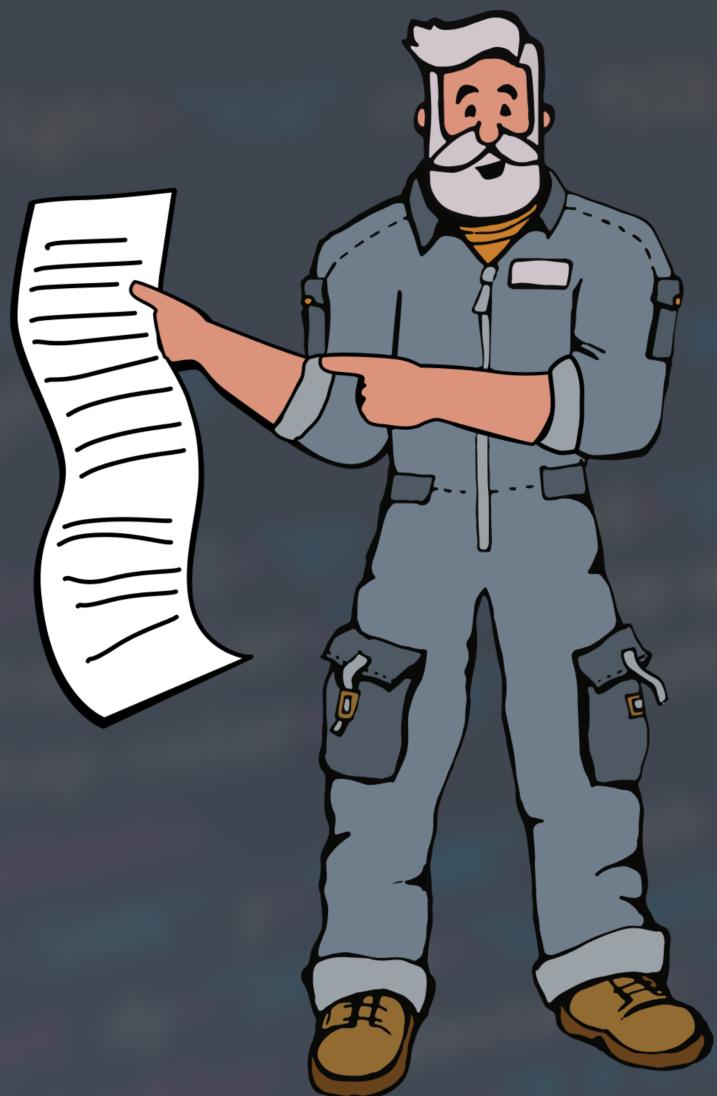


JavaScript

Собеседование

#40



Coderslang Master

Вступление

Привет! **Спасибо** за загрузку этой электронной книжки. Я написал ее, чтобы помочь тебе **подготовиться к техническому собеседованию на JavaScript**.

JavaScript — это язык с множеством подводных камней и на собеседованиях часто задают хитрые вопросы, чтобы проверить глубину твоих знаний.

Эта книга — сборник заметок, которые я пишу на learn.coderslang.com. Я включил только статьи с тегом `js-test`, но их намного больше и тебя могут заинтересовать другие **руководства и статьи о программировании**.

Вот что ты найдешь внутри:

- 40 вопросов с JavaScript собеседований с примерами кода
- Детальное объяснение к каждому вопросу и правильный ответ
- Сюрприз на последней странице :)

Пустое место на страницах было оставлено намеренно. Прежде чем ты посмотришь правильный ответ и объяснение, ты должен потратить хотя бы несколько минут на то, чтобы разобраться самостоятельно и учесть все нестандартные ситуации.

Если ты хочешь **учиться с нуля**, эта электронная книжка не принесет особой пользы. Она написана для тех, кто хочет получить свою первую работу или повышение в IT.

Но не отчаивайся, у меня есть отличный [Full Stack JS курс](#) который ты можешь начать даже с нуля.

Главное — желание и мотивация!

JavaScript собеседование #1: Преобразование типов



```
1 let str = '1';
2 str = +!str;
3 console.log(typeof str);
```

Что будет выведено на экран?

