Яндекс

Типы данных (часть 1)

Наталья Дружинина, разработчик интерфейсов

План

- Примитивы
- Числа
- Строки
- Массивы
- Объекты
- Функции

Примитивы

Примитивы — это данные, которые не являются объектом и не имеют свойств и методов.

- boolean
- undefined
- null
- number
- string

В JS есть 6 типов данных. Из них 5 относятся к примитивным типам.
Это типы boolean, undefined, null, number, string, symbol. Примитивы это данные, которые не являются объектом и не имеют свойств и методов. В этой лекции мы рассмотрим все типы, кроме symbol.

Объекты

〉 Object
- Array
- Function

Все остальные данные в JS - это объекты. В том числе, массивы и функции тоже являются разновидностью объектов.

Boolean (логический)

```
const trueValue = true;

const falseValue = false;

typeof trueValue; // "boolean"
```


Начнем изучение типов данных с примитивов, с типа Boolean. В предыдущей лекции вам уже немного рассказывали про типы данных. В этой лекции мы немного повторим пройденное, и разберем каждый тип подробнее.

Булевый тип может принимать только два значения, true или false.

К значению false преобразуется 0, пустая строка, null, undefined, NaN. Все остальные значения преобразуются к true.

Для определения типа данных переменной в рантами используется оператор typeof. Он возвращает строку с названием типа данных. В данном случае, для переменной со значение true typeof вернет тип boolean.

Boolean

Неявное (нестрогое) перед сравнением приводит типы к одному,
после сравнивает
Явное (строгое) не преобразует — предпочтительнее использовать явное

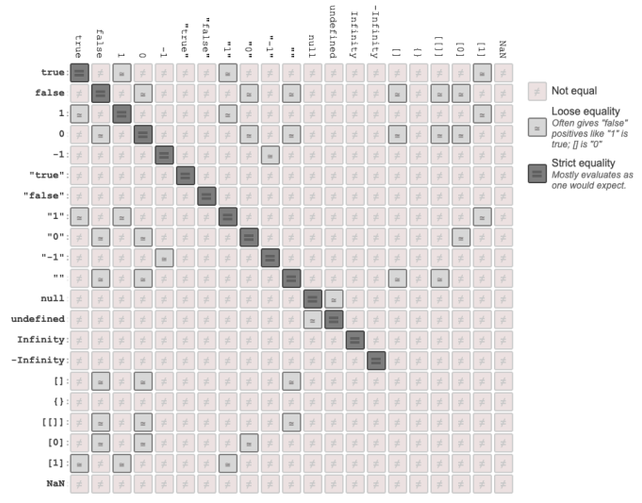
```
// Неявное сравнение
'1' == 1; // true


// Явное сравнение
1 === 1; // true
'1' === 1; // false
```


Результатом сравнения всегда будет true или false. При неявном сравнении происходит приведение типов к одному, после чего производится сравнение. При этом нужно помнить правила приведения, которые не всегда очевидны.

При явном сравнении приведения типов не происходит. Всегда лучше использовать явное сравнение, если у вас нет объективных причин использовать неявное.

Сравнение



Javascript equality table

Если вы хотите больше узнать о явном и неявном сравнении, то посмотрите эту таблицу. Тёмно серым отмечены равенства при явном сравнении. Светло-серым - при неявном. По таблице хорошо видно, что для неявного сравнения есть много нюансов.

Логические операции

```
// Логическое И
true && false;   // false

// Логическое ИЛИ
true || false;   // true

// Логическое НЕ (отрицание)
!false;          // true

// Сокращённое вычисление
value || getValue();
shouldRequest && doRequest();
```


Как и в других языках программирования, в javascript есть логические операторы. Их три - И, ИЛИ и НЕ. Они могут применяться не только к нулевым значениям, но и к любым другим. При этом для вычисления результата произойдет приведение типа к булевому.
Логические операции в JS часто используются и для выполнения кода по условию. Это работает за счет особенностей вычисления операций ИЛИ и И.
Оператор ИЛИ вычисляет операнды слева направо. Каждый операнд он конвертирует в логическое значение. Если оно равно true, то выполнение прекращается. Если false, то вычисляется следующий операнд.
Оператор И работает схожим образом. Он прекращает вычисление, если текущий операнд был приведен к значению false.

