Интернет-технологии и распределённая обработка данных

Лекция 13

Функции JavaScript

1. Объявление функции
2. Аргументы и параметры функций
3. Вызов функции
4. Функции как объекты
5. Замыкания

Функции

Функция – блок кода, который определяется один раз и может выполняться (*вызываться*) многократно.

При определении функции можно задать *параметры*.

При вызове функции на месте параметров указываются *аргументы*.

Результатом вызова функции является *возвращаемое значение*.

Объявление (определение) функции

Способы объявить функцию:

– в декларативном стиле;

– в функциональном стиле;

– в стиле ООП.

Объявление функции в декларативном стиле

- Ключевое слово function
- Идентификатор (обязательно)
- Список имён формальных параметров (и значений по умолчанию) в круглых скобках разделенных запятыми
- тело функции в фигурных скобках вида { }

```
function идентификатор(параметры) {
инструкции
return выражение
}
```

```
function square(number) {
  return number * number;
}
alert(square(5));



function showMessage() {
alert( 'Привет всем присутствующим!' );
}
showMessage();
showMessage();
```

Функции и области видимости

Напоминание: в JavaScript именно функции определяют область видимости.

Локальные переменные видны только в функции, глобальные – везде.

Перефразируем: **функция имеет доступ к переменными своего окружения**. Причем доступ и на чтение, и на запись.

Область видимости переменной

Область видимости переменной (scope) – та часть скрипта, для которой эта переменная определена.

В JavaScript область видимости бывает только *глобальной* и *локальной*. Область видимости связана с функциями (а не с блоками кода, как в других языках!).

Глобальная область видимости

Переменная, объявленная вне функции на уровне всего скрипта, называется *глобальной*. Она имеет глобальную область видимости – определена для всего скрипта.

```
var x = 10;          // глобальная переменная x

function f() {
    alert(x);        // она видна везде в скрипте
}
```

При use strict – ошибка!

```
function showMessage() {
message = 'Привет'; // без var!
//без use strict переменная создается
// автоматически на уровне всего скрипта
}
showMessage();
alert( message ); // Привет
```

Локальная область видимости

Переменные, объявленные внутри функции, определены только в её теле. Они называются *локальными* и имеют локальную область видимости.

Параметры функций также считаются локальными переменными.

Локальная переменная имеет преимущество перед одноимённой глобальной переменной!

Локальные переменные

```
function showMessage() {
// локальная переменная, объявленные через var
// видна только внутри функции
var message = 'Привет!';
alert( message );
}

showMessage(); // 'Привет!'
alert( message ); // <-- будет ошибка, т.к.
// переменная видна только внутри
```

Блоки if/else, switch, for, while, do..while не влияют на область видимости переменных. При объявлении переменной в них – видна во всей функции

```
function count() {
//переменные i,j не будут уничтожены по окончании цикла
for (var i = 0; i < 3; i++) {
var j = i * 2;
}
alert( i ); // i=3, последнее значение i в цикле
alert( j ); // j=4, последнее значение j в цикле
}
```

```
if (true) {
var x = 5;
}
console.log(x); // 5 – область видимости не блок
```

поведение меняется, если используется оператор let

```
if (true) {
let y = 5;
}
console.log(y); // ReferenceError
```

Подъём локальной переменной

Внутри функции локальные переменные видимы даже **до строки с объявлением**! Выглядит так, будто все объявления переменных внутри функции транслятор «подымает» к началу функции (variable hoisting).

Правда при таком «подъёме» начальные значения не учитываются (до выполнения инструкции `var`).

```
//переменные, которые еще не были инициализированы, возвратят
//значение undefined
console.log(x === undefined); // true
var x = 3;

//будет интерпретироваться так же, как:
var x;
console.log(x === undefined); // true
x = 3;
```

```
var myvar = "my value";
 (function() {console.log(myvar); // undefined
var myvar = "local value";
})();

//будет интерпретироваться так же, как:
var myvar = "my value";
 (function() {
var myvar;
console.log(myvar); // undefined
myvar = "local value";})();
```

//let (const) **не будет подниматься** вверх блока

```
function do_something() {
console.log(foo); // ReferenceError
let foo = 2;
}
```

Поднятие функций

Только для декларативного объявления - "всплытие"(hoisting) в начало функции: **создаются интерпретатором до выполнения кода**, поэтому их можно вызвать *до* объявления

```
{
  print(square(5));


 // инициализация "всплывает" вместе с декларацией переменной square
 // Аналогичный код в функциональном стиле работать не будет
  function square(n){return n*n}
```

