Функции

Мікалай Янкойць

Адукар

Повестка дня

1. Что такое функции
2. Контекст выполнения. Области видимости
3. Всплытие
4. Стек. Рекурсия
5. Управление памятью

Что такое функции и зачем они нужны

Функции – обособленные блоки кода, которые можно вызывать из других мест программы (как правило, по осмысленному имени).

- Избавляют от дублирования кода
- Позволяют «расширять лексикон» языка программирования
- Помогают структурировать код и упрощать его внутреннюю логику
- Создают новые возможности (например, рекурсивный вызов, обратный вызов)

Объявление функции / function declaration

```
1  function sayHello() {
2      console.log("Hello World!");
3  }
4
5  sayHello();
6  sayHello();
```

sayHello – имя функции

Код в { скобках } – тело функции

sayHello(); – вызов функции

Функция выполняется только тогда, когда её вызывают!

Аргументы / параметры

```
1  function showSquare(x) {
2      console.log(x*x); // x – обычная переменная!
3  }
4
5  showSquare(3); // говорим, что переменная x = 3
6  // в консоль выводится 9
7
8  function sayHiToUser(firstName, secondName) {
9      console.log("Привет, " + userName +
10                     " " + secondName + "!");
11 }
12
13 sayHiToUser("Аня", "Иванова");
14     // "Привет, Аня Иванова!"
```

Возврат значения

Функция может возвращать результат с помощью директивы return

```
1  function cube(x) {
2      return x*x*x;
3  }
4
5  let cubeOfSix = cube(6);
6  console.log(cubeOfSix); // 216
7
8  console.log(cube(5)); // 125
```

Ещё про возврат значения

```
1  function checkAge(age) {
2      if (age < 18) {
3          return "Прости, ты слишком молод";
4      } else {
5          return "Добро пожаловать!";
6      }
7      console.log("Эта строка никогда не выполнится!");
8  }
9
10 console.log(checkAge(17));
11 console.log(checkAge(33));
```

- В одной функции может быть несколько return
- Дойдя до return, функция прекращает выполнение

И ещё про возврат значения

```
1  function doSomething() {
2      /* ... */
3      return;
4  }
5
6  function doSomethingOther(a, b) {
7      a = a**b;
8      b = b % 2;
9  }
10
11 console.log(doSomething()); // undefined
12 console.log(doSomethingOther(4, 6)); // undefined
```

- return; – то же самое, что return undefined;
- Отсутствие return – то же, что return; в последней строке тела функции

Функциональное выражение / function expression

```
1  const sayHi = function() {
2      console.log("Hello World!");
3  }
4  sayHi(); // Hello World!
5
6  let getLarger = function(a, b) {
7      if (a > b)
8          return a;
9      return b;
10 }
11 console.log(getLarger(7, 12)); // 12
12
13 getLarger = "а теперь здесь строка";
14 getLarger(5, 3); // ошибка!
```

Функция – обычное значение переменной, как строка или число!

И объявлять её можно, как обычную переменную.

Arrow functions*

```
1  let square = x => x*x;
2  // почти то же самое, что:
3  let square = function(x) { return x*x; }
4
5  let add = (a, b) => a+b;
6  // почти то же самое, что:
7  let add = function(a, b) { return a+b; }
8
9  let randomTil100 = () => Math.random()*100;
10 // почти то же самое, что:
11 let randomTil100 = function() {
12                          return Math.random()*100; }
```

*позже тема будет разобрана подробно

Параметры по умолчанию

```
1  function power(base = 2, exp = 2) {
2      return base ** exp;
3  }
4
5  power(3, 3);           // 3^3 = 27
6  power(6);              // 6^2 = 36
7  power();               // 2^2 = 4
8  power(undefined, 8);   // 2^8 = 256
```

- Значения по умолчанию устанавливаются после = в списке аргументов
- Не передать последние аргументы – то же, что передать в них undefined

Практика

1. Напишите функцию, которая получает три числа и возвращает их сумму.
2. Напишите функцию, которая подсчитывает сумму чисел от 1 до заданного X.
3. Напишите функцию, которая подсчитывает сумму цифр числа.
4. Напишите функцию, которая считает факториал числа.

Контекст выполнения / execution context

Код всегда выполняется в некотором **контексте** – окружении, хранящем все параметры, необходимые для правильной работы программы.

Каждый вызов функции создаёт специальный контекст, в котором хранятся все доступные переменные, аргументы функции, **области видимости** и другие полезности.

Код вне функций выполняется в глобальном контексте (единственном!)

