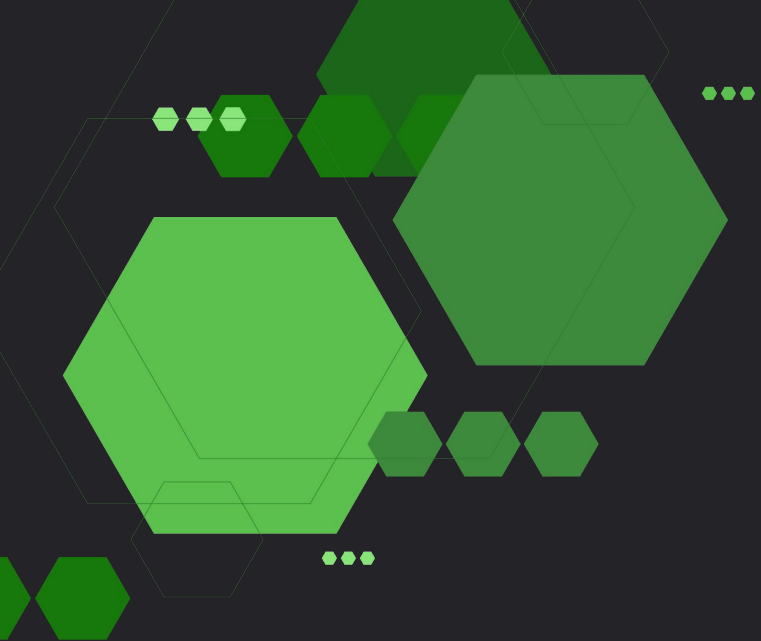
Практическое задание на рекурсию

Задание:



Дана последовательность натуральных чисел от N до 0. Напишите код, который запрашивает число N и выводит все нечетные числа из этой последовательности по одному в консоль, сохраняя их порядок. Например:

$N = 8 \rightarrow 7\ 5\ 3\ 1$



Свойство
arguments

Свойство arguments

Свойство arguments

В Javascript возможно вызвать функцию с любым количеством аргументов, независимо от того, как она была определена.

```
function sum(a, b) {  
    return a + b;  
}  
sum(3, 5, 9, 15); // 8
```

Дополнительные аргументы не вызовут ошибку, но и не будут учтены при выполнении функции

Свойство arguments

У каждой функции есть переменная **arguments**, в которой хранятся все аргументы функции под своими порядковыми номерами.

Важно: arguments - псевдомассив!

```
function sum() {  
    let result = 0;  
  
    for (let number of arguments) {  
        result += number;  
    };  
    console.log(result);  
}  
sum(3, 5, 9, 15); // 32
```

Оператор расширения

Остаточные параметры могут быть обозначены через **оператор расширения** - ...

Мы можем положить первые несколько параметров в переменные, а остальные – собрать в массив, или же положить в массив все параметры (в таком случае получится почти то же самое, что и при использовании arguments).

```
function test(...allParams) {  
    // ...  
}  
  
test(1, 2, 3, 4, 5);
```

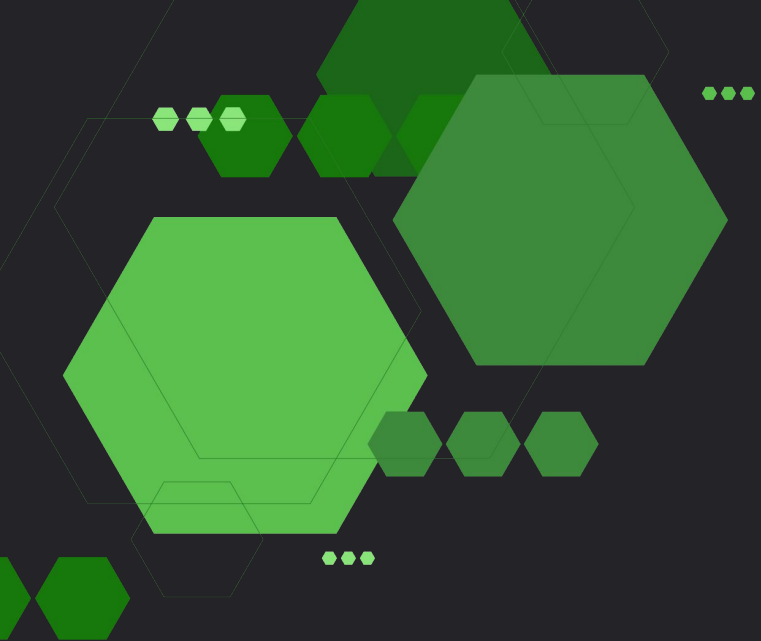
```
function test(param1, param2, ...allParams)  
{  
    // ...  
}  
  
test(1, 2, 3, 4, 5);
```

Какой вариант лучше использовать?

У использования arguments есть несколько недостатков:

- arguments - псевдомассив. Он не поддерживает методы массивов, поэтому мы не можем, например, вызвать arguments.map(...).
- arguments всегда содержит все аргументы функции — мы не можем получить их часть. А остаточные параметры позволяют это сделать.
- Стрелочные функции не имеют arguments

Таким образом, остаточные параметры - более гибкий инструмент для работы с аргументами.



**Привязка
контекста this**

Привязка контекста this

При передаче методов объекта в качестве колбэков, например для `setTimeout`, возникает известная проблема – потеря **this**.