//функции, определённые через выражения, не поднимаются

```
/* декларативно*/
foo(); // "bar"
 function foo() {
console.log("bar");
}

/* функционально - через выражение */
baz(); // TypeError: baz is not a function
var baz = function() {
console.log("bar2");};
```

Неважно, где именно в функции и сколько раз объявляется переменная. Любое объявление срабатывает один раз и распространяется на всю функцию

```
function count() {
for(var i=0; i < 3; i++) {
var j = i * 2;
}
alert( i ); // i=3
alert( j ); // j=4
}
```

```
function count() {
var i, j; // передвинули
//объявления var в начало
for(i = 0; i < 3; i++) {
j = i * 2;
}
alert( i ); // i=3
alert( j ); // j=4
}
```

Следствие variable hoisting

При повторном объявлении и без инициализации переменная даже не потеряет своего значения:

```
var x = 10;
alert(x);          // выведет "10"
var x;             // повторное объявление
alert(x);          // всё равно выведет "10"
```

Внешние переменные

```
var userName = 'Робот';

function showMessage() {
//обращение к внешней переменной
var message = 'Привет, я ' + userName;
alert(message);
}

showMessage(); // Привет, я Робот
```

```
var userName = 'Робот';

function showMessage() {
// присвоение во внешнюю переменную
userName = 'Петя';
/*если бы внутри функции была бы объявлена своя локальная переменная var userName, то все обращения использовали бы её, и внешняя переменная осталась бы неизменной*/
var message = 'Привет, я ' + userName;
alert(message);
}

showMessage(); // Привет, я Петя
```

Параметры

Параметры копируются в локальные переменные функции.

```
function showMessage(from, text) {
// параметры from, text
from = "** " + from + " **";
// здесь может быть сложный код оформления
alert(from + ': ' + text);
}

showMessage('Маша', 'Привет!');
showMessage('Маша', 'Как дела?');
```

```
function showMessage(from, text) {
// меняем локальную переменную from
from = '**' + from + '**';
alert( from + ': ' + text );
}

var from = "Маша"; /*внешняя переменная from, значение которой при запуске функции копируется в параметр функции с тем же именем*/

showMessage(from, "Привет");
// старое значение from без
//изменений, в функции была изменена копия
alert( from );
```

Аргументы по умолчанию

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

При объявлении функции необязательные аргументы, как правило, располагают в конце списка

```
function showMessage(from, text) {
if (text === undefined) {//значение по умолчанию
text = 'текст не передан';}
alert( from + ": " + text );
}
showMessage("Маша", "Привет!"); // Маша: Привет!
showMessage("Маша"); // Маша: текст не передан
```

Способы указания значения «по умолчанию»:

1) проверить, равен ли аргумент `undefined`, и если да – то записать в него значение по умолчанию
2) Использовать оператор `||`:

```
function showMessage(from, text) {
text = text || 'текст не передан';
...
}
```

Здесь аргумент отсутствует, если передана пустая строка, `0`, или вообще любое значение, которое в логическом контексте является `false`

3) Использовать знак равенства =

```
function showMenu(title = "Без заголовка", width = 100, height =
200) {
alert(title + ' ' + width + ' ' + height);
}


showMenu("Меню"); // Меню 100 200
```

Параметр по умолчанию используется при отсутствующем аргументе или равном undefined. При передаче любого значения, кроме undefined, включая пустую строку, ноль или null, параметр считается переданным, и значение по умолчанию не используется.

```
 function showMenu(title = "Заголовок", width = 100, height = 200) {
alert('title=' + title + ' width=' + width + ' height=' + height);
}


// По умолчанию будут взяты 1 и 3 параметры
// title=Заголовок width=null height=200
showMenu(undefined, null);
```

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

```
function sayHi(who = getCurrentUser().toUpperCase()) {
 alert('Привет, ' + who); }
function getCurrentUser() { return ‘Маша'; }
sayHi(); // Привет, МАША
```

выражения getCurrentUser().toUpperCase() будет вычислено, и соответствующие функции вызваны, когда функция вызвана без параметра.

В частности, выражение по умолчанию не вычисляется при объявлении функции.

функция getCurrentUser() будет вызвана именно в последней строке, так как не передан параметр.