В первой строке, мы объявили переменную `str` и присвоили ей значение `'1'`. Это — строка.

Дальше, происходят два преобразования типов:

- `!str` дает нам `false`, значение типа `boolean`
- потом `+false` превращает `boolean` `0`. А это уже число.

В последней строке, оператор `typeof` проверяет тип значения, которое хранится в переменной `str`. Так как в `str` хранится число `0`, то `typeof` вернет строку `number`.

ОТВЕТ: строка `number` будет напечатана на консоль

JavaScript собеседование #2: Как создать массив



```
1 let a1 = [];
2 let a2 = Array();
3 let a3 = new Array();
```

Какой из перечисленных способов создания массива в JavaScript - правильный?

Первый массив, `a1` объявлен используя литерал пустого массива `[]`. Тут нет никаких проблем и это один из самых распространенных способов создания массива в JavaScript.

Второй, `a2` создается, вызывая `Array` как функцию. Тоже рабочий подход, который создаст пустой массив.

Третий массив, `a3` создается явным вызовом конструктора `Array`, после использования ключевого слова `new`.

ОТВЕТ: Все варианты правильные.

JavaScript собеседование #3: Складываем строки с числами и boolean



```
1 const x = '2' + 3 - true + '1';
2 console.log(x);
```

Получится ли у нас вывести значение `x` на экран? Если да, то что мы увидим?

Чтобы ответить на этот вопрос правильно, ты должен понимать как работает преобразование типов в JS.

Арифметические операции `+` и `-` имеют одинаковый приоритет, поэтому значение `x` будет вычислено слева направо без каких-либо проблем.

Сначала, мы сложим строку `'2'` с числом `3`. В результате получится строка `'23'`.

Дальше, мы попробуем вычесть boolean значение `true` из строки `'23'`. Чтобы у нас что-то получилось, JavaScript сначала преобразует оба значения в числа. Страна `'23'` станет числом `23`, а логическое значение `true` станет числом `1`. Мы вычтем одно из другого и получим результат этого шага — число `22`.

Последний этап — сложение числа `22` и строки `'1'`. Применяя те же правила сложения, что и на первом шаге, мы получим строку `'221'`.

ОТВЕТ: В выражении на первой строке нет ошибок. Значение `x` станет равно `'221'`, и будет выведено на экран.

JavaScript собеседование #4: Какие ошибки поймает try/catch

```
let e1, e2;
try {
    console.log(null.length);
} catch (e) {
    e1 = e;
}
try {
    console.log(undefined.length);
} catch (e) {
    e2 = e;
}

console.log(e1.message.split(' ')[0] === e2.message.split(' ')[0]);
```

Что будет выведено на экран?

У нас есть две переменные и два блока `try/catch`, которые должны поймать ошибки и сохранить их в переменные `e1` и `e2`.

После этого, мы анализируем, сравниваем содержимое сообщений об ошибках и выводим результат на экран.

Первым делом, давай определим, что лежит внутри `e1` и `e2`. Чтобы сделать это, нам нужно разобраться что происходит внутри `try` блоков.

Попытка доступа и к `null.length`, и к `undefined.length` бросит исключение, так как свойства `length` нет ни у `undefined`, ни у `null`.

Эти ошибки будут пойманы в блоках `catch`, как `e` и сохранены в переменные `e1` и `e2`.

Содержимое ошибок будет немного отличаться. Если мы добавим логирование и выведем `e.message` на экран в `catch` блоке, мы получим такие строки:

```
Cannot read property 'length' of null  
Cannot read property 'length' of undefined
```

Дальше, `.split(' ')[0]` даст нам первое слово каждого предложения, которое равно `Cannot` в обоих случаях. Так что в конце концов, вся программа сводится к такому сравнению:

```
console.log('Cannot' === 'Cannot')
```

ОТВЕТ: Выражение внутри `console.log` равно `true`. Это сообщение и будет выведено на экран.

JavaScript собеседование #5: Используем стрелочную функцию как геттер



```
const obj = { id: 1, getId: () => this.id };  
console.log(obj.getId());
```

С функцией `getId` точно все в порядке? Что будет выведено на экран?