Логические операции

```
let value = 0;

/*
0 приводится к false
функция будет вызвана
*/
value || getValue();
```


При этом важно не забывать про логику приведения типов. Что, если в предыдущем примере значение value будет равно нулю? 0 при приведении к булевому типу превращается в false. Поэтому будет выполнена вторая часть условия. Ошибки такого рода встречаются достаточно часто, поэтому будьте осторожны при использовании сокращённых вычислений.

undefined

Специальный тип данных, имеющий смысл «значение не было присвоено». Явно присваивать не рекомендуется

```
let notDefined; // undefined

typeof notDefined; // "undefined"
```


Про типы данных undefined и null вы уже слышали на прошлой лекции. Undefined это специальный тип данных, имеющий смысл «значение не было присвоено». Явно присваивать его в переменную можно, языком это не запрещено, но делать так не рекомендуется.
Оператор typeof для переменной со значением undefined вернет undefined, потому что это отдельный тип данных.

null

Специальный тип данных, имеющий смысл «ничего», «пусто», «значение неизвестно».

```
let thereIsNoValue = null;

typeof thereIsNoValue; // "object" (!)
// не используйте typeof для проверки на null

if (thereIsNoValue === null) {...}
```


Null это тоже отдельный тип данных. Означает «ничего», «пусто», «значение неизвестно». Предназначен для явного присвоения.
С типом null есть одна интересная особенность. Оператор typeof для него возвращает не null, как можно было бы ожидать. А object. Хотя тип данных null объектом не является.
Это баг языка. В ES5 было предложение его поправить, но оно было отклонено. Поскольку при его фиксе работа очень многих сайтов, код которых использовал эту особенность, сломалась бы.
Вывод 1: языки программирования пишут тоже люди, поэтому они могут содержать ошибки или нелогичные моменты.
Вывод 2: перед тем, как фиксить баг, подумайте, будет ли это безопасно и не сломает ли обратную совместимость.

Numbers (числа)

```
const intPositive = 1;
const intNegative = -1;

const floatPositive = 45.3544;
const floatNegative = -0.378;

const exp = 5e2; // 500
const hex = 0xFF; // 255
```


Следующий тип данных, который мы рассмотрим, это Number, или числовой тип. В javascript, в отличие от многих других языков программирования, нет различия между целыми и вещественными числами. Помимо привычного представления, числа можно записывать также в экспоненциальном виде и в других системах счисления, например в шестнадцатиричной.

Базовая математика

```
5 + 2 = 7; // сложение
5 - 2 = 3; // вычитание
5 * 2 = 10; // умножение
5 / 2 = 2.5; // деление
5 % 2 = 1; // остаток от целочисленного деления
5 ** 2 = 25; // возведение в степень
```


Операции базовой математики особо не отличаются от других языков. Есть сложение, вычитание, умножение, деление, деление по модулю и возведение в степень.

Математические функции (Math)

```
Math.abc(-5); // 5 (абсолютное значение)

Math.floor(7.43); // 7 (округление к меньшему)
Math.ceil(7.43); // 8 (округление к большему)
Math.round(7.43); // 7 (округление к ближайшему)

Math.sqrt(16); // 4 (извлечение квадратного корня)
Math.pow(2, 8); // 256 (возведение в степень)
```

Все методы Math

Для более сложных вычислений в JS есть много математических функций. Они все являются свойствами или методами объекта Math. Например, там есть функции для округления вещественных чисел, для извлечения корня или возведения в степень, и много других. Список всех методов вы сможете посмотреть по ссылке на этом слайде.

Бесконечность (Infinity)

```
1 / 0 === Infinity; // true
-1 / 0 === -Infinity; // true
1 / Infinity === 0; // true

// Проверка на бесконечность
isFinite(10); // true
isFinite(1 / 0); // false
isFinite('0'); // true

// Проверка на бесконечность без преобразования к числу
Number.isFinite('0'); // false
```

В JS есть несколько специальных числовых значений. Если число превышает самое большое представимое значение, то ему присваивается значение Infinity, то есть бесконечность. Если отрицательное число становится меньше самого маленького представимого, то получается -Infinity.
Бесконечности ведут себя ровно так же, как в математике. Например, мы можем поделить на 0 и получить бесконечность.
Для проверки на бесконечность есть специальная функция isFinite. Причем можно вызвать глобальную функцию isFinite, и тогда она сначала приведет аргумент к числу, после чего выдаст результат. А можно воспользоваться методом isFinite у объекта Number, который проводит проверку без преобразования. И выдает true только для конечных числовых значений.