вычисление значения по умолчанию будет происходить В МОМЕНТ ВЫЗОВА функции

такие выражения могут адресоваться к предыдущим параметрам:

```
function multiply(a, b = 2*a) {
return a*b;
}

multiply(5); // 50
```

Возврат значения

```
function calcD(a, b, c) {
return b*b - 4*a*c;
}
var test = calcD(-4, 2, 1);
alert(test); // 20
```

```
function checkAge(age) {
if (age > 18) {
return true;
} else {
return confirm('Родители разрешили?');
}}
var age = prompt('Ваш возраст?');
if (checkAge(age)) {
alert( 'Доступ разрешен' );
} else {
alert( 'В доступе отказано' );}
```

`return` может также использоваться без значения, чтобы прекратить выполнение и выйти из функции

```
function showMovie(age) {
if (!checkAge(age)) {
return;
}

alert( "Фильм не для всех" ); // (*)
// ...
}
```

когда функция не вернула значение или `return` был без аргументов, считается что она вернула `undefined`

```
function doNothing() { /* пусто */ }
alert( doNothing() ); // undefined

function doNothing() {
return;
}

alert( doNothing() === undefined ); // true
```

Анонимные функции

Функциональное выражение, которое не записывается в переменную, называют анонимной функцией.

```
//выводит вопрос на подтверждение question и, в
//зависимости от согласия пользователя, вызывает
//функцию yes() или no():
function ask(question, yes, no) {
//question Строка-вопрос
//yes Функция
//no Функция

    if (confirm(question)) yes()
    else no();
  }
```

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();  }

function showOk() {
  alert( "Вы согласились." );}

function showCancel() {
  alert( "Вы отменили выполнение." );}

 // использование
ask("Вы согласны?", showOk, showCancel);
```

```
  function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();}
```

Здесь функции объявлены прямо внутри вызова ask(...),
даже без присвоения им имени:

```
  ask(
    "Вы согласны?",
    function() { alert("Вы согласились."); },
     function() { alert("Вы отменили выполнение.");
  }
);
```

Выбор имени функции

Функции, которые начинаются с "show" – что-то показывают:

```
showMessage(..)   // префикс show, "показать" сообщение
```

Функции, начинающиеся с "get" – получают, и т.п.:

```
getAge(..)          // get, "получает" возраст
calcD(..)           // calc, "вычисляет" дискриминант
createForm(..)      // create, "создает" форму
checkPermission(..) // check, "проверяет"
        //разрешение, возвращает true/false
```

- `lowerCamelCase`
- Имя функции должно понятно и чётко отражать, что она делает.
- для имён функций, как правило, используются глаголы
- Одна функция – одно действие
- **в функции не должно быть ничего, кроме самого действия и поддействий, неразрывно связанных с ним**
- Имена функций, которые используются *очень часто*, иногда делают сверхкороткими:

  во фреймворке jQuery есть функция $,

  во фреймворке Prototype – функция $$,

  в библиотеке LoDash – функция с названием из одного символа подчеркивания _.

отличие в поведении?

```
function checkAge(age) {
if (age > 18) {
return true;
} else {
// ... return confirm('Родители разрешили?'); } }
```

```
function checkAge(age) {
if (age > 18) {
return true; } // ...
return confirm('Родители разрешили?'); }
```

отличие в поведении?

```
function checkAge(age) {
 return (age > 18) ? true : confirm('Родители разрешили?'); }

function checkAge(age) {
return (age > 18) || confirm('Родители разрешили?'); }

function checkAge(age) {
if (age > 18) { return true;
} else {
return confirm('Родители разрешили?'); } }
```

```
function pow(x, n) {
    var result = x;
    for (var i = 1; i < n; i++) {
      result *= x; }
    return result; }
var x = prompt("x?", '');
var n = prompt("n?", '');
if (n <= 1) {
    alert('Степень ' + n + 'не поддерживается,
                        введите целую степень, большую 1' );
} else {
    alert( pow(x, n) ); }
```

Объявление функции в функциональном стиле

Выполняется внутри *выражения*.

```
var f = function(параметры) {
// тело функции
};
```

Такие функции, как правило, **анонимны**:

```
var square = function(number) {
  return number * number;
};
```

Но могут иметь определённое имя. (это имя удобно использовать для рекурсии, или в отладчике (debugger)):

```
var factorial = function fact(n) {
return n<2 ? 1 : n*fact(n-1)
};
alert(factorial(3));

//анонимно
var sayHi = function(person) {
  alert( "Привет, " + person );
};
sayHi('Маша');
```

Function Declaration (инструкция) – функция, объявленная в основном потоке кода.

Function Expression (выражение) – объявление функции в контексте какого-либо выражения, например присваивания.

