

Яндекс


Яндекс

ОСНОВЫ JavaScript

Роман Парадеев


Errata





Используется ли GPU
при выполнении кода
WebAssembly?





Слабое или строгое
сравнение
используется в *switch*?

Switch: строгое сравнение

```
let num = 42;

switch(num) {
  case '42': console.log('NO'); break;
  case 42:   console.log('YAY'); break;
}
```

Switch в PHP: слабое сравнение

```
$num = 42;  
  
switch($num) {  
    case '42':    console.log('YAY'); break;  
    case 42:      console.log('NO');  break;  
}
```


О чём эта лекция?

- › **Основы синтаксиса:** операторы, выражения, циклы
- › **Система типов:** число, строка, объект, массив, функция
- › **Основные концепции:** примитивы/объекты, выражения/утверждения, литералы, истинность/ложность, область видимости
- › **Демо**

Спецификация ECMAScript



JavaScript выполняется на самых разных платформах и устройствах: на сервере, в браузере, в мобильном приложении, в микроконтроллере, дроне и электрочайнике.

Уровень поддержки языка в этих платформах неодинаков. Мы будем придерживаться современной версии стандарта языка – **ES2017**, но время от времени будем рассматривать подходы, характерные для более старых версий (**ES3** и **ES5**).

Такие слайды будут помечены фотокарточкой Дугласа Крокфорда. Видео: [Crockford on JavaScript](#).

Strict Mode

`'use strict'` – директива, приводящая код в соответствие стандарту ES5. Указывается в начале файла или функции.

```
function sum(a, b) {  
    'use strict';  
  
    return a + b;  
}
```

Однострочный комментарий

// Однострочный комментарий

// Ещё один однострочный комментарий

Многострочный комментарий

```
/* Многострочный комментарий  
   (не рекомендуется) */
```

```
/* Этот комментарий сломается  
   из-за регулярного выражения  
   /[a-z]*/.test('Hello'); */
```

JSDoc

```
/**  
 * Поиск записи в книге по телефону  
 * @param {String} query  
 * @returns {Number} result  
 */  
function findByPhone(query) { ... }
```

Переменные

Перед использованием переменные нужно задекларировать.
Декларацию с присваиванием можно совместить.

```
let name;           // декларация  
name = 'Roman';    // присваивание
```

```
let age = 28;       // декларация с присваиванием
```

Константы

Переменную можно задекларировать как константу – тогда её значение нельзя будет изменять.

```
const name = 'Roman';
```

```
name = 'Pyotr'; // TypeError
```


Константы для ссылочных типов

Это не совсем верно для ссылочных типов. `const` запрещает замену самого объекта (или массива), но не изменение его содержимого.

```
const friends = [];
```

```
friends.push( 'Pyotr' ); // no error
```

Составные операторы присваивания

В JavaScript существуют несколько составных операторов присваивания: `+=`, `-=`, `/=`, `*=`, `%=`. Их удобно использовать, когда нужно модифицировать уже имеющееся значение.

```
let foo = 1;
```

```
foo += 1; // foo = foo + 1;
```

Идентификаторы

В качестве первого символа имени переменной можно использовать буквы, символ доллара (\$) и подчёркивание (_). Последующие символы могут содержать всё вышеперечисленное, а также цифры.

```
let appleCount; // camelCase, существительное
```

```
let __dirname; // системная переменная
```

```
let привет; // не рекомендуется
```

Зарезервированные слова

arguments	break	case	catch
class	const	continue	debugger
default	delete	do	else
enum	export	extends	false
finally	for	function	if
implements	import	in	instanceof
interface	let	new	null
package	private	protected	public
return	static	super	switch
this	throw	true	try
typeof	var	void	while

Зарезервированные слова

arguments

class

default

enum

finally

implements

interface

package

return

this

typeof

break

const

delete

export

for

import

let

private

static

throw

var

case

continue

do

extends

function

in

new

protected

super

true

void

catch

debugger

else

false

if

instanceof

null

public

switch

try

while

Литералы

Литералы – особая нотация, порождающая значение определённого типа. Например, строковый литерал используется для создания строки.

```
let myNumber = 100;    // number literal
let myString = 'Hello'; // string literal
let myBoolean = true;  // boolean literal
let myObject = {};     // object literal
let myArray = [];      // array literal
```

Строки

```
let foo = 'Hello World';
```

```
let bar = "Hello World";
```

```
let who = 'world';
```

```
let baz = `Hello ${who}`; // template literal
```

Числа

```
let integer = 10;  
let decimal = 10.1;
```

```
console.log(0.1 + 0.2); // 0.30000000000000004
```

```
console.log(2 - 'meh'); // NaN  
isNaN(2 - 'meh');      // true
```