Not a Number (NaN)

```
let notNum = Math.sqrt(-1); // NaN

// Сравнение с NaN
notNum !== notNum; // true

isNaN(notNum); // true
isNaN('I am string'); // true

Number.isNaN('I am string'); // false
```


Еще одно специальное значение, это значение Not a Number, не число. Оно возвращается, если математическая операция не может быть вычислена. Оно не равно ни одному другому числу, в том числе самому себе. Эту особенность можно использовать для проверки на не число. Но лучше использовать функцию isNaN, которая, как и isFinite, существует в двух вариантах. Рекомендуется использовать вариант без приведения типов, чтобы избежать неожиданных ошибок.

Стандарт чисел

64-bit двойной точности (double), соответствуют стандарту IEEE 754

```
Number.MAX_VALUE; // 1.79e+308
Number.MIN_VALUE; // 5e-324, самое близкое к нулю

Number.MAX_SAFE_INTEGER;  // 2^53 - 1 или 9007199254740991
Number.MAX_SAFE_INTEGER + 1 === Number.MAX_SAFE_INTEGER + 2; // true

Number.MIN_SAFE_INTEGER;  // -(2^53 - 1) или -9007199254740991
Number.MIN_SAFE_INTEGER - 1 === Number.MIN_SAFE_INTEGER - 2; // true

Number.isSafeInteger(Math.pow(2, 54)); // false
Number.isSafeInteger(-Math.pow(2, 54)); // false
```

Хранение чисел в V8 и проблемы производительности

Все числа в js хранятся в 64-битном формате, этот же формат называется double в других языках программирования. В языке определены связанные с этим способом хранения константы. MAX_VALUE и MIN_VALUE содержат максимальное и наименьшее (самое близкое к нулю) представимые значения. Константы MAX_SAFE_INTEGER и MIN_SAFE_INTEGER - максимальное и минимальное безопасное целое число. Эти константы являются свойствами объекта Number.

Особенности вычислений

```
0.1 + 0.2 === 0.3; // false

0.1 + 0.2 === 0.30000000000000004; // true

Number.EPSILON; // 2^-52

Math.abs((0.1 + 0.2) - 0.3) < Number.EPSILON; // true
```


С этим стандартом хранения связаны важные особенности вычислений. Вещественных чисел бесконечно много, а размер памяти под число ограничен 64 битами. Из этого очевидно следует, что не любое вещественное число представимо в js, и числа будут хранится с некоторой погрешностью.
Самый простой пример, который демонстрирует погрешность, это сложение 0.1 + 0.2. Мы ожидаем получить 0.3, но на самом деле получаем очень похожее, но не равное ему число. Поэтому проверка равенства через === не сработает. Для сравнения результатов математических операций есть специальная константа EPSILON, которая представляет очень маленькое число. Мы можем проверить, что результат операции отличается от ожидаемого не больше, чем на значение EPSILON.

Strings (строки)

Строки представляют собой последовательность unicode-символов и используются для представления текста.

Следующий тип данных, который мы рассмотрим, это строки. Строки представляют собой последовательность unicode-символов и используются для представления текста.
В отличие от других языков программирования, в JS нет символьного типа данных, как например в C++ или Java. Одиночный символ, это просто строка длины 1.

Создание строки

```
const emptyString = ''; // пустая строка

// Длина строки
emptyString.length; // 0
```


Чтобы создать строку, достаточно поместить текст между парой одинаковых кавычек. Кавычки без текста внутри создадут пустую строку. Узнать длину строки можно с помощью свойства length. В этом примере мы получили строку длины 0.

Создание строки

```
// Можно использовать одинарные кавычки
const russianString = 'строка текста';

russianString.length; // 13

// Можно использовать двойные кавычки
const russianString = "строка текста";

russianString.length; // 13
```


Можно использовать одинарные, двойные или обратные кавычки. В ваших приложениях лучше использовать единый стиль написания строк. В рамках лекции будем использовать одинарные кавычки, либо обратные, если потребуется интерполяция.
Вопрос, не видите ли вы здесь противоречий с определением примитива?