```
// Function Declaration
function sum(a, b) {
return a + b;
}
// Function Expression
var sum = function(a, b) {
return a + b;
}
```

Нельзя вызвать до объявления

```
sayHi("Вася"); // ошибка!
var sayHi = function(name) {
alert( "Привет, " + name );
}

//error: sayHi is not a function
```

```
function test() {
  foo();  // TypeError "foo is not a function"
  bar();  // "this will run!"

  var foo = function () { alert("this won't run!"); }

  function bar() {
    alert("this will run!");
  }
}

test();
```

Function Declaration vs. **Function Expression**

- **Declaration** объявляет переменную с именем функции и присваивает ей объект функции
- Для **Declaration** действует «подъём» вверх
- **Expression** само по себе переменную не объявляет
- В **Expression** имя функции опционально и доступно только внутри выражения (выражения с именем функции называются *Named Function Expression*, NFE, а выражения без имени – *анонимными функциями*)

	Function Declaration	Function Expression
Время создания	До выполнения первой строчки кода.	Когда управление достигает строки с функцией.
Можно вызвать до объявления	Да (т.к. создаётся заранее)	Нет
Условное объявление в `if`	Не работает	Работает

Условное объявление функции

Разное объявление функции в зависимости от условия

//Function Declaration при use strict видны только внутри //блока, в котором объявлены – будет ошибка!

```
var age = +prompt("Сколько вам лет?", 20);
if (age >= 18) {
  function sayHi() {
    alert( 'Прошу вас!' );  }
} else {
  function sayHi() {
    alert( 'До 18 нельзя' );  }}
sayHi();
```

Разное объявление функции в зависимости от условия

```
//в зависимости от условия, создаётся именно та функция,
//которая нужна – ошибки нет!
var age = +prompt('Сколько вам лет?');
var sayHi;
 if (age >= 18) {
   sayHi = function() {
     alert( 'Прошу Вас!' );  }
} else {
   sayHi = function() {
     alert( 'До 18 нельзя' );  }}


sayHi();
```

//в зависимости от условия, создаётся именно та функция,
//которая нужна – ошибки нет!

```
var age = prompt('Сколько вам лет?');
var sayHi = (age >= 18) ?
  function() { alert('Прошу Вас!');  } :
  function() { alert('До 18 нельзя'); };

sayHi();
```

Функциональные выражения

В JavaScript функция является значением, таким же как строка или число.

Как и любое значение, объявленную функцию можно вывести, вот так:

```
function sayHi() {
alert( "Привет" );
}
alert( sayHi ); // выведет код функции
//после sayHi нет скобок
```

Функцию можно скопировать в другую переменную:

```
function sayHi() {
//создается функция и помещается в переменную sayHi
alert( "Привет" );
}
// копируем функцию в новую переменную func
var func = sayHi;
func(); // Привет    // можно вызвать и sayHi();
sayHi = null;
sayHi();
// ошибка, т.к. null – не функция
```

Свойство «name»

```
'use strict';
//  Function Declaration
function f() {} // f.name == "f"
//Named Function Expression
let g = function g1() {}; // g.name == "g1"

alert(f.name + ' ' + g.name) // f g1
```

при создании анонимной функции с одновременной записью в переменную или свойство – её имя равно названию переменной (или свойства).

```
'use strict';
// свойство g.name = "g"
let g = function() {};

let user = {
  // свойство user.sayHi.name == "sayHi"
  sayHi: function() {}
};
alert(user.sayHi.name + '' + g.name);
```

Стрелочные функции

Современный стандарт языка поддерживает анонимные **стрелочные функции(fat arrow function)**.

более короткий синтаксис и лексика this

Тело стрелочной функции может иметь краткую или блочную форму

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

```
var func = x => x * x;                    // краткий синтаксис,
                                          // неявно возвращает результат
'use strict';
let inc = x => x+1; alert( inc(1) ); // 2
let sum = (a,b) => a + b; alert( sum(1, 2) ); // 3

var func = (x, y) => { return x + y; };    // блочный синтаксис,
                                           // явно возвращает результат
```

Если нужно задать функцию без аргументов, то также используются скобки, в этом случае – пустые:

```
// Пустая стрелочная функция возвращает undefined
let empty = () => {};
(() => "foobar")() // вернёт "foobar"


'use strict';
// вызов getTime() будет возвращать текущее время
let getTime = () => new Date().getHours() + ':' +
new Date().getMinutes();
alert( getTime() ); // текущее время
```

```
var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10
```

Когда тело функции достаточно большое, то можно его обернуть в фигурные скобки {…}. При этом результат уже не возвращается автоматически:

```
var complex = (a, b) => { if (a > b) {
return a;
} else {
return b; }
 }
```

```
var elements = [
"Hydrogen",
"Helium",
"Lithium",
"Beryllium”
 ];