Логический тип

```
let yeah = true;  
let nope = false;
```

```
console.log(!yeah);           // false  
console.log(yeah && nope);    // false  
console.log(yeah || nope);    // true
```

Объект

```
let me = {  
  name: 'Roman',  
  age: 28,  
};
```

```
me.age += 1;  
console.log(me.age); // 29
```

Объект

```
let me = {};
```

```
me.name = 'Roman';
```

```
me.age = 28;
```

```
console.log(me); // { name: 'Roman',  
                  //   age: 28 }
```

Массив

```
let fruits = ['apple', 'banana', 'citrus'];  
  
console.log(fruits[2]); // citrus  
  
fruits[2] = 'chicken';  
console.log(fruits[2]); // chicken
```

Массив

```
let fruits = ['apple'];  
  
fruits.push('banana');  
console.log(fruits); // ['apple', 'banana']  
  
fruits.pop();  
console.log(fruits); // ['apple']
```

Функция

```
let sayHi = function (who) {  
  console.log(`Hi, I am ${this.name}`);  
}  
  
console.log(sayHi); // [Function: sayHi]  
sayHi('Roman');    // "Hi, I am Roman"
```

Метод

```
let me = {  
  name: 'Roman',  
  sayHi: function () {  
    console.log(`Hi, I am ${this.name}`);  
  },  
};  
me.sayHi(); // "Hi, I am Roman"
```

undefined и null

Оба типа означают отсутствие информации, но с небольшими отличиями.

- › **undefined** – отсутствие значения. Например, неинициализированная переменная, непереданный параметр функции, несуществующее свойство.
- › **null** – отсутствие объекта

Когда использовать undefined?

```
let foo;  
console.log(foo); // undefined
```

```
foo = {};  
console.log(foo.bar); // undefined
```

```
console.log(); // undefined
```

Когда использовать null?

```
function findUserId(id) {  
    // ищем пользователя в базе данных  
}
```

```
findUserId(1); // { login: grumpy }  
findUserId(-1); // null
```

Свойства и методы

У всех значений, кроме undefined и null, есть свойства и методы.

```
let greeting = 'Hello world';
```

```
greeting.toUpperCase();
```

```
greeting.length;
```

Прототипы

Значение наследует их у своего прототипа. Так, методы строки наследуются от `String.prototype`. Подробнее про прототипы – в следующих лекциях.

```
let greeting = 'Hello world';  
  
greeting.toUpperCase ===  
  String.prototype.toUpperCase // true
```

Выражения и утверждения

```
let x;
```

```
x = 3 + 5;
```

```
if (x === 0) {  
    x = 123;  
}
```

Statements

Утверждение (**statement**) описывает некоторую выполняемую операцию. Программа – последовательность утверждений.

```
let x;  
x = 3 + 5;  
if (x === 0) {  
    x = 123;  
}
```

Statements

Утверждение (statement) описывает некоторую выполняемую операцию. Программа – последовательность утверждений.

```
let x; // declaration statement
x = 3 + 5;
if (x === 0) {
    x = 123;
}
```

Statements

Утверждение (statement) описывает некоторую выполняемую операцию. Программа – последовательность утверждений.

```
let x;  
x = 3 + 5; // assignment statement  
if (x === 0) {  
    x = 123;  
}
```


Statements

Утверждение (statement) описывает некоторую выполняемую операцию. Программа – последовательность утверждений.

```
let x;  
x = 3 + 5;  
if (x === 0) { // conditional statement  
    x = 123;  
}
```

Expressions

Выражения (**expression**) производят значения.

```
let x;  
x = 3 + 5;  
if (x === 0) {  
    x = 123;  
}
```

Expressions

Выражения (expression) производят значения.

```
let x;  
x = 3 + 5; // 8  
if (x === 0) {  
    x = 123;  
}
```

Expressions

Выражения (expression) производят значения.

```
let x;  
x = 3 + 5;  
if (x === 0) { // false  
    x = 123;  
}
```

Expressions

Выражения (expression) производят значения.

```
let x;  
x = 3 + 5;  
if (x === 0) {  
    x = 123; // 123  
}
```

Statement или expression?

```
let x;  
if (y >= 0) {  
    x = y;  
} else {  
    x = -y;  
}
```

```
let x = y >= 0 ? y : -y;
```

Statement или expression?

```
let x;  
if (y >= 0) {  
    x = y;  
} else {  
    x = -y;  
}  
// statement
```

```
let x = y >= 0 ? y : -y;
```

Statement или expression?

```
let x;  
if (y >= 0) {  
    x = y;  
} else {  
    x = -y;  
}
```

```
let x = y >= 0 ? y : -y;  
// expression
```


