Функции

Функции как тип данных

- Являются отдельным объектным типом данных
  - Могут иметь собственные свойства
  - Наследуют методы `Object.prototype` и добавляют свои
- First-class citizen
  - Могут присваиваться переменным
  - Могут передаваться как аргументы и результат работы функции
- Определяют область видимости

Создание функций: declaration vs expression

Function Declaration

- Иструкция объявления функции
- Находится в основном потоке кода

```
add(1, 2); // => 3
function add(a, b) {
   return a + b;
}

// Function declaration всплывает наверх
function add(a, b) {
   return a + b;
};
add(1, 2); // => 3, вызов существующей функц
```

Function Expression

- Литерал/анонимная функция
- Находится в контексте выражения

```
add(1, 2); // ReferenceError
let add = function(a, b) {
   return a + b;
};

add(1, 2); // add находится в TDZ, ошибка
// Переменная add объявляется
// и инициализируется функцией только здесь:
let add = function(a, b) {
   return a + b;
};
```

Создание функций: declaration vs expression

Function Declaration

- Можно вызвать до объявления
- Можно объявить только в основном потоке кода (не внутри блоков `if/while/etc`) (*strict*)

```
'use strict';
let flag = true;
if (flag) {
   function checkFlag() {
       console.log('flag is true');
   }
} else {
   function checkFlag() {
       console.log('flag is false');
   }
}
checkFlag(); // ReferenceError
```

Function Expression

- Нельзя вызвать до объявления
- Можно объявить там же где и любой другой тип данных - внутри блоков, inline

```
let flag = true, checkFlag;
if (flag) {
   checkFlag = function() {
       console.log('flag is true');
   }
} else {
   checkFlag = function() {
       console.log('flag is false');
   }
}
checkFlag(); // 'flag is true'
```

Создание функций: Named Function Expression

- NFE - это Function Expression с указанным именем функции
- Имя NFE доступно только внутри этой функции
- NFE всегда объявляется в контексте выражения

```
function f(n) {
    return n ? n * f(n - 1) : 1;
};
f(5); // 120

let g = f;
f = null;
g(5); // TypeError: f is not a function
```

```
let f = function factorial(n) {
    return n ? n * factorial(n - 1) : 1;
};

let g = f;
f = null;
g(5); // 120
factorial(5); // ReferenceError
```

```
function g() { return 1; }
g();        // => 1

(function g() { return 1; });
g();        // Reference Error: g is now defined!
```

Создание функций: методы объектов

- Метод объекта - обычное свойство, значением которого является функция вместо данных
- `propName: function(...) {...}` - присвоение function expression свойству
- `propName: function fName(...) {...}` - присвоение NFE свойству
- `propName(...) {...}` - более краткая нотация для методов
  - не могут использоваться как конструкторы (`new f(...)`)
  - можно использовать ключевое слово `super` для ссылки на прототип

```
let obj = {
    fa: function() { ... },
    fb: function fb() { ... },
    fc() { ... }
}

obj.fd = function() { ... };
```

Создание функций: конструктор `Function`

- А я разве уже говорил, что функции - это объекты?
- Можно объявить функцию "на лету", когда ее код заранее неизвестен
- Всегда создается глобальная функция, без учета локальной области видимости

```
let body = 'return a + b;';
var add = new Function('a, b', body);
add(1, 2); // 3
```

Создание функций: Arrow notation

- Краткая форма записи для inline-функций
- Синтаксис `(param1, param2, …, paramN) => { statements }`
- Не создает свой контекст выполнения (`this`)
- Не может быть конструктором
- Лучше всего подходит для функций-не-методов

```
let a = [1, 2, 3, 4, 5];

// если нужно вычислить одно выражение, то фигурные скобки и return можно опустить
// аналогично a.sort(function(a, b) { return b - a; });
a.sort((a, b) => b - a);

// если у функции только один параметр, то скобки вокруг списка аргументов можно опустить
// аналогично a.map(function(x) {return x + 1;});
a.map(x => x + 1);

// если функция не имеет параметров, круглые скобки обязательны
// аналогично a.filter(function() { return Math.round(Math.rand()); })
console.log(a.filter( () => Math.round(Math.random()) ));
```

Параметры

- Функция может содержать произвольное число параметров
- Может вызываться с произвольным числом аргументов
- Если число параметров больше числа переданных аргументов, недостающие параметры становятся `undefined`
- Если нужно передать много параметров, можно использовать конфигурационный объект
- Объекты передаются по ссылке

```
function f(options, data) {
    data; // undefined
    options.customField = 'test';
}

let options = { option1: "...", option2: 200, ... };
f(options);
options.customField; // 'test'
```