//сравните
var elements2 = elements.map(function(s){ return s.length });
var elements3 = elements.map( s => s.length );
```

До появления стрелочных функций, каждая новая функция имела своё значение this (новый объект в случае конструктора, undefined в строгом режиме вызова функции, контекст объекта при вызове функции как "метода объекта" и т.д.). Стрелочные функции захватывают значение this

окружающего контекста

Поскольку значение this определяется лексикой, правила строгого режима относительно this игнорируются

```
var f = () => {'use strict'; return this};

f() === window; // или глобальный объект
```

Оставшиеся правила строго режима применяются как обычно.

Помимо упрощённого синтаксиса, такие функции всегда неявно привязываются в МОМЕНТ ОБЪЯВЛЕНИЯ к текущему лексическому *контексту* выполнения:

```
function Person(){
  this.age = 0;

  setInterval(() => {
        this.age++; // в данном случае this будет
//ссылаться на создаваемый объект obj, а не на window
  }, 1000);
}

var obj = new Person();
```

// Parentheses are optional when there's only one argument:

(singleParam) **=> {** *statements* **}**

singleParam **=>** { *statements* }

// A function with no arguments requires parentheses:

() => { *statements* **}**

// Advanced:

// Parenthesize the body to return an object literal expression:

params => ({*foo: bar*})

// Rest parameters are supported

(*param1*, *param2*, **...rest**) => { *statements* }

Объявление функции в стиле ООП

функция по сути является объектом, можно использовать оператор new и Function конструктор, чтобы создавать функции динамически во время выполнения (можно конструировать функцию, код которой неизвестен на момент написания программы).

Редко из соображений производительности и безопасности

```
new Function(params, code)
```

params – Параметры функции через запятую в виде строки.

code – Код функции в виде строки.

```
var sum = new Function('a, b', ' return a+b; ');
var result = sum(1, 2);
alert( result ); // 3
```

```
var sum = new Function("x", "y", "return x + y;");
var mult = new Function("x,y", "return x * y;");

alert(sum(3, 15));
alert(mult(4, 3));
```

Конструктор Function редко используется (медленный способ; ошибка в строке тела функции выдаёт SyntaxError).

Методы

Функции как *методы* объектов, реализующих ООП.

Метод это функция, заданная как значение свойства объекта.

```
class Greeting{
constructor(prefix){
    this.prefix = prefix; }
// это метод:
hello(name){
   return `${this.prefix}, ${name}`; }
}
var obj = new Greeting("Привет");
// вызов метода (obj передаётся в качестве контекста `this`)
obj.hello('Вася');
```

*Функции в JavaScript

В JavaScript любая функция:

получает при вызове дополнительный аргумент – *контекст вызова* (внутри функции доступен через `this`);

всегда возвращает некое значение;

может выступать в качестве *подпрограммы*, *метода* объекта, *конструктора* объекта;

является объектом (со всеми вытекающими последствиями).

«Самоопределяемые функции»

```
var f = function() {
    alert("A");
    f = function () {
        alert("B");
    }
}

f(); // "A"
f(); // "B"
f(); // "B"
```

Вложенные функции

Допускается вложение определений одних функций в другие функции:

```
function hypotenuse(a, b) {
    function square(x) { return x * x; }

    return Math.sqrt(square(a) + square(b));
}
```

```
function sayHiBye(firstName, lastName) {

alert( "Привет, " + getFullName() );
alert( "Пока, " + getFullName() );

function getFullName() {
return firstName + " " + lastName;
}
}

sayHiBye("Маша", "Селезнева");
// Привет, Маша Селезнева ; Пока, Маша Селезнева
```

//Здесь, для удобства, создана вспомогательная
//функция `getFullName()`.

если переменная не найдена во внешнем объекте переменных, то она ищется в ещё более внешнем – внешней функции:

```
var phrase = 'Привет';

function say() {

function go() {
alert( phrase ); // найдёт переменную снаружи
}

go();
}

say();
```

Вложенная функция имеет доступ к локальным переменным обрамляющей функции (уже говорили об этом):

```
var scope = "global";
function outer() {
    var scope = "local";
    function inner() {
        scope = "inner";
    }
 inner(); // этот вызов меняет локальную
      //переменную scope
    alert(scope);
}
outer(); // выводит "inner"
```

Функции в блоке

Объявление функции Function Declaration, сделанное в блоке, видно только в этом блоке.