Вот как это может произойти в случае с `setTimeout`:

```
let user = {
  firstName: "Ivan",
  sayHi: function() {
    console.log(`Hello, ${this.firstName}!`);
  }
};

setTimeout(user.sayHi, 1000); // Hello, undefined!
```

Привязка контекста this

Вариант 1

```
let user = {
  firstName: "Ivan",
  sayHi: function() {
    console.log(`Hello,
${this.firstName}!`);
  }
};

setTimeout(function() {
  user.sayHi(); // Hello, Ivan!
}, 1000);
```

Вариант 2

```
let user = {
  firstName: "Ivan",
  sayHi: function() {
    console.log(`Hello,
${this.firstName}!`);
  }
};

let bindedSayHi =
user.sayHi.bind(user);
bindedSayHi(); // Hello, Ivan!
setTimeout(bindedSayHi, 1000); //
Hello, Ivan!
```

Метод **bind()**

bind() - встроенный метод у функций, который позволяет зафиксировать this. Метод bind() создаёт новую функцию, которая при вызове устанавливает в качестве контекста this предоставленное значение. В метод также передаётся набор аргументов, которые будут использованы в привязанной функции при её вызове.

```
let boundFunc = func.bind(context, [arg1], [arg2], ...);
```

- **context** - значение, передаваемое в качестве this в целевую функцию
- **arg1, arg2, ...** - аргументы целевой функции (необязательный параметр)

Привязка контекста `this`

Частичное применение функций

bind() может использоваться не только для привязывания `this`, но и в некоторых других ситуациях.

Например, для частичного применения функций: мы можем создать новую функцию на основе существующей, зафиксировав некоторые из существующих параметров.

В данном случае мы не будем использовать `this`. Но для `bind` это обязательный параметр, так что мы должны передать туда какое-нибудь значение, например, `null`.

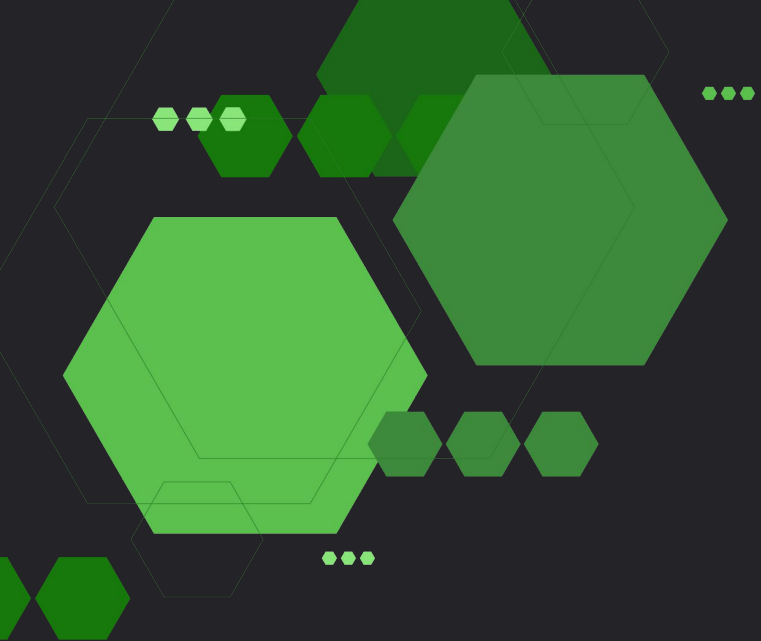
Практическое задание на bind

Задание:



Дана функция `askPassword()`, которая должна проверить пароль и затем вызвать `user.loginOk`/`user.loginFail` в зависимости от ответа.

Однако, вызов функции приводит к ошибке.
Нужно найти её и исправить



**Использование
call/apply**

Метод call()

Метод **call()** вызывает функцию с указанным значением `this` и предоставленными аргументами.

```
func.call(context, arg1, arg2, ...)
```

- **context** - значение, передаваемое в качестве `this` в целевую функцию
- **arg1, arg2, ...** - аргументы целевой функции (необязательный параметр)

Метод apply()

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

```
func.apply(context, argsArray)
```

- **context** - значение, передаваемое в качестве `this` в целевую функцию
- **argsArray** - массив с аргументами целевой функции (необязательный параметр)

По функционалу метод полностью идентичен методу **call**, фундаментальное различие между ними заключается в том, что функция `call()` принимает список аргументов, в то время как функция `apply()` принимает единственный массив аргументов.

Случаи применения call/apply

Использование call для связи конструкторов объекта в цепочку

```
function Product(name, price) {  
  this.name = name;  
  this.price = price;  
}
```

```
function Food(name, price) {  
  Product.call(this, name, price);  
  this.category = 'Food';  
}
```

```
Food.prototype = new Product();
```

Случаи применения call/apply

Заимствование метода

Используется в случаях, когда метод, который мы хотим использовать, уже определен в другом объекте. Мы можем вызвать call/apply на методе другого объекта и передать в качестве контекста наш объект

Патчинг JS-библиотек

Вы можете использовать call/apply, чтобы скорректировать поведение функции, к коду которой у вас нет доступа (например, она берется из внешней JS-библиотеки).

call/apply/bind - что использовать?

- **bind()** - возвращает копию функции и принимает в качестве параметра объект, который будет использован в качестве this при вызове функции. В качестве дополнительных параметров можно передать аргументы функции.
- **call()** - делает почти то же самое, что и bind(), но вызывает функцию сразу вместо создания копии.
- **apply()** - работает точно так же, как call(), единственное отличие состоит в том, что аргументы функции передаются в виде массива.