Объектные обертки (boxing)

Для string, number, boolean существуют объектные обертки String, Number, Boolean, позволяющие использовать свойства и методы у примитивов
Для undefined и null оберток нет, у них нет свойств и методов

```
let simpleStr = 'simple';

let upperStr = simpleStr.toUpperCase();
// SIMPLE
```


У примитивов действительно нет свойств и методов. Но для чисел, строк и булевых примитивных значений есть соответствующие им объектные обертки, у которых методы и свойства есть. При вызове метода у примитива он преобразуется в объект, метод исполняется, после чего преобразуется обратно в примитив.

Экранирование

```
const escapeCodesString = 'a\'b'; // a'b

escapeCodesString.length; // 3

const escapeCodesString = "a\"b"; // a"b

escapeCodesString.length; // 3
```


Некоторые символы в строках нужно экранировать, например, кавычки. Без экранирования кавычка будет означать конец строки. Символы экранируются с помощью обратного слэша. На длину строки это не влияет.

Специальные символы

```
const escapeCodesString = 'a\\b'; // a\b

escapeCodesString.length; // 3

const escapeCodesString = 'a\n\tb'; // a
                                    //     b

escapeCodesString.length; // 4
```

Символ обратного слэша экранируется всегда. С помощью escape-последовательностей можно записывать в строку спецсимволы, такие как перенос строки, табуляция и так далее.

Неизменяемость строк

Все примитивы в JavaScript, в том числе строки, неизменяемы (immutable)

```
const russianString = 'кот';

// Возможно обращение к символу по индексу
russianString[1]; // 'о'

// Редактирование невозможно
russianString[1] = 'и';
russianString; // 'кот'
```

Unicode

Строки всегда хранятся в кодировке UTF-16

```
// Поддерживаются все символы из unicode
const utfString = '中文 español English বাংলা 日本語 ਪੰਜਾ';

const emoji = '🤔';

// Некоторые unicode-символы имеют длину больше 1
// Будьте осторожны при обращении по индексу
emoji.length; // 2
emoji[0]; // 
emoji[1]; // 
```

"💩".length === 2

Строки в JS всегда хранятся в кодировке UTF-16, что позволяет использовать символы из различных языков, таких как китайский, русский и тд.

Конкатенация и интерполяция

Интерполяция предпочтительнее

```
const world = 'World';
const concatString = 'Hello, ' + world + '!'; // Hello, World!

const world = 'World';
const concatString = `Hello, ${world}!`; // Hello, World!
```

Операции со строками. slice

```
// Обрезаем строку до 20 символов
const longString = `Очевидно проверяется, что математический анализ
существенно масштабирует интеграл по поверхности, что неудивительно.
Первая производная, очевидно, позиционирует
ортогональный определитель.`;

const shortString = longString;

if (longString.length > 20) {
  shortString = longString.slice(0, 20);
}

shortString; // 'Очевидно проверяется'
shortString.length; // 20
```

Поиск в строке

```
const tweet = `PWA. Что это такое? Третий доклад на WSD в Питере
Сергея Густуна #wstdays`;

// Находим индекс первого вхождения подстроки в строке
tweet.indexOf('#wstdays'); // 65

// Искомая подстрока отсутствует
tweet.indexOf('#holyjs'); // -1
```

Поиск в строке

```
const tweet = `PWA. Что это такое? Третий доклад на WSD в Питере
Сергея Густуна #wstdays`;

tweet.includes('#wstdays'); // true
tweet.includes('#holyjs'); // false
```

Сравнение строк

```
'hi' === 'hi'; // true

'a' < 'b';     // true
'a' < 'ab';    // true
'bar' < 'foo'; // true

'1' > '12';    // false
'2' > '12';    // true
'12' < '5';    // true
```

Сравнение строк. localeCompare

```
const animals = ['Мышь', 'Ёж', 'Лис', 'Волк'];
animals.sort();
animals; // ['Ёж', 'Волк', 'Лис', 'Мышь']

'Ё' < 'В'; // true
'Ё' < 'М'; // true

'Ё'.localeCompare('В'); // 1
'Ё'.localeCompare('М'); // -1

animals.sort(function (str1, str2) {
  return str1.localeCompare(str2);
});
animals; // ["Волк", "Ёж", "Лис", "Мышь"]
```

Преобразование строки в число

```
Number('123');      // 123
Number('12.8');     // 12.8
Number('12.8  ');   // 12.8
Number('  12.8');   // 12.8
Number('    ');     // 0
Number('');         // 0
Number('12.8s')     // NaN
Number('s12.8')     // NaN
```