```
'use strict';
if (true) {
    sayHi(); // работает
    function sayHi() {
     alert("Привет!");  }}
sayHi(); // не работает
```

такое объявление – ведёт себя в точности как если бы `let sayHi = function() {...}` было сделано в начале блока.

Перегрузки функций в JavaScript нет!

Если в JavaScript объявлять функции с одним именем, последнее объявление перекрывает все предыдущие:

```
function f() {
    alert("first");
}
function f() {
    alert("second");
}
function f(x) {
    alert("third");
}
f(); // выводит "third"
```

Вызов функции

В JavaScript функции могут вызываться четырьмя способами:

- как *функции*
- как *методы*
- как *конструкторы*
- *косвенно*, с помощью методов `call()` и `apply()`

Вызов функции как функции

Выполняется в виде *выражения вызова*.

```
var h = hypotenuse(3, 4);
```

Контекст вызова в нестрогом режиме и в ECMAScript 3 = **глобальный объект**, а в строгом режиме = **`undefined`**.

Вызов функции как метода

Метод – функция, которая хранится в свойстве объекта.

Вызов метода = выражение обращения к свойству + выражение вызова.

Контекстом вызова является объект, у которого выполняется обращение к свойству (вызывается метод).

Для доступа к текущему объекту из метода используется ключевое слово this

```
var matrix = {
    size: 0,
    setSize: function(s) {
        this.size = s;
    },
    printSize: function () {
        alert(this.size);
    }
}

matrix.setSize(20);
matrix.printSize(); //20
```

Нюанс контекста вызова (this)

Контекст вызова определяется именно **способом вызова** функции (а не местом вызова, как можно подумать)!

Это важно, если функция, которая вызывается как метод, вызывает свою вложенную функцию как функцию.

```
var obj = {
    method: function () {
        function checkThis()
        {
            alert(this == obj);
        }
        checkThis();
    }
}

// выводит false
obj.method();
```

```
var obj = {
    method: function () {

        function checkThis()
        {
         alert(this == obj);
        }
        checkThis();
    }
}

// выводит false
obj.method();
```

```
var obj = {
    method: function () {
        var self = this;
        function checkThis()
        {
           alert(self == obj);
        }
        checkThis();
    }
}

// выводит true
obj.method();
```

Вызов функции как конструктора

Это вызов функции (как функции или как метода), который предварён ключевым словом `new`.

1. Создаётся новый объект и назначается в качестве `this`.
2. Отрабатывает функция.
3. Возвращаемым значением функции **всегда** будет **объект** – либо объект после `return`, либо созданный на шаге 1 (если `return` отсутствует или возвращает не объект).

Имя функции-конструктора пишется с большой буквы.

Если функция вызывается как конструктор и не имеет аргументов, то можно не указывать круглые скобки после её имени:

```
//но рекомендуется все же указывать
// эквивалентные строки
var o = new Object();
var o = new Object;
```

```
function Animal(name) {
  this.name = name;
   this.canWalk = true; }
   var animal = new Animal("ёжик");

//интерпретатор
function Animal(name) {
// this = {};  – создается новый пустой объект
// в this пишем свойства, методы
this.name = name;
this.canWalk = true;
// return this;
}
```

new function() { … }

```
//функцию-конструктор объявляют и тут же используют
//создается единственный объект класса
var animal = new function() {
    this.name = "Васька";
    this.canWalk = true;
};
```

Создание методов в конструкторе

```
function User(name) {
this.name = name;
this.sayHi = function() {
alert( "Моё имя: " + this.name );
};
}
var ivan = new User("Иван");
ivan.sayHi(); // Моё имя: Иван

/* ivan = { name: "Иван", sayHi: функция } */
```

Косвенный вызов

Любая функция – это особый объект.

У этого объекта есть методы `call()` и `apply()`.

Оба принимают в качестве первого аргумента контекст вызова функции.

Далее идут аргументы функции: у `call()` через запятую, у `apply()` – в виде массива (или в виде *объекта, подобного массиву*).

```
func.call(context, arg1, arg2, ...)
//первый аргумент call становится this функции func
//остальные передаются «как есть».
```

```
var user = {
firstName: "Василий",
surname: "Петров",
patronym: "Иванович"
};
function showFullName(firstPart, lastPart) {
 alert( this[firstPart] + " " + this[lastPart] ); }
// f.call(контекст, аргумент1, аргумент2, ...)
showFullName.call(user, 'firstName', 'surname') // "Василий Петров"
showFullName.call(user, 'firstName', 'patronym') // "Василий Иванович
«
showFullName.call(user) // "Василий Петров"
```

Вызов функции при помощи func.apply работает аналогично func.call, но принимает массив аргументов вместо списка

Используется, если неизвестно с каким кол-вом аргументов вызвать функцию

