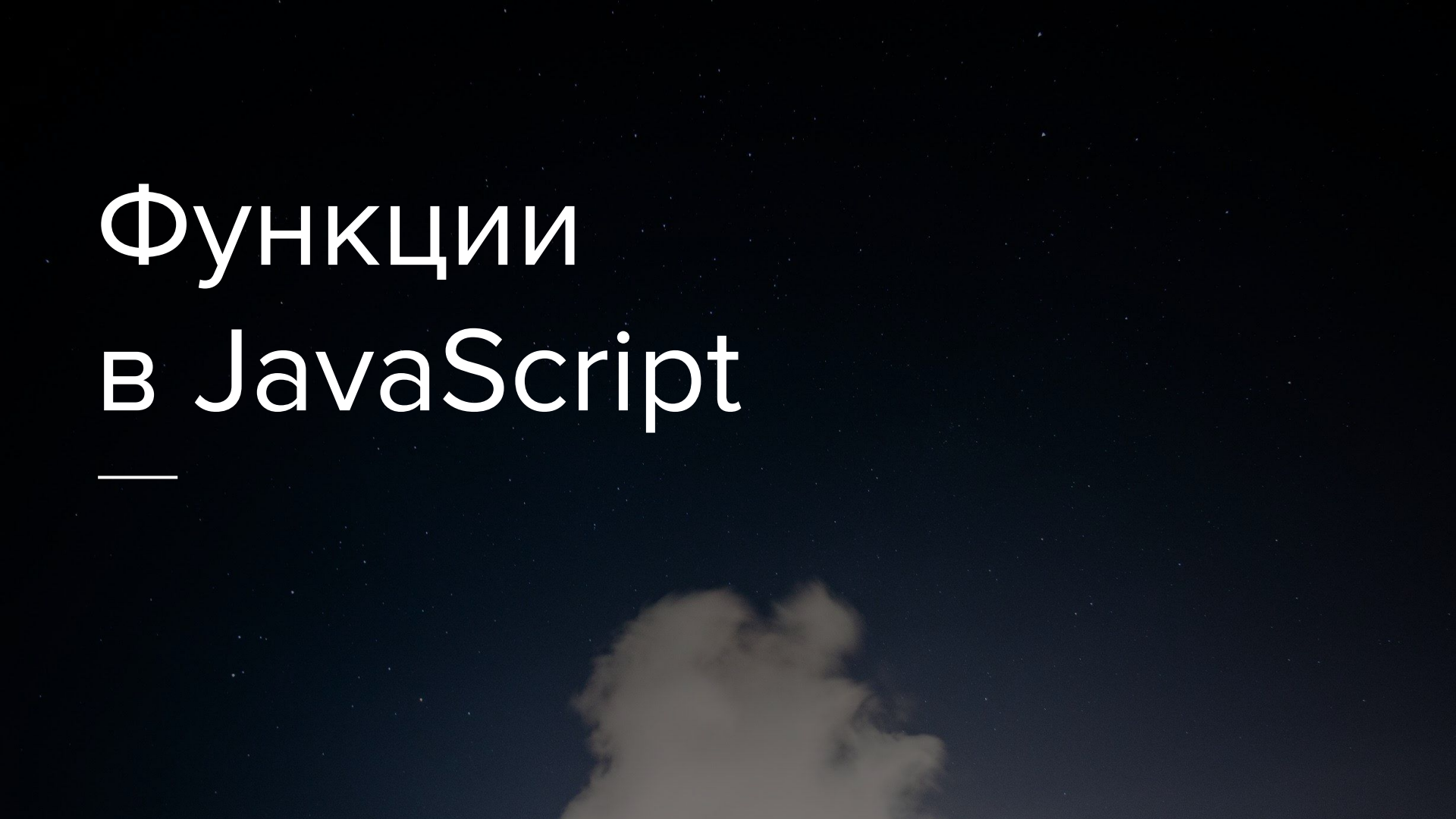
Функции в JavaScript



Функция

В общем смысле, это "подпрограмма", которую можно вызывать из внешнего (или внутреннего, в случае рекурсии) по отношению к функции кода.

Функции позволяют избежать избыточности кода, тк функцию создают один раз, а вызывать ее на выполнение можно многократно из разных точек программы.

Часто используемые функции можно объединять в библиотеки.

Функция

Функция состоит из последовательности инструкций, называемой телом функции.

Значения могут быть переданы в функцию,
а функция вернёт значение.

В JavaScript функции являются объектами - Function

Именованные функции
Function Declaration

Именованные функции доступны везде в области видимости т.е к ней можно обращаться и до ее создания и после, тк они создаются интерпретатором до выполнения кода.

Объявление функции

```
function имяФункции(параметр1, ..., параметрN="значение по-умолчанию"  ) {
    тело функции;
    [return;]
}
```

Вызов функции - последовательное выполнение кода из тела функции

```
имяФункции(значение1, ..., значениеN);
```

Функции в блоке

При использовании строгого стандарта ‘use strict’;, функция, объявленная в блоке, работает только в блоке. Вне блока ее вызов приведет к ошибке.

```
'use strict';
if (true) {
      function printHello(name) {
          console.log('Hello ' + name);
      }
     printHello('Grisha');  //работает Hello Grisha;
}
printHello(); // ошибка: Uncaught ReferenceError:
                  // printHello is not defined at arrays.js:9
```

На **имена функций** распространяются те же правила, что и на имена переменных. Имя функции должно быть глаголом и отображать то, что она делает, т.к. функция – это действие.

Функция не должна выполнять больше одного действия.