Так как `getId` - это стрелочная функция, то у нее нет собственного `this`. В приведенном примере стрелочная функция не будет корректно работать как геттер.

Она не привязана к `this` объекта `obj` и когда мы пытаемся вернуть `this.id`, то на самом деле вернем `undefined`, а не `1`.

ОТВЕТ: в консоль будет выведено сообщение `undefined`.

JavaScript собеседование #6: Переменное количество аргументов

```
● ● ●  
1 const args = [ 1, 2, 3 ];  
2 const arrowFunction = (x, y) => {  
3   return (arguments[2]);  
4 }  
5 const regularFunction = function (x, y) {  
6   return (arguments[2]);  
7 }  
8 console.log(arrowFunction(...args) === regularFunction(...args));
```

Функции в JavaScript могут принимать произвольное количество аргументов. Но что будет выведено на экран? `true` или `false`?

В JS, у всех функций есть доступ к внутреннему массиву `arguments` который хранит все аргументы, переданные в функцию.

Мы можем получить доступ к этим элементам по индексу. Поэтому логично было бы предположить, что обе функции `regularFunction` и `arrowFunction` вернут `true`.

Единственная проблема в том, что стрелочные функции — исключение из правила и у них нет доступа к массиву `arguments`.

В строке 8 может произойти две разные ситуации. Скорее всего ты увидишь ошибку `ReferenceError: arguments is not defined`.

Но может быть и другой сценарий. Например, если запустить этот код в Node.js, то в `arguments[2]` скорее всего будет лежать что-то похожее на такой объект

```
Module {  
  id: '.',  
  path: '/workdir_path',  
  exports: {},  
  parent: null,  
  filename: '/workdir_path/scriptName.js',  
  loaded: false,  
  children: [],  
  paths: [  
    '/node_modules'  
  ]  
}
```

В этом случае ошибки не будет и в консоль выведется сообщение `false`, так как `3` не равняется объекту показанному выше.

ОТВЕТ: сообщения `false` или `ReferenceError` появятся в консоли после выполнения кода.

JavaScript собеседование #7: Какой тип у массива

```
1 const array = ['this', 'is', 'an', 'array'];
2 if (typeof array === 'array') {
3   console.log('ARRAY!')
4 } else {
5   console.log('SOMETHING WEIRD...')
6 }
```

Что вернет вызов `typeof array`? Что будет выведено на экран?

В первой строке мы создаем массив и сохраняем его в константу `array`. После этого, во второй строке, мы пробуем определить тип этого массива с помощью оператора `typeof`.

К сожалению (или к счастью) в JS нет типа `array`, поэтому на экран просто не может вывестись строка `ARRAY!`.

На самом деле, все массивы в JavaScript - это объекты, `typeof array` вернет строку `object` и мы попадем в ветку `else`.

ОТВЕТ: На экран будет напечатано сообщение `SOMETHING WEIRD` потому что у всех массивов в JavaScript тип `object`.

JavaScript собеседование #8: Нулевой таймаут



```
1 setTimeout(() => console.log('timeout log'), 0);
2 console.log('plain log');
```

Какое из сообщений будет выведено первым?

В JavaScript, `setTimeout(func, delay)` берет функцию `func` и откладывает ее выполнение на `delay` миллисекунд.

Может показаться, что так как задержка `0`, то функция должна быть выполнена немедленно, но это не так.

Функция будет помещена в **очередь сообщений** и выполнится асинхронно. Это произойдет только после того, как завершится текущий синхронный код.

Функция `console.log` во второй строке является частью синхронного кода, поэтому она выполнится раньше, чем `console.log` из таймаута в первой строке.

В большинстве браузеров у `setTimeout(f, 0)` есть задержка около `3 ms` которая определяется скоростью внутренней обработки JavaScript кода.

ОТВЕТ: Сначала будет напечатано сообщение `plain log`, а потом - `timeout log`.

JavaScript собеседование #9: Promise.reject внутри блока try/catch

```
1 try {
2     Promise.reject('an error occurred');
3 } catch (e) {
4     console.log('the error was caught!');
5 }
```

Поймаем ли мы ошибку в блоке `catch`?