```
func.call(context, arg1, arg2); // идентичен вызову
func.apply(context, [arg1, arg2]);
```

```
showFullName.call(user, 'firstName', 'surname');
showFullName.apply(user, ['firstName', 'surname']);
```

```
var arr = [];
arr.push(1);
arr.push(5);
arr.push(2);

// получить максимум из элементов arr
alert( Math.max.apply(null, arr) ); // 5
//в качестве контекста все равно, что передавать
//метод Math.max не использует this
```

```
function delta(dx) {
    return this.x + dx;
}

var o = { x: 10 };
var dc = delta.call(o, 1); //11
var da = delta.apply(o, [2]);//12
alert(dc);
alert(da);
```

Первый аргумент `call()` и `apply()`:

В строгом режиме всегда становится контекстом вызова «как есть».

В нестрогом режиме: если первый аргумент `null` или `undefined` – контекстом будет глобальный объект, если это примитивное значение – контекстом будет объект-обёртка.

Аргументы и параметры функций

Типы аргументов функции могут быть как примитивами (строки, числа, логические(boolean)), так и объектами (включая Array или функции):

- Значения-примитивы передаются в функции **по значению**: значение КОПИРУЕТСЯ, так что если функция изменит значение параметра, это изменение не будет иметь внешнего эффекта.
- объекты передаются в функцию **по ссылке:** переприсваивание самой ссылки также не имеет внешнего эффекта, НО если функция изменяет свойства объекта, то эти изменения будут видимы вне функции (побочный

```
function f(x)
{
    x = -1;
}

var y = 10;
f(y);
alert(y); // "10"
```

```
function f(o)
{
    o.x = -1;
}

var y = { x: 10 };
f(y);
alert(y.x); // "-1"
```

Следствие слабой типизации: часто необходим ручной контроль и (или) приведение типов аргументов.

```
function foo(str) {
    if (typeof str == "string") {
        return str+"foo " ;
    }
    else {
        throw new TypeError("We need string!");
    }
}
```

Если при вызове функции аргументов меньше, чем параметров, то незаданные параметры будут равны undefined:

```
function f(x, y) {
    alert(x);
    alert(y);
}

f(1); // печатает "1" и "undefined"
```

В JavaScript распространён следующий приём: аргументы функции передаются через свойства объекта.

Отчасти это решает проблему с именованными и необязательными параметрами.

```
function printRange(p) {
    p = p || {};
    var lo = p.lo || 1;
    var hi = p.hi || 1;
    var delta = p.delta || 1;
    for (var i = lo; i <= hi; i += delta)
        alert(i);
}

printRange({ lo: 1, hi: 10, delta: 2 });
```

Объект Arguments

Объект, созданный конструктором `Arguments`, используется для хранения **всех** аргументов, переданных функции в порядке передачи при вызове.

В теле любой функции этот объект доступен через идентификатор `arguments`.

```
function max()
{
    var m = Number.NEGATIVE_INFINITY;
    for(var i = 0; i < arguments.length; i++)
        if (arguments[i] > m) m = arguments[i];
    return m;
}
var largest = max(1, 10, 100, 2, 3, 1000, 4, 5,
10000, 6); // => 10000
```

```
function myConcat(separator) {
    var result = ""; // initialize list
// iterate through arguments
    for (var i = 1; i < arguments.length; i++) {
        result += arguments[i] + separator;
    }
    return result;}


 // returns "red, orange, blue, «
 myConcat(", ", "red", "orange", "blue");
  // returns "elephant; giraffe; lion; cheetah; "
myConcat(";","elephant","giraffe","lion","cheetah");
```

Внимание: в **нестрогом режиме** изменение `arguments` ведёт к изменению соответствующих именованных аргументов!

А в "use strict"; нет

```
function f(x)
{
    alert(x);
    arguments[0] = 0; // изменится x
    alert(x);
}

f(100); // выведет "100" и "0"
```

Аргументы и оператор развёртки

Современный стандарт языка позволяет отказаться от arguments, заменив его оператором развёртки (spread) (...) :