Возвращаемое значение RETURN

Функция **всегда возвращает значение**.
Инструкция **return указывает**, **что** именно **должна вернуть функция**. После того, как **отработает return функция завершается**, код после return выполнен не будет. Функция **может вернуть только одно значение**.
Если return не указан, функция вернет значение по-умолчанию:

- undefined;
- значение this (если функция была создана через оператор new)

Параметры функции

При вызове функции ей можно передать данные, которые та использует в зависимости от ее логики.

```
function printHello(name) {
    console.log('Hello ' + name);
}
printHello('Grisha');    // Hello Grisha;
```

Функцию можно вызвать с любым количеством аргументов.
При объявлении функции необязательные аргументы, как правило, располагают в конце списка.

Если параметр не передан при вызове – он считается равным undefined.

Параметры по умолчанию

можно задать в теле функции:

```
function printUsersInfo(name, surname) {
        surname = surname || 'фамилия не известна';
        console.log('Пользователь: ' + name + ' ' + surname);
}
printUsersInfo('Grisha');
```

Нельзя задать по умолчанию тип данных. Проверку на тип данных нужно делать внутри функции.

а можно “**возможность ES6/ES2015**”

```
function printUsersInfo(name, surname='фамилия не известна') {
        console.log('Пользователь: ' + name + ' ' + surname);
}
printUsersInfo('Grisha');
```

Параметры по умолчанию могут быть не только значениями, но и выражениями.

Переменное количество аргументов ARGUMENTS

В JavaScript любая функция может быть вызвана с произвольным количеством аргументов.

Все значения, переданные в функцию при вызове, находятся в объекте arguments (не имеет значения, есть ли они списке параметров и используются ли они в теле функции).

Доступ к элементам arguments осуществляется через [ ], arguments можно перебрать в цикле for. Но массивом arguments не является!

Оператор spread вместо arguments

“возможность ES6/ES2015”

Чтобы получить именно массив аргументов можно использовать оператор ... .

```
function printUsersInfo(name, surname, ...other_info) {
  console.log('Пользователь: ' + name + ' ' + surname + ' ' + other_info);
}
printUsersInfo('Гриша', 'Петров', 26, '+79991234455');
// Пользователь: Гриша Петров 26,+79991234455
```

В other_info попадёт массив всех аргументов, начиная с третьего.
other_info – настоящий массив, со всеми доступными массивам методами, в отличие от arguments.
Spread должен быть в конце, тк в него попадают оставшиеся аргументы

Оператор spread при вызове функции

используется **для передачи массива параметров** как списка.

“возможность ES6/ES2015”

```
function sum(x, y, z) {
        return x + y + z;
}

let numbers = [1, 2, 3];

let result = sum(...numbers);

console.log(result);  // 6
```

Объявление функции Function Expression

Функции могут быть созданы внутри выражения, их можно присвоить переменной. Такие функции как правило анонимны.
Такие функции **можно вызвать только после объявления**
Объявление анонимных функций.

```
let имяПеременной = function(парам1, ... , парамN=значение по умолчанию) {
    тело функции;
    [return;]
};
```

Вызов функции

```
имяПеременной(значение1, ..., значениеN);
```

Функции через =>

Вместо

```
let plus = function(a, b) {
        return a + b;
};
plus(2,4);
```

можно записать **“возможность ES6/ES2015”**

```
let plus = (a, b) => a+b;

console.log(plus(2,4));
```

Функции через =>

let plus = (a, b) => a+b;

Слева от => находится аргумент,
справа – выражение, которое нужно вернуть.

Если **аргумент один**, скобки использовать не нужно, если **аргументов несколько**, их оборачивают в скобки.

Если нужно задать функцию **без аргументов**, то используются пустые скобки:

```
let random = () => Math.random();
```

Когда **тело функции достаточно большое**, то можно его обернуть в фигурные скобки {...}. Такая функция должна делать явный return.

Функции-стрелки **не имеют своего this.**

Функции-стрелки **не имеют своего arguments.**

Рекурсия

В теле функции могут быть вызваны другие функции для выполнения подзадач.

Рекурсия – когда **функция вызывает сама себя** (как правило, с другими аргументами).

Глубина рекурсии - общее количество вложенных вызовов.

Максимальная глубина рекурсии в браузерах ограничена - **10000 вложенных вызовов**.

Например, необходимо вывести на консоль числа от 0 до n в обратном порядке.

Способ без рекурсии:

```
function writeN(n) {
  for (; n >=0; n--) {
   console.log(n);
}
```

Рекурсивное решение:

```
function writeN(n) {
  if(n >= 0) {
    console.log(n);
    write(n - 1);
  }
}
```

Замыкания

Функция, объявленная внутри другой функции получает доступ к переменным и аргументам последней.
Такие переменные продолжают существовать и остаются доступными внутренней функцией даже после того, как внешняя функция, в которой они определены, была исполнена.

```
function имяФункции1(параметр1) {
  var переменная;
  function имяФункции2() {
    //имеет доступ к параметр1 и переменная
  }
}
```

Такие вложенные функции называют замыканиями, они замыкают на себя переменные и аргументы функции, внутри которой определены.

Замыкания

Пример 1, функция возвращает количество собственных вызовов:

```
function createCounter() {
  var numberOfCalls = 0;
    return function() {
        return numberOfCalls++; // запоминает
    };
}

var count = createCounter();

console.log( count() ); // 1
console.log( count() ); // 2
console.log( count() ); // 3
```

Замыкания

Пример 2:

```
let printHelloFunction = function(name) {
  return function() {
     console.log('Hello, ' + name);
  }
};
let greetingLelik = printHelloFunction('Lelik');
greetingLelik();
```