Преобразование строки в число

```
parseFloat('123');      // 123
parseFloat('12.8');     // 12.8
parseFloat('12.8  ');   // 12.8
parseFloat('  12.8');   // 12.8
parseFloat('   ');      // NaN
parseFloat('');         // NaN
parseFloat('12.8s');    // 12.8
parseFloat('s12.8');    // NaN
```

```
/* Вторым аргументом указывается
   система счисления
*/
parseInt('123', 10); // 123
parseInt('12.8', 10); // 12
parseInt('12.8   ', 10) // 12
parseInt('   12.8', 10); // 12
parseInt('', 10); // NaN
parseInt('12.8s', 10); // 12
parseInt('s12.8', 10); // NaN
```

Преобразование строки в массив

```
const str = '1.2.3';

// ["1", ".", "2", ".", "3"]
const arr = str.split('');

// ["1", "2", "3"]
const anotherArr = str.split('.');

'🤔'.split(''); // ["", ""]
```

Функции для работы со строками

toLowerCase - приводит строку к нижнему регистру
toUpperCase - приводит строку к верхнему регистру
trim - удаляет пробельные символы с обеих сторон
startsWith - начинается ли строка с подстроки
endsWith - заканчивается ли строка подстрокой

Все функции для работы со строками

Arrays (массивы)

Массивы — спископодобные структуры, позволяющие хранить произвольные наборы данных, и содержащие методы для работы с этими данными.

Создание массива

```
// Пустой массив
const emptyArray = [];

emptyArray.length; // 0

// Массив чисел
const arrayOfNumbers = [1, 2, 3, 4];

arrayOfNumbers.length; // 4

// Массив строк
const arrayOfStrings = ['a', 'b', 'c'];

arrayOfStrings.length; // 3
```

Создание массива

```
// то же, что и emptyArr = []
const emptyArr = new Array();
arr.length; // 0

// пустой массив длины 5
const arrFive = new Array(5);
arr.length; // 5

// то же, что и mixedArr = [1, true, 'text']
const mixedArr = new Array(1, true, 'text');

// ['t', 'e', 'x', 't']
const textArr = Array.from('text');
```

Внутреннее устройство массива (eng)

Проверка на массив

```
const arr = [1, 2, 3];

// не используйте typeof для проверки
typeof arr; // "object"

Array.isArray(arr); // true
```

Добавление в массив. Push

```
const emptyArray = [];

// Добавляем элементы
emptyArray.push('a');
emptyArray.push('b');
emptyArray; // ['a', 'b']

emptyArray.length; // 2
```

Удаление из массива. Pop

```
const someArray = ['a', 'b'];

// Удаляем последний элемент
someArray.pop();  // 'b'
someArray;        // ['a']

someArray.length; // 1
```

Добавление в массив. Unshift

```
const emptyArray = [];

emptyArray.unshift(1);
emptyArray.unshift(2);

emptyArray; // [2, 1]
```

Удаление из массива. Shift

```
const someArray = ['a', 'b'];

someArray.shift(); // 'a'
someArray; // ['b']

someArray.length; // 1
```

Удаление из массива. Delete

Не меняет длину массива

```
let arr = [1, 2, 3, 4];

delete arr[2];

arr; // [1, 2, empty, 4]
```

Операции с массивами

〉 Некоторые операции мутируют исходный массив, некоторые нет
〉 Список мутирующих:
- copyWithin
- fill
- pop
- push
- shift
- unshift
- reverse
- sort
- splice

Объединение массивов

```
const arrayOfNumbers = [1, 2, 3, 4];
const arrayOfStrings = ['a', 'b', 'c'];

const concatedArray = arrayOfNumbers.concat(arrayOfStrings);

concatedArray;  // [1, 2, 3, 4, 'a', 'b', 'c']

arrayOfNumbers; // [1, 2, 3, 4]
arrayOfStrings; // ['a', 'b', 'c']
```

Итерирование по массиву

```
const items = [1, 2, 4, 10, 9000];

for (let i = 0; i < items.length; i++) {
  let value = items[i];
  // ...
}

items.forEach(function (value, i) {
  // ...
});

for (const item of items) {
  // ...
}
```