Statement или expression?

```
let x;  
if (y >= 0) {  
    x = y;  
} else {  
    x = -y;  
}
```

```
let x = y >= 0 ? y : -y;  
// expression  
  
foo(y >= 0 ? y : -y);  
// выражение можно,  
// например, передать  
// аргументом функции
```

Точка с запятой

После утверждения ставится точка с запятой. Если оно не заканчивается **блоком**.

```
let x;  
x = 3 + 5;  
if (x === 0) {  
    x = 123;  
}
```

Точка с запятой

После утверждения ставится точка с запятой. Если оно не заканчивается **блоком**.

```
let x;  
x = 3 + 5;  
if (x === 0) { // блок – код, обрамлённый  
    x = 123;    // фигурными скобками  
}
```

Автоматическая расстановка ;

Если не поставить точку с запятой, интерпретатор может поставить её сам. Но иногда он ошибается.

```
foo()  
['one', null].filter(Boolean)
```

Автоматическая расстановка ;

Если не поставить точку с запятой, интерпретатор может поставить её сам. Но иногда он ошибается.

```
foo() // здесь должна быть точка с запятой!  
['one', null].filter(Boolean)
```

Автоматическая расстановка ;

Если не поставить точку с запятой, интерпретатор может поставить её сам. Но иногда он ошибается.

```
foo() // здесь должна быть точка с запятой!  
['one', null].filter(Boolean)
```

```
// но интерпретатор считает иначе  
foo()['one', null].filter(Boolean)
```

Автоматическая расстановка ;

Если не поставить точку с запятой, интерпретатор может поставить её сам. Но иногда он ошибается.

```
foo();  
['one', null].filter(Boolean);
```

Автоматическая расстановка ;

Если не поставить точку с запятой, интерпретатор может поставить её сам. Но иногда он ошибается.

```
foo();
```

```
['one', null].filter(Boolean);
```

```
// теперь интерпретатор нас понял
```

```
foo(); ['one', null].filter(Boolean);
```


Выражения и утверждения

- › Утверждение (**statement**) описывает некоторую выполняемую операцию.
- › Выражения (**expression**) производят значения.
- › После утверждения ставится точка с запятой. Если оно не заканчивается **блоком**.
- › Не стоит полагаться на автоматическую расстановку точек с запятой. Результаты могут не совпадать с ожиданиями.

Примитивы и сложные объекты

Все значения в JavaScript относятся к двум видам: примитивы и объекты

- › **примитивы**: boolean, number, string, null, undefined
- › **объекты** – всё остальное: object, array, regular expressions, ...

Примитивы неизменяемы (иммутабельны) и сравниваются по значению. Объекты можно изменять и равны они только сами себе.

Примитивы

- › Логический тип: true, false
- › Числа: 1736, 1.351
- › Строки: 'abc', "abc"
- › undefined, null

Сравнение примитивов

Примитивы сравниваются по значению. Например, одинаковые строки равны.

```
console.log(3 === 3);           // true
```

```
console.log('Hi' === "Hi");    // true
```

```
console.log(false === !true);  // true
```

Изменение примитивов

Примитивы иммутабельны – их нельзя изменить, можно только создать новое значение.

```
let greeting = 'Hello';  
greeting[0] = 'h'; // пробуем изменить  
                  // первый символ  
  
console.log(greeting); // Hello
```

Передача в функцию

Примитивы передаются в функцию по значению. Это значит, что внутри функции мы работаем с копией, и не можем изменить исходное значение.

```
function inc(x) { x += 1; }
```

```
let x = 1;
```

```
inc(x);
```

```
console.log(x); // 1
```

Объекты

- › Простые объекты: { name: 'Roman', age: 28 }
- › Массивы: ['foo', 'bar', 'baz']
- › Регулярные выражения: / [a-z] * /
- › Функции

Сравнение объектов

Объекты равны только самим себе. Одинаковые объекты равны не будут.

```
console.log({} === {}); // false
```

```
let obj = {}:  
console.log(obj === obj); // true
```


Изменение объектов

В отличие от примитивов, объекты можно изменять.

```
let me = { age: 28 };  
me.age = -1;
```

```
console.log(me.age); // -1
```

Передача в функцию

Объекты передаются в функцию по ссылке. Это значит, что бы работаем с исходным значением и можем его изменить. (Обычно так делать не следует).

```
function inc(x) { x.val += 1; }
```

```
let x = { val: 1 }
```

```
inc(x);
```

```
console.log(x.val); // 2
```

Предотвращение случайного изменения

```
function myFunction(arr, obj) {  
  let localArr = arr.slice();  
  let localObj = Object.assign({}, obj);  
  
  // работаем с localArr и localObj,  
  // не боясь испортить исходные данные  
}
```

Область видимости

Областью видимости переменной называют часть программы, в которой к этой переменной можно обратиться

Переменная, объявленная внутри блока, недоступна снаружи этого блока (block scope).