```
const myConcat = (sep, ...strings) => strings.join(sep) // !!!
```

Объект Arguments: callee и caller

В учебниках и в коде иногда можно встретить упоминание двух свойств объекта Arguments:

1. `callee` – ссылается на выполняемую функцию
2. `caller` – ссылается на вызвавшую функцию

Работа с этими свойствами не приветствуется!

1. `callee` не работает в строгом режиме. Вместо него можно и нужно использовать имя функции.

2. `caller` не поддерживается современными движками. Обычно пишут `arguments.callee.caller` (но помним про пункт 1).

Функции как объекты

Ранее упоминалось, что функции в JavaScript являются полноценными объектами.

Это так!

Заметим, что оператор `typeof` возвращает для функций не `"object"`, а `"function"`.

У объекта-функции есть свойства и методы:

Свойство или метод	Что делает
`length`	Число объявленных параметров функции. Доступно только для чтения.
`prototype`	Прототип функции. Используется при вызове функции как конструктора.
`call()` и `apply()`	Методы для косвенного вызова функции.
`bind()`	Создаёт новую функцию, которая при вызове устанавливает в качестве this предоставленное значение. (ECMAScript 5).
`toString()`	Метод возвращает исходный код функции в виде строки.

bind() – закрепление контекста вызова

Вызов `bind()` возвращает новую «привязанную функцию», у которой контекст вызова `this` жёстко установлен в указанное значение. **ПФ** - это "необычный функциональный объект" (термин из **ECMAScript 6**), который является оберткой над исходным функциональным объектом. Вызов **ПФ** приводит к исполнению кода обернутой функции.

```
function f(y) { return this.x + y; }
var obj = { x : 10 };
var g = f.bind(obj);
// вызов g(x) вызовет obj.f(x)
var result = g(2);   // result = 12
```

bind() – закрепление аргументов

При вызове `bind()` можно закрепить не только `this`, но и значения некоторых аргументов функции:

```
function f(y) { return this.x + y; }
var obj = { x : 10 };
var g = f.bind(obj, 2);
// теперь вызов g() всегда означает obj.f(2)
var result = g();   // result = 12
```

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

// контекст не используется, поэтому null
// фиксируем: x = 1, y = 2
// (это называется каррирование или карринг - создание новой функции путём фиксирования аргументов существующей)

var sumOneTwo = sum.bind(null, 1, 2);
var result = sumOneTwo(10); // result = 13
```

Так как функции – это объекты, их можно хранить и обрабатывать как объекты:

```
function f(x) {
    return "Str";
}

var o = f;
o(1);

var arr = [f];
arr[0](1);
```

В JavaScript к объекту можно в любой момент присоединить и инициализировать новое свойство. Функции не исключение!

```
// функция f из предыдущего слайда
f.newProp = "XYZ";
alert(f.newProp);
```

Можно передавать одни функции в качестве аргументов другим функциям:

```
function f(x, y, action) {
    alert(action(x, y));
}

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

f(10, 15, sum);
f(10, 15, function (x, y) { return x * y; });
```

Вполне возможно, что объект-функция возвращается другой функцией:

```
function func() {
    function res(x) {
        return x;
    }
    return res;
}

var f = func();
alert(f(10));
```

Замыкание

Если функция `F()` возвращает свою вложенную функцию `g()`, то все переменные из **scope** `F()` (и их значения) будут доступны в `g()`.

Это явление называется *замыканием* (closure).

```
function func(param) {
    var clos = param;
    function res(x) {
        return x + clos;
    }
    return res;
}

var f1 = func(100);
alert(f1(10));   // "110"

var f2 = func(200);
alert(f2(10));   // "210"
```

```
function uniqueID() {
    var counter = 0;
    return function () {
        return counter++;
    }
}

var u1 = uniqueID();
alert(u1());  // 0
alert(u1());  // 1

var u2 = uniqueID();
alert(u2());  // 0
```

```
function counter() {
    var n = 0;
    return {
        count: function () { return n++; },
        reset: function () { n = 0; }
    };
}

var c = counter(), d = counter(); //два счётчика
c.count() // => 0
c.count() // => 1
d.count() // => 0: действуют независимо
c.reset()
c.count() // => 0
```

Немедленно вызываемые функции

В современном JavaScript распространён приём: объявляется выражение-функция (без имени) и сразу же происходит вызов этой функции:

```
(function() {
    // тело функции
}());
```

```
(function() {
    // тело функции
})();
```

*) Внешние скобки нужны, чтобы транслятор понимал это как выражение-функцию (если он может подумать, что это инструкция).

Перепишем один из примеров для замыканий с использование данного приёма:

```
var uniqueID = (function() {
    var counter = 0;
    return function () {
        return counter++;
    }
}());
alert(uniqueID());  // 0
alert(uniqueID());  // 1
```

НВФ хороши тем, что дают изолированную область (по переменным), выполняемую один раз.

Это используют, например, в различных сценариях инициализации.

Или делают из НВФ подобие замкнутого модуля.