Дыры в массивах

```
const items = [1, , 3];
items; // [1, empty, 3]
items.length; // 3

const forIndexes = [];
for (let i = 0; i < items.length; i++) {
  forIndexes.push(i);
}
forIndexes; // [0, 1, 2]

const forEachIndexes = [];
items.forEach(function (value, item) {
  forEachIndexes.push(item);
});
forEachIndexes; // [0, 2] - игнорирует дыры
```

Поиск в массиве. indexOf

```
const array = [1, 2, 3, 100, 2, 200];

// возвращает индекс первого найденного элемента
array.indexOf(2); // 1

// Если значение не найдено, возвращает -1
array.indexOf(10000); // -1

// использует строгое сравнение
array.indexOf('2'); // -1

// Второй аргумент - индекс, с которого начинать поиск
array.indexOf(2, 4); // 3
```

Поиск в массиве. find

```
const array = [1, 2, 3, 100, 2, 200];

const foundValue = array.find(function (item) {
  return item > 3;
})
foundValue; // 100
const notFoundValue = array.find(function (item) {
  return item > 100000;
});
notFoundValue; // undefined
```

Операции с массивами. Slice

```
const items = [1, 2, 3, 4, 5, 6];

// Копируем в новый массив
const itemsCopy = items.slice(); // [1, 2, 3, 4, 5, 6]

// Копируем часть массива
const itemsPart = items.slice(2, 4); // [3, 4]

// Исходный массив не меняется
items; // [1, 2, 3, 4, 5, 6]
```

Операции с массивами. Splice

```
const items = [1, 2, 3, 4, 5, 6];

// Вставляет по индексу 2 новый элемент
items.splice(2, 0, 'new');

// Меняет исходный массив
items; // [1, 2, 'new', 3, 4, 5, 6]

// удаляет 2 элемента, начиная с индекса 3
items.splice(3, 2);

items; // [1, 2, 'new', 5, 6]
```

Сортировка массива

```
const arrayOfNumbers = [4, 1, 1000, 3, -1, 5];

// сортирует исходный массив
arrayOfNumbers.sort();
arrayOfNumbers; // [ -1, 1, 1000, 3, 4, 5 ] (!)
// Сортирует как unicode строки

// Можно передать функцию сортировки
arrayOfNumbers.sort(function (a, b) {
  return a - b;
});
arrayOfNumbers; //  [-1, 1, 3, 4, 5, 1000]
```

Сортировка массива

В спецификации ECMAScript не определен алгоритм сортировки, который должен быть реализован. Поэтому разные движки могут реализовывать разные алгоритмы.

Например, в V8 раньше использовался QuickSort, сейчас перешли на TimSort

Сортировка массивов в V8

Функции для работы с массивами

shift — выталкивает из массива первый элемент и возвращает его
unshift — добавляет элемент в начало массива
reverse — инвертирует порядок элементов в исходном массиве
filter — выбирает элементы исходного массива, удовлетворяющие условию, в новый массив
map — создает новый массив, содержащий преобразованные элементы исходного массива
reduce - преобразует исходный массив к одному новому значению

Все функции для работы с массивами

Objects (объекты)

Объект — неупорядоченный набор пар вида «ключ-значение». Каждая такая пара называется свойством объекта (функции называются методами), каждое свойство должно иметь уникальное имя.

Создание объекта

```
// пустой объект
const emptyObject = {};

// присваивание через точечную нотацию
emptyObject.propertyName = 'foo'; // { propertyName: 'foo' }

// получение значения свойства через точечную нотацию
emptyObject.propertyName; // 'foo'

// присваивание через скобочную нотацию
emptyObject['some-field'] = 'bar';
emptyObject['some-field']; // 'bar'
```

Создание объекта

```
const propName = 'text';
const obj = { [propName]: 'textValue' };

obj; // { text: 'textValue' }
```

Создание объекта

```
const obj = {};
const anotherObj = {field: 'value'};

obj[anotherObj] = 'some text';

// Ключи всегда строки
obj; // {[object Object]: "some text"}

anotherObj.toString(); // [object Object]
```

Удаление свойств объекта

```
const obj = {a: 1, b: 2, c: 3};

// Очень медленный оператор
delete obj.b;
obj; // {a: 1, c: 3}
```

Обращение к свойствам объекта

```
const tweet = {
    id: '782188596690350100',
    text: 'Я и ИоТ, пятый доклад на WSD в Питере Вадима Макеева #wstdays',
    user: {
        id: 42081171,
        name: 'Веб-стандарты',
        screenName: 'webstandards_ru',
        followersCount: 6443
    },
    hashtags: ['wstdays']
};

tweet.id; // '782188596690350100'
tweet.user.screenName; // 'webstandards_ru'
tweet['id']; // 782188596690350100'
```