Область видимости / scope

У каждой переменной есть область видимости – «граница», только внутри которой можно получить доступ к этой переменной.

```
1  let x = "Я глобальная";
2
3  function someFunc() {
4      console.log(x); // работает!
5      let y = "А я локальная";
6      console.log(y); // работает!
7  }
8
9  someFunc();
10 console.log(y);      // ошибка!
```

Локальные и глобальные переменные

Переменные, объявленные в основном теле программы – **глобальные**, их область видимости тоже называется глобальной. Такие переменные доступны в любом месте программы.

Переменные, объявленные внутри функций – **локальные**, в локальной области видимости. Вне функций они недоступны.

Область видимости: разница между let и var

```
1  for (var i = 1; i < 10; i++) {
2      console.log(i);
3  }
4  console.log(i); // работает, выдаёт 9
5  { var oldSchool = "я в блоке, но глобальная"; }
6  console.log(oldSchool); // работает!
7  function testVar() {
8      var test = "я объявлена в функции";
9  }
10 testVar();
11 console.log(test);  // ошибка!
```

При объявлении переменной через var её область видимости – либо глобальная, либо ограниченная функцией.

Область видимости: разница между let и var

```
1  for (let i = 1; i < 10; i++) {
2      console.log(i);
3  }
4  console.log(i); // ошибка!
5  { let newSchool = "я в блоке, и я локальная"; }
6  console.log(newSchool); // ошибка!
7  function testLet() {
8      let test = "я объявлена в функции";
9  }
10 testLet();
11 console.log(test);  // ошибка!
```

При объявлении переменной через let (или const) её область видимости может быть локально ограничена циклом, блоком или функцией.

Ещё о локальных и глобальных переменных

```
1  var old = 7;
2  function joke() {
3      console.log(old);
4      var old = 5;
5  }
6  joke();              // undefined!
7  console.log(old);    // 7
8
9  {
10     console.log(old); // ошибка!
11     let old = 3;
12 }
```

Будьте внимательны
с именами переменных
и их областями видимости!

Старайтесь использовать let,
а не var, если это возможно.

Инициализация скрипта. Всплытие / hoisting

Переменные, объявленные через var, и функции, объявленные через declaration, инициализируются до выполнения первой строки!

```
1  console.log(doSomething()); // работает!
2  console.log(x); // не ошибка, но undefined
3      // значит, x существует, просто без значения
4
5  var x = 5; // значение x получит здесь
6
7  function doSomething() {
8      return "я работаю везде";
9  }
```

Стек вызовов / call stack

При каждом вызове функции её контекст выполнения сохраняется в специальной структуре – стеке вызовов.

```
1  function stackIt() {
2      console.log("тут вывод в консоль"); // уровень 3
3  }
4
5  function callFirst(param) {
6      stackIt(); // уровень 2
7      console.log("тут выводим z: " + param); // уровень 2
8  }
9
10 // начало выполнения
11 // уровень 0: глобальный контекст
12 let z = 1;
13 callFirst(z); // уровень 1
```

```
global                  // 0
    callFirst           // 1
        stackIt         // 2
            console.log // 3
        stackIt         // 2
    callFirst           // 1
        console.log     // 2
    callFirst           // 1
global                  // 0
```

Ещё про стек вызовов

- Стек вызовов работает по принципу LIFO (last in, first out). Контекст только что вызванной функции сохраняется **на вершине стека**, а после выполнения управление **всегда** возвращается на **предыдущий** уровень.
- В контексте выполнения функции сохраняется её точка вызова. При возврате управления код продолжит выполняться **с той же точки**.
- Вложенная функция «видит» локальные переменные «родительской» функции, находящейся на более низком уровне стека.
- Максимальное количество уровней стека зависит от интерпретатора (а значит, может быть разным в разных браузерах). На 10 тысяч уровней можно рассчитывать!

Рекурсия

Функция может вызывать саму себя. Такой вызов называется рекурсивным.

```
1  function recursivePower(base, exp) {
2      if (exp == 1)   // 1 - базис рекурсии
3          return base;
4      return base * recursivePower(base, exp-1);
5  }
6
7  recursivePower(2, 10); // 1024
```

Значение, на котором рекурсия заканчивается – базис.
Общее количество вложенных вызовов – глубина рекурсии.

Управление памятью в JavaScript

Главная концепция – **принцип достижимости.**

Гарантированно хранятся в памяти **корни:**

- все глобальные переменные
- все значения из стека вызова (т.е. локальные переменные и аргументы функции, которая выполняется, и всех функций "в режиме ожидания")

Все остальные значения хранятся только до того момента, пока они достижимы из корней.

Недостижимые значения автоматически удаляются сборщиком мусора (Garbage collector).

Внеклассное чтение

https://learn.javascript.ru/function-basics

https://www.w3schools.com/js/js_let.asp

https://www.w3schools.com/js/js_const.asp

https://learn.javascript.ru/recursion

https://learn.javascript.ru/function-expressions*

*позже тема будет разобрана подробно