Параметры: `arguments` (how things were)

- `arguments` автоматически создается при входе в функцию
- Полезен для работы с переменным числом аргументов
- array-like object, содержит все значения аргументов
- Нельзя объявить переменную или изменить значение `arguments` (*strict*)
- Переопределяется в каждой вложенной функции

```
function f(a, b) {
    arguments.length; // 4
    Array.isArray(arguments); // false
    Array.from(arguments); // [1, 2, 3, 4]

    arguments[0] === a; // true
    arguments[1] === b; // true
}
f(1, 2, 3, 4);

function join(separator) {
    let args = Array.from(arguments).slice(1);
    return args.join(separator);
}
join(':', 'a', 'b', 'c'); // => 'a:b:c'
```

```
function outer() {
    [...arguments];     // [1, 2, 3]
    let x = 15;
    function inner() {
        [...arguments]; // ['a', 'b', 'c']
        x; // 15, из внешней области видимости
    }

    inner('a', 'b', 'c');
}
outer(1, 2, 3);
```

Параметры: rest-параметры (how things are)

- Для переменного числа аргументов удобнее использовать rest синтаксис
- Является массивом
- Содержит только "дополнительные" (неименованные) аргументы
- Может находиться только в конце списка параметров

```
function join(separator, ...values) {
    return values.join(separator); // не нужно вырезать первый параметр
}
join(':', 'a', 'b', 'c'); // => 'a:b:c'

function outer(...outerArgs) {
    outerArgs;     // [1, 2, 3]
    let x = 15;
    function inner(...innerArgs) {
        outerArgs; // [1, 2, 3], из внешней области видимости
        innerArgs; // ['a', 'b', 'c']
        x; // 15, из внешней области видимости
    }

    inner('a', 'b', 'c');
}
outer(1, 2, 3);
```

Параметры: значения по умолчанию (how things were)

- Раньше нельзя было указывать значения параметров по умолчанию
- Решение - дополнительный код в начале функции

```
function makeRequest(url, timeout, callback) {
    timeout = timeout || 2000;
    callback = callback || function() {};
    . . .
}
makeRequest("...", 0, function() {});  // 0 - валидное значение для timeout!

function makeRequest(url, timeout, callback) {
    timeout = timeout !== undefined ? timeout : 2000;
    callback = callback !== undefined ? callback || function() {};
    . . .
}
makeRequest("...", 0, function() {});  // все ok, используем timeout = 0
```

Параметры: значения по умолчанию (how things are)

- Можно использовать не только литералы, но и выражения
- Можно использовать значения предыдущих параметров в списке
- `arguments` отражает аргументы, указанные при вызове функции

```
function f(a, b = 2000, c = () => {}) {. . .}
f('foo');         // b, c по умолчанию
f('foo', 500);    // c по умолчанию

// Опциональный параметр - необязательно последни
function f2(a, b = 2000, c) {. . .}

f2("foo", undefined, 3); // b = 200, c = 3
f2("foo"); // b = 200, c = undefined

f2("foo", null, 3); // b = null, c = 3
```

```
let value = 5;
function getValue() {
    return value++;
}

function add(first, second = getValue()) {
    return first + second;
}
add(1, 1); // => 2, getValue не вызвана
add(1);    // => 6, getValue вызвана
add(1);    // => 7, getValue вызвана

function add(first, second = first) {
    return first + second;
}
add(1, 1); // => 2
add(1);    // => 2
```

Параметры: связь с деструктурированием

- Присвоение значений параметрам функции при ее вызове работает в точности как деструктурирование массивов
- Можно использовать деструктурирование объектов в параметрах
- Можно использовать rest-синтаксис в списке параметров
- Можно использовать spread-синтаксис в списке аргументов