Итерирование по ключам объекта

```
const keys = Object.keys(tweet);

keys; // ['id', 'text', 'user', 'hashtags']

for (let i = 0; i < keys.length; i++) {
  const key = keys[i];
  const value = tweet[key];
  // Что-то делаем
}

for (let key in tweet) {
    const value = tweet[key];
  // Что-то делаем
}
```

Object.keys предпочтительнее

Проверка наличия свойства у объекта

```
tweet.hasOwnProperty('text'); // true
tweet.hasOwnProperty('nonExistentProperty'); // false

'text' in tweet; // true
'nonExistentProperty' in tweet; // false

tweet.nonExistentProperty; // undefined
```

Functions (функции)

- Функция — это фрагмент исполняемого кода
- Может иметь входные параметры и возвращать значение
- В JavaScript функция является объектом

Декларация функции

```
function getFollowersCount() {
  return 1;
}

typeof getFollowersCount; // "function"
```

Декларация функции

```
function noop() {
}

function noop() {
  return undefined;
}

function noop() {
  return;
}
```

Передача по значению и по ссылке

```
const tweet = {
  user: { followersCount: 1 }
};

function incrementFollowersCount(count) {
  count++;
}

incrementFollowersCount(tweet.user.followersCount);

tweet.user.followersCount; // 1
```

Передача по значению и по ссылке

```
const tweet = {
  user: { followersCount: 1 }
};

function incrementFollowersCount(user) {
  user.followersCountcount++;
}

incrementFollowersCount(tweet.user);

tweet.user.followersCount; // 2
```

Функции - объекты высшего порядка

Они могут быть переданы в другие функции в качестве аргумента, могут быть возвращены из функции, а так же могут иметь личные свойства, как и другие объекты.

Передача функций в качестве аргументов

```
const tweets = [
  { hashtags: ['wstdays'], likes: 16, text: 'Я и IoT, пятый…' },
  { hashtags: ['wstdays', 'mails'], likes: 33, text: 'Вёрстка писем…' },
  { hashtags: ['wstdays', 'html'], likes: 22, text: '<head> — всему…' },
  { hashtags: ['wstdays'], likes: 8, text: 'Доброе утро! WSD…' },
  { hashtags: ['nodejs'], likes: 7, text: 'Node.js, TC-39 и модули,…' },
  { hashtags: ['svg'], likes: 19, text: 'Как прятать инлайновые…' },
  { likes: 9, text: 'Наглядная таблица доступности…' },
];
```

Передача функций в качестве аргументов

```
const result = [];

// Выбираем только твиты с хештегом #wstdays
function filterWithWstdaysHashtag(tweet) {
  const hashtags = tweet.hashtags;
  if (Array.isArray(hashtags) && hashtags.includes('wstdays')) {
    result.push(tweet);
  }
}

// Теперь в result лежат отфильтрованные твиты
tweets.forEach(filterWithWstdaysHashtag);
```

Передача функций в качестве аргументов

```
// Выбираем только твиты с хештегом #wstdays
function filterWithWstdaysHashtag(tweet) {
  const hashtags = tweet.hashtags;
  return Array.isArray(hashtags) && hashtags.includes('wstdays');
}

// Получаем из объекта твита likes
function getLikes(tweet) {
  return tweet.likes;
}

const likesArray = tweets
  .filter(filterWithWstdaysHashtag)
  .map(getLikes);

likesArray; // [16, 33, 22, 8]
```

Передача функций в качестве аргументов

```
const likesCount = likes.reduce(function (sum, item) {
  return sum + item;
}, 0);

likesCount; // 79
```

Итоги. Типы данных

```
// примитивы
typeof true;        // 'boolean'
typeof undefined;   // 'undefined'
typeof null;        // 'object' (!)
typeof 'строка';    // 'string'
typeof 12.4;        // 'number'

// объекты
typeof { baz: 1 }; // 'object'
typeof [1, 2];     // 'object', используйте Array.isArray()

function foo() {}
typeof foo;         // 'function'
```

Яндекс

Спасибо

Наталья Дружинина

Разработчик интерфейсов

dotokoto@yandex-team.ru