Основы JS

Инструкции, выражения и операторы

Директива "use strict";

- Не ключевое слово, а строковый литерал
- Может использоваться в начале сценария или в начале тела функции
- Изменяет многие правила работы интерпретатора на более строгие
- Вводятся новые зарезервиронные слова (`interface, let, yield, package etc`)
- Запрещается использовать необъявленные переменные
- Запрещается использовать инструкцию `with`
- Запрещается повторное объявление свойств в литералах объектов и параметров функций
- etc, etc, etc (порядка 20 мелких улучшений и правок)

Объявление переменных: `var`

- Не нужно указывать тип переменной (динамическая типизация)
- Неинициализированная переменная имеет значение `undefined`
- `var` в глобальной области видимости => свойство глобального объекта
- Можно объявить переменную несколько раз (но зачем?)
- Обращение к необъявленной переменной => `ReferenceError`
- (*strict*) Присвоение необъявленной переменной => `ReferenceError`

```
'use strict';
var i;
console.log(i);  // => undefined

var a = 15;
var a = 19, b = 23;
console.log(a);  // => 19

c = 19;          // ReferenceError
console.log(d);  // ReferenceError

var RegExp = null; // whoops!
```

Область видимости переменных (scope): `var`

- Область видимости переменной (scope) - функция
- Всплытие объявления переменной (hoisting)

```
'use strict';
var scope = 'global';    // Глобальная переменная

function f() {
    var scope = 'local';       // Локальная переменная
    console.log(scope);        // => 'local' - локальная переменная затенила глобальную
    console.log(window.scope); // => 'global' (в браузере)
}
f();
```

```
var scope = 'global';
function f() {
    console.log(scope);   // => undefined (?)
    var scope = "local";
    console.log(scope);   // => 'local'
}

f();
```

```
var scope = 'global';
function f() {
    var scope;            // всплытие
    console.log(scope);   // => undefined
    scope = 'local';      // инициализация
    console.log(scope);   // => 'local'
}
f();
```

Цепочки областей видимости

- Глобальные переменные - свойства глобального объекта
- Локальные переменные - свойства некоего скрытого объекта
- Каждый фрагмент кода имеет цепочку областей видимости (*scope chain*)
- Scope chain используется при разрешении имени переменной

```
'use strict';
var a = 'a0', b = 'b0', c = 'c0';

function f1() {
    var a = 'a1', b = 'b1';

    function f2() {
        var a = 'a2';

        console.log(a, b, c); // => a2, b1, c0
    }

    f2();
}
f1();
```

Объявление переменных: `let, const`

- Область видимости ограничена функцией / блоком `{}`
- `let` в глобальном scope не добавляет свойство глобального объекта
- Повторное объявление переменной => `SyntaxError`
- `const` объявляет константу, обязательная инициализация

```
let count = 40;
let count = 50;     // SyntaxError, повторное объявление
if (...) {
    let count = 50; // ok, новая область видимости
}

const x = 10;
x = 15;             // TypeError, константу нельзя изменить
const y;            // SyntaxError, константа должна быть инициализирована

let RegExp = null;
window.RegExp;      // function ...
```

Temporary Dead Zone (TDZ)

- Объявления `let/const` не всплывают к верху блока
- При анализе блока `{}` все `let/const` переменные помещаются в TDZ
- Обращение к переменной, находящейся в TDZ => `ReferenceError`
- Переменная удаляется из TDZ когда выполнение доходит до ее объявления

```
typeof value;       // 'undefined'; мы вне блока if (...), поэтому value еще нет в TDZ
if (...) {
    console.log(typeof value); // ReferenceError, value в TDZ
    let value = "blue";        // value удаляется из TDZ
    console.log(typeof value); // 'string'
}
```

`var` vs `let, const`

	`var`	`let, const`
Scope	Функция	Блок
Перезаписывает системные переменные	+	-
Использование переменной до объявления	+	-
Повторное объявление переменной	+	-
Hoisting (всплытие)	+	-

- Во всех случаях `let, const` предпочтительнее `var`
- Есть подход, при котором всегда используется `const`, а `let` - только если переменная предназначена для изменения