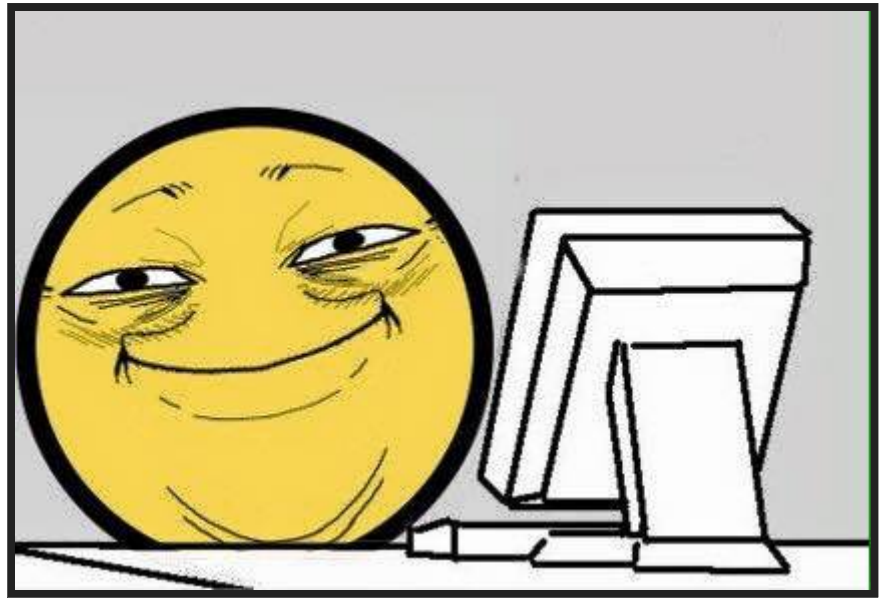
```
// объявление функции
f(param1, param2 = defaultValue2, ...param3) {...}
// вызов функции
f(arg1, arg2, arg3, ...arg4);

// Определение значений параметров по списку аргументов
let [param1, param2 = defaultValue2, ...param3] = [arg1, arg2, arg3, ...arg4];

function drawChart({size = 'big', cords = {x: 0, y: 0}, radius = 25} = {} ) {
  console.log(size, cords, radius); // => 'big' {x: 18, y: 30}, 30
  . . .
}

drawChart({cords: {x: 18, y: 30}, radius: 30});
```

`this`: контекст вызова функции

- `this` - ключевое слово
- определяется видом функции и способом ее вызова
- может содержать любое значение (как правило, ссылка на объект)
- можно считать дополнительным неявным аргументов функции
- нельзя присвоить значение напрямую
- вложенные функции не наследуют `this` от вызывающей функции, а определяют свое

`this`: контекст вызова функции

- Определить чему равно `this` очень легко!
- 3 способа создания функции * 3 способа вызова функции * 2 варианта работы интерпретатора
- 3 способа задать контекст вручную
- Пара специальных случаев - `eval()`, глобальная область видимости
- Всего - более 20 различных вариантов




Ölbaum
@oscherler
JavaScript makes me want to flip the table and say “F k this shit”, but I can never be sure what “this” refers to.
РЕТВИТОВ
805
ОТМЕТОК «НРАВИТСЯ»
687
15:03 - 30 окт. 2015 г.

`this`: тривиальные случаи

- В глобальной области видимости `this` - глобальный объект
- В функции, созданной через `new Function()` `this` - глобальный объект
- При вызове обычной функции `this` равно `undefined` (*strict*)

```
'use strict';

this;  // Window/Global

var a = 5, b = 6;  // var - чтобы a и b стали свойствами глобального объекта
let x = new Function('', 'return this.a + this.b;');
x();   // 11

function f() {
    return this;
}
f();   // undefined
```

`this`: вызов метода объекта

- При вызове метода объекта `this` указывает на этот объект
- Если метод находится в прототипе объекта, `this` все равно указывает на изначальный объект

```
let obj = {
   a: 2,
   b: 3,
   add() { return this.a + this.b; }
};
obj.add(); // 5
obj"add"; // 5

let mulF = function() { return this.a * this.b; }
obj.mul = mulF;
mulF();     // TypeError, this === undefined
obj.mul();  // 6 ok, this === obj
```

```
let obj2 = {
    a: 10,
    b: 100,
    div() { return this.b / this.a; }
};
Object.setPrototypeOf(obj, obj2);
obj.div(); // 1.5
// метод берется из прототипа obj2,
// this указывает на изначальный объект obj
```

`this`: вызов конструктора

- Каждую (почти) функцию в JS можно вызвать как конструктор: `new f()`
- `this` внутри вызова конструктора указывает на новый пустой объект
- Этот объект наследует от свойства `prototype` функции-конструктора

```
let Constr = function() {
   this.a = 1;
   this.greet = function() { return 'hi'; };
   return this;
}
Constr.prototype = {
    getA()  { return this.a; },
    greet() { return 'hello'; }
}
let obj = new Constr();
Object.getPrototypeOf(obj) === Constr.prototype;  // true
obj.getA();   // 1
obj.greet();  // 'hi'

Constr.prototype.getA();  // undefined
Constr.prototype.greet(); // 'hello'
```