Переменная, объявленная внутри функции, недоступна снаружи этой функции (functional scope).

Function scope

```
function sayHello() {  
  let message = 'Hello world';  
  console.log(message);  
}
```

```
sayHello();           // Hello world  
console.log(message); // ReferenceError
```

Block scope

```
if (a > b) {  
    let message = 'a is great!';  
}
```

```
console.log(message); // ReferenceError
```

var: только function scope

```
if (a > b) {  
    var message = 'a is great!';  
}
```

```
console.log(message); // a is great!
```

Всплытие (Hoisting)

```
var message;
```

```
if (a > b) {  
    message = 'a is great!';  
}
```

```
console.log(message); // a is great!
```


var: только function scope

```
if (a > b) {  
    var message = 'a is great!';  
}  
  
console.log(message); // a is great!
```

Immediate Invoked Function Expression



```
if (a > b) {  
    (function () {  
        var message = 'a is great!';  
    })(); // IIFE  
}
```

```
console.log(message); // ReferenceError
```

Определение типа – typeof

Для определения типа примитива используется выражение typeof.

```
let age = 10;  
console.log(typeof age) // 'number'
```

```
let name = 'Roman';  
console.log(typeof x) // 'string'
```

Определение типа – typeof

undefined	'undefined'
null	'object'
Boolean value	'boolean'
Number value	'number'
String value	'string'
Function	'function'
Всё остальное	'object'

Определение типа – typeof

По историческим причинам typeof считает null за объект.
Поэтому при проверке на объект нужно явно исключать null.

```
if (typeof x === 'object' && x !== null) {  
    console.log( 'Несомненно, объект!' );  
}
```

Определение типа – instanceof

instanceof используется для проверки типов значений, созданных с помощью конструкторов (о конструкторах будет отдельная лекция).

```
let date = new Date();  
let error = new Error();  
  
console.log(date instanceof Date); // true  
console.log(error instanceof Date); // false
```

Прочие проверки

```
Array.isArray([]) // true
```

```
Array.isArray({}) // false
```

```
Object.is(NaN, NaN) // true
```

```
Object.is(-0, +0) // false
```

Неявное приведение типов

При вычислении операции над значениями разных типов, интерпретатор постарается привести их к общему типу. Неявного приведения типов следует избегать.

```
20 - '5' // 15
```

```
20 == '20' // true
```


Правила нестрогого сравнения

	true	false	1	0	-1	"true"	"false"	"1"	"0"	"-1"	""	null	undefined	Infinity	-Infinity	[]	{}	[]]	[0]	[1]	NaN
true																					
false																					
1																					
0																					
-1																					
"true"																					
"false"																					
"1"																					
"0"																					
"-1"																					
""																					
null																					
undefined																					
Infinity																					
-Infinity																					
[]																					
{}																					
[]]																					
[0]																					
[1]																					
NaN																					

Правила нестрогого сравнения

Всегда используйте строгое сравнение!

Явное приведение типов

Для приведения значения к примитивному типу используются функции Number, String, Boolean.

```
let str = '100';
```

```
let num = Number(str); // 100
```

```
let str2 = String(num); // '100'
```

truthy и falsy

Boolean приводит к false только пустую строку, 0, null, undefined, NaN и сам false. Всё остальное приводится к true.

```
Boolean(false)    // false
```

```
Boolean('false')  // true
```

```
Boolean(0)        // false
```

```
Boolean([])       // true
```

Из массива в строку и обратно

```
let fruitString = 'apple, banana';
```

```
let fruitArray = fruitString.split(', ');  
console.log(arr); // ['apple', 'banana']
```

```
let fruitString2 = fruitArray.join(';');  
console.log(fruitString2); // 'apple;banana'
```

Обход значений массива или строки

```
let fruits = ['apple', 'banana'];  
  
for (let fruit of fruits) {  
    console.log(fruit); // 'apple', 'banana'  
}
```

Обход индексов массива или строки

```
let fruits = ['apple', 'banana'];  
  
for (let i = 0; i < fruits.length; i += 1) {  
    let x = fruits[i];  
    console.log(x); // 'apple', 'banana'  
}
```

Обход ключей объекта

```
let me = { name: 'Roman', age: 28 };  
let keys = Object.keys(me);  
  
for (let key of keys) {  
    console.log(key); // 'name', 'age'  
}
```


Обход значений объекта

```
let me = { name: 'Roman', age: 28 };  
let values = Object.values(me);  
  
for (let value of values) {  
    console.log(value); // 'Roman', 28  
}
```

Обход объекта



```
let me = { name: 'Roman', age: 28 };

for (let key in me) {
  if (me.hasOwnProperty(key)) {
    console.log(key); // 'name', 'age'
  }
}
```

Demo



Конец