Оператор эквивалентности (a === b)

```
5 === '5';       // false, разные типы
null === null;   // true
undefined === undefined; // true
true === true;   // true
false === false; // true
NaN === NaN;     // false (!) нужно использовать Number.isNaN()
16.94 === 16.94; // true, числа с одинаковыми значениями
0 === -0;        // true
'abc' === 'abc'; // true; строки одинаковой длины с одинаковыми 16-битными значениями

[1, 2, 3] === [1, 2, 3];  // false, ссылки на разные объекты
let a = [1, 2, 3], b = a;
a === b;                  // true, ссылки на один и тот же объект
```

Логические операторы &&, ||

- Работают на нескольких "уровнях"
- Работают со значениями любых типов, базируясь на их "истинности"
- "Ленивое" выполнение - правый операнд может быть не вычислен
- В качестве результата возвращается один из операндов

```
true && false;     // false
true || false;     // true

let obj = {x: 1};
let p = null;
obj && obj.x;      // 1
p && p.x;          // null

flag && doSomething(); // эквивалентно if (flag) doSomething();

let max = maxWidth || preferences.maxWidth || DEFAULT_MAX_WIDTH;
value = value || 0;
```

Условный (тернарный) оператор `?:`

- Синтаксис `cond ? expr1 : expr2;`
- Вычисляется одно из выражений в зависимости от истинности условия
- Отличие от `if` в том, что тернарный оператор возвращает значение

```
return x > 0 ? x : -x;

let greeting = "hello " + (username ? username : "there");

value = value === undefined ? value : 0;
```

Оператор `typeof`

- Синтаксис `typeof varName`
- Возвращает строку, описывающую тип значения переменной

```
typeof undefined;     // 'undefined'
typeof null;          // 'object' (!)
typeof true;          // 'boolean'
typeof 56;            // 'number'
typeof 'abc';         // 'string'

function f() {};
typeof f;             // 'function'
typeof {};            // 'object'
typeof new Number(5); // 'object'
```

Оператор `in` - существование свойства

- Синтаксис `prop in obj`
- `prop` - строка, `obj` - объект, возвращает `true/false`

```
let trees = ['redwood', 'bay', 'cedar', 'oak', 'maple'];
0 in trees        // true
6 in trees        // false
'bay' in trees    // false
'length' in trees // true

'PI' in Math      // true

let obj = {
    x: 19,
    y: undefined
};
'x' in obj;       // true
obj.y;            // undefined
obj.z;            // undefined
'y' in obj;       // true
'z' in obj        // false
```

Оператор `instanceof`

- Проверяет принадлежность объекта классу
- Синтаксис `obj instanceof ConstructorName`
- `obj` - объект, `ConstructorName` - имя функции-конструктора
- Проверяется вся цепочка прототипов

```
let simpleStr = 'This is a simple string';
let myString  = new String('String in object wrapper');
let myObj     = {};

simpleStr instanceof String; // false
myString  instanceof String; // true
myString  instanceof Object; // true
myObj instanceof Object;     // true
```

Инструкции ветвления

- `if (cond) statement1 [else statement2]`
- `switch (cond) { statements }`
- `switch` сравнивает значения без приведения типов (как и ===)

```
if (x < 0)
    return -x;
else
    return x;

switch (n) {
    case 1:
        console.log('1');
        break;
    case 2:
        console.log('2');
        break;
    ...
}
```

```
let animal = 'Giraffe';
switch (animal) {
    case 'Cow':
    case 'Giraffe':
    case 'Dog':
    case 'Pig':
        console.log('Will go on Noah\'s Ark.');
        break;
    case 'Dinosaur':
    default:
        console.log('Will not.');
}
```

Циклы while

- while (cond) statement
- do statement while (cond)

```
let n = 17;
while (n % 2) {
    console.log(n);     // 17
    n = (n - 1) / 2;
}
console.log(n);         // 8

let m = 14;
do {
    console.log(m);     // 34, 17
    m = (m % 2) ? (m - 1) : m;
    m /= 2;
} while (m % 2);
console.log(m);         // 8
```