`this`: вызов функции с заданным контекстом

- `call/apply` вызывает функцию с заданным контекстом и аргументами
- Первый аргумент `call/apply` - контекст при вызове функции (*strict*)
- Вызов `call` - содержит список аргументов функции
- Вызов `apply` - второй аргумент - массив аргументов для функции

```
let obj = {
   a: 2,
   b: 3,
   add: function(c, d) {
       return this.a + this.b + c + d;
   }
};
obj.add(5, 6); // 16
obj.add.call({a: 10, b: 20}, 5, 6); // 41
obj.add.apply({a: 10, b: 20}, [5, 6]); // 41
```

```
let aLike = {
    0: 'a', 1: 'b', 2: 'c',
    length: 3
};
aLike.join(':');  // TypeError
Array.prototype.join.call(aLike, ':'); // 'a:b:c'

// Единственный способ разложить массив
// в список аргументов до появления spread
Math.max.apply(undefined, [1, 17, 11]); // => 17
```

`this`: создание функции с заданным контекстом

- `Function.prototype.bind()` создает новую функцию
- Связывает (bind) новую функцию с указанным контекстом
- Связывает новую функцию со значениями аргументов (частичное применение/currying)

```
let sum = function(x, y) { return x + y; },
    sum5 = sum.bind(null, 5);
sum5(9); // 14

sum = function() { return this.x + this.y; };
boundSum = sum.bind({x: 1, y: 2});
boundSum();                    // 3
boundSum.call({x: 5, y: 10}); // все равно 3
```

```
let obj = {
   x: 2,
   y: 3,
   addFunction() {
       return this.x + this.y;
   }
};
let f = obj.addFunction;
f(); // TypeError, потеряли контекст

f = obj.addFunction.bind(obj);
f(); // 5, все ok
```

`this`: вложенные функции

- Обычные вложенные функции не могут обратиться к this снаружи
- Можно присвоить `this` другой переменной, которая будет в scope
- Можно использовать `bind`
- Можно использовать arrow-функции, которые разделяют значение `this` и `arguments` со внешней функцией

```
let obj = {
    sort: 'ASC', // ASC/DESC
    data: [5, 9, -4, 19, 0, 3],
    sortData() {
        this.data.sort(function(a, b) {
            return this.sort === 'ASC' ? a - b : b - a;
        })
    }
};
obj.sortData(); // TypeError;
```

this: вложенные функции

```
let obj = {
  ...
  sortData() {
    let self = this;
    this.data.sort(function(a, b) {
      return self.sort === 'ASC' ? a - b : b - a;
    })
  }
}
obj.sortData();
// ok, self в отличие от this есть в scope
```

```
let obj = {
  ...
  sortData() {
    this.data.sort(
      (a, b) =>
        this.sort === 'ASC' ? a - b : b - a
    );
  }
}
obj.sortData();
// ok, у arrow функции нет собственного контекста
```

```
let obj = {
  ...
  sortData() {
    this.data.sort(function(a, b) {
      return this.sort === 'ASC' ? a - b : b - a;
    }.bind(this));
  }
}
obj.sortData();
// ok, вложенная функция связана с obj
```

Итоги подведем: как создать функцию в зависимости от задачи

- Для обычных функций в общем потоке выполнения - FE/NFE/Arrow functions (и не используем в теле функции `this`)
- Для методов объектов - краткий синтаксис method definition
- Для вложенных (особенно коротких!) функций - Arrow function notation
- Для функций, код которых заранее неизвестен, - `new Function()`

Замыкания

- JS использует лексическую область видимости
- При вызове функции действует scope, который имелся на момент ее создания, а не вызова
- Замыкание - это функция, хранящая ссылку на область видимости, в которой она создана
- Все функции в JS являются замыканиями

```
let scope = "global";
function checkScope() {
    let scope = "local";

    function inner() {
        return scope;
    }
    return inner();
}
checkScope(); // => ? "local"
```

```
var scope = "global";
function checkScope() {
   let scope = "local";

   function inner() {
       return scope;
   }
   return inner;
}
checkScope()(); // => ? "local"
```

Замыкания: инкапсуляция данных

```
let scope = "global";
function checkScope() {
    var scope = "local";

    return {
        getScopeVar()  { return scope; },
        setScopeVar(s) { scope = s; }
    };
}
let obj = checkScope();
obj.getScopeVar();            // => ? "local"
obj.setScopeVar("new scope");
obj.getScopeVar();            // => ? "new scope"

var obj2 = checkScope();
obj2.getScopeVar();           // => ? "local"!
```