# Более сложные функции

IT SHATLE

# Рекурсия

Рекурсия – это приём программирования, полезный в ситуациях, когда задача может быть естественно разделена на несколько аналогичных, но более простых задач. Или когда задача может быть упрощена до несложных действий плюс простой вариант той же задачи.

# Рекурсия

Рекурсивной функцией (recursive function) обычно называют функцию, которая вызывает сама себя, например:

```
function factorial(num) {

if (num <= 1) {

  return 1;

} else {

  return num * factorial(num-1);
}
```

# Задача

Создать функцию, которая которая возводит x в степень n.

# Контекст выполнения, стек

Контекст выполнения – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this и прочую служебную информацию.

Когда функция производит вложенный вызов, происходит следующее:

- Выполнение текущей функции приостанавливается.
- Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – стеке контекстов выполнения.
- Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
- После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

# Замыкание

Замыкание (closure) - это функция, которой доступны переменные из области видимости другой функции. Обычно для создания замыкания одну функцию определяют внутри другой:

```
function createComparisonFunction(propertyName) {

        return function(objectl, object2){

                let valuel objectl[propertyName] ;

                let value2 object2[propertyName] ;

                if (valuel < value2){ return -1;

                } else if (valuel > value2){ return 1;

                 } else { return 0;}};}
```

IT SHATLE

# Замыкание

Замыкание всегда получает последнее значение любой переменной из внешней функции.

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment.

Все функции «при рождении» получают скрытое свойство [[Environment]], которое ссылается на лексическое окружение места, где они были созданы.

# Замыкание

Пример:

```
function makeCounter() {

  let count = 0;

  return function() {   return count++;};  }

let counter = makeCounter();

let counter2 = makeCounter();

alert(counter());

alert(counter2());
```

# setTimeout

setTimeout позволяет вызвать функцию один раз через определённый интервал времени.

```
1)  function sayHi(phrase, who) {
 alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Привет", "Джон");
```

```
2) setTimeout(() => alert('Привет'), 1000);
```

# clearTimeout

```
let timerId = setTimeout(...);

clearTimeout(timerId);
```

Вызов setTimeout возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения.

# setInterval

```
// повторить с интервалом 2 секунды

let timerId = setInterval(() => alert('tick'), 2000);


// остановить вывод через 5 секунд

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

# Задача 1

Напишите функцию printMessage(n), которая выводит сообщения на экран каждые n секунд.

IT SHATLE

# Задача 2

Напишите функцию, которая выводит через 5 секунд на экран сообщение “прошло 5 секунд”

IT SHATLE

# Задача 3

Напишите функцию printNumbers(from, to), которая выводит число каждую секунду, начиная от from и заканчивая to.

IT SHATLE

# Задача 4

По нажатию на кнопку показывать сообщение через 5 секунд, во время этих 5 секунд должно появляться слово “loading”, а после появления сообщения - скрываться

# call, apply

У функций также есть методы apply() и call().Оба они вызывают функцию с конкретным аргументом this, задавая значение объекта this в теле функции. Метод apply ( ) принимает два аргумента: значение this внутри функции и массив аргу ментов, который может быть экземпляром Array или объектом arguments.

Позволяют явно  указать, передать контекст функции

```
sum.apply(this, arguments);
```

```
sum.call(this, numl, num2);
```

# call

```
function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// используем 'call' для передачи различных объектов
в качестве 'this'

sayHi.call( user ); // John
sayHi.call( admin ); // Admin
```

Он запускает функцию func, используя первый аргумент как её контекст this, а последующие – как её аргументы.

# apply

Он выполняет func, устанавливая this=context и принимая в качестве списка аргументов псевдомассив args.

Единственная разница в синтаксисе между call и apply состоит в том, что call ожидает список аргументов, в то время как apply принимает псевдомассив.