Циклы for

- for ([init]; [cond]; [final-expr]) statement
- for (variable in obj) statement - проход по свойствам объекта

```
let arr = [1, 3, 5, 7, 9];
for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}

let obj = {a: 1, b: 2};
for (let prop in obj) {
    console.log(prop, obj[prop]);  // => 'a' 1; 'b' 2;
}
```

Цикл `for...of`

- `for (variable of iterable) statement`
- Выполняет проход по значениям *итератора*
- Итераторы существуют для `Array, String, Map, Set` etc
- Несколько вариантов итераторов: `values(), keys(), entries()`

```
let arr = ['a', 'b', 'c'];
for (let val of arr) console.log(val);           // 'a', 'b', 'c'
for (let val of arr.keys()) console.log(val);    // 0, 1, 2
for (let val of arr.entries()) console.log(val);
// [0, 'a'], [1, 'b'], [2, 'c']
```

Метки и переходы

- Метка - любой идентификатор, в т.ч. совпадающий с существующими
- `break` - выход из текущего цикла или конструкции `switch`
- `continue` - немедленный переход к следующей итерации цикла
- `return` - выход из текущей функции и возврат значения
- `return` может находиться только в теле функции

```
mainloop: while(token != null) {
    ...
    continue mainloop;
    ...
}

function square(x) { return x*x; }
```

Обработка исключительных ситуаций: `throw/catch/finally`

- Код, который может вызвать ошибку, заключается в блок `try`
- При возникновении ошибки управление передается в блок `catch`
- Блок `finally` выполняется всегда, в т.ч. при выходе через `return`
- `catch/finally` - опциональны
- 8 встроенных типов ошибок, можно объявлять свои

```
try {
    ...
    throw new Error('oops');
    ...
} catch (e) {
    if (e instanceof Error)
        console.log('its an Error!');
    console.error(ex.message);
} finally {
    console.log('finally');
    return;
}
```

Деструктурирущее присваивание

- Извлечение значений из массива/объекта в отдельные переменные
- В лишние переменные записывается `undefined`
- Можно пропускать ненужные значения
- Можно указывать значения по умолчанию

```
let [fName, lName] = ['James', 'Bond'];
console.log(fName, lName); // 'James Bond'

let fName = 'simply';
let [, lName] = ['James', 'Bond'];
console.log(fName, lName); // 'simply Bond'

let [fName = 'Anonymous', lName = 'Mighty'] = [];
console.log(fName, lName); // 'Mighty Anonymous'

[a, b] = [b, a]; // Обмен значениями
```

```
let {w, h} = {w: 100, h: 200};
console.log(w, h);          // => 100 200

let {w: width, h: height} = {w: 100, h: 200};
console.log(width, height); // => 100 200

let w, h;
({w, h = 300} = {w: 100});
console.log(w, h);          // => 100 300

let {width, height, point: [x, y]} = {
    width: 100,
    height: 200,
    point: [40, 50]
};
console.log(
    width, height, x, y); // => 100 200 40 50
```

Rest и Spread: . . .

- Rest: список значений => массив
- Rest-переменная всегда должна быть последней в списке
- Spread: итератор => список значений

```
// REST
let [a, ...b] = [1, 2, 3];
console.log(a, b);          // => 1 [2, 3];

// создание "клона" easy-way
let colors = [ "red", "green", "blue" ];
let [ ...clonedColors ] = colors;

function f(a, ...etc) {
    console.log(a, etc);    // => 1 [2, 3]
}
f(1, 2, 3);

function f(a, ...[b, c]) { // Serious business!
    console.log(a, etc);    // => 1 2 3
}
f(1, 2, 3);
```

```
// SPREAD
// конкатенация массивов easy-way
let colors = ['red', 'green'];
let newColors = ['yellow', ...colors, 'blue'];

// еще конкатенация
let colors1 = ['red', 'green'],
    colors2 = ['blue', 'yellow'];
let allColors = [...colors1, ...colors2];

// Вместо Math.max(arr[0], arr[1], ...);
let arr = [1, 2, 3, 4, 5];
Math.max(...arr);

// Преобразование итератора в массив
let myArray = [1, 'abc', true];
let entries = [...myArray.entries()];
// [[0, 1], [1, 'abc'], [2, true]];
```