Обычные блоки `try/catch` в JavaScript ловят только ошибки в синхронном коде.

Так как у промиса во второй строке нет собственного блока `.catch`, то ошибка, которую вызовет `reject` останется необработанной.

Возникнет исключение `UnhandledPromiseRejectionWarning` и код внутри блока `catch` не будет выполнен, потому что в синхронном коде ошибок нет.

ОТВЕТ: Ошибка не будет поймана и сообщение `the error was caught!` НЕ будет выведено в консоль.

JavaScript собеседование #10: null + undefined



```
1 console.log(null === null);
2 console.log(undefined === undefined);
3 console.log(null + undefined === null + undefined);
```

В чем разница между `null` и `undefined`? Что будет выведено на экран?

В первой строке, мы проверяем равенство `null === null` и получаем результат `true`.

Во второй строке, мы проверяем равенство `undefined === undefined` и снова получаем `true`.

А для того, чтобы понять результат третьей строки, нам нужно посчитать чему равна сумма `null + undefined`. Для движка JavaScript, сложно "понять" чему должно быть равно такое выражение, поэтому он его считает как `NaN`.

Итак, с двух сторон `==` у нас непонятный `not-a-number`. И вопрос. Равен ли `NaN` сам себе?

И ответ - **НЕТ**.

В JavaScript `NaN` - это единственное значение не равное самому себе.

ОТВЕТ: На экран будут выведено `true`, `true` и `false`.

JavaScript собеседование #11: Область видимости

● ● ●

```
1 const animals = [ 'Cow', 'Horse', 'Dog', 'Cat', 'Rabbit' ];
2
3 for (let i = 0; i < animals.length; i++) {
4     const animals = [ 'Whale', 'Dolphin' ];
5     console.log(animals[i]);
6 }
```

В JavaScript у переменных есть несколько областей видимости. Что будет выведено на экран?

В первой строке мы видим создание массива `animals`, в котором хранится 5 строк.

Длина этого массива используется в условии цикла `for`, поэтому цикл будет работать до тех пор, пока счетчик `i` не станет равен 5.

Внутри цикла создается еще один массив `animals`. С таким объявлением массива проблем нет, так как его область видимости распространяется только на блок кода внутри цикла `for`.

Важно помнить, что хотя мы и объявили новый массив `animals` с тремя элементами, `animals.length` в условии цикла до сих пор связана со внешним массивом, и не изменит свое значение.

А `console.log` внутри цикла будет работать только со внутренним массивом, в котором всего два элемента.

Как только мы выйдем за границы массива, то не получим ошибку как в `C++` или `Java`. Вместо этого, на экран будет выведено сообщение `undefined` на трех последних итерациях цикла.

ОТВЕТ: В консоль будут напечатаны строки `Whale`, `Dolphin`, а после них `undefined`, `undefined`, `undefined`. Каждое значение будет выведено с новой строки.

JavaScript собеседование #12: Math.min()



```
const x = Math.min();  
const y = 0;  
  
console.log(x > y);  
  
// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Насколько маленькое значение у `Math.min()` в JavaScript?

Функция `Math.min()` принимает произвольное количество аргументов и возвращает самый маленький из них.

В нашем случае, она вызвана без аргументов и это нестандартная ситуация.

Если в JavaScript вызвать функцию `Math.min()` без параметров, то она вернет значение `Infinity`.

ОТВЕТ: в консоль будет выведено значение `true`, потому что `Infinity` больше чем `0`.

JavaScript собеседование #13: Большие числа



Мы же просто выводим число, что может пойти не так?

Под капотом, в JavaScript нет целых чисел.

Все они представлены как 64-битные числа с плавающей точкой.

Такой формат еще называют double precision.

Первые 52 бита используются, чтобы хранить двоичные цифры, 11 битов хранят позицию плавающей точки и 1 бит отвечает за знак и определяет положительное число или отрицательное.

Когда "не хватает места" чтобы сохранить большое целое число, происходит округление до другого ближайшего целого, которое полностью помещается в выделенные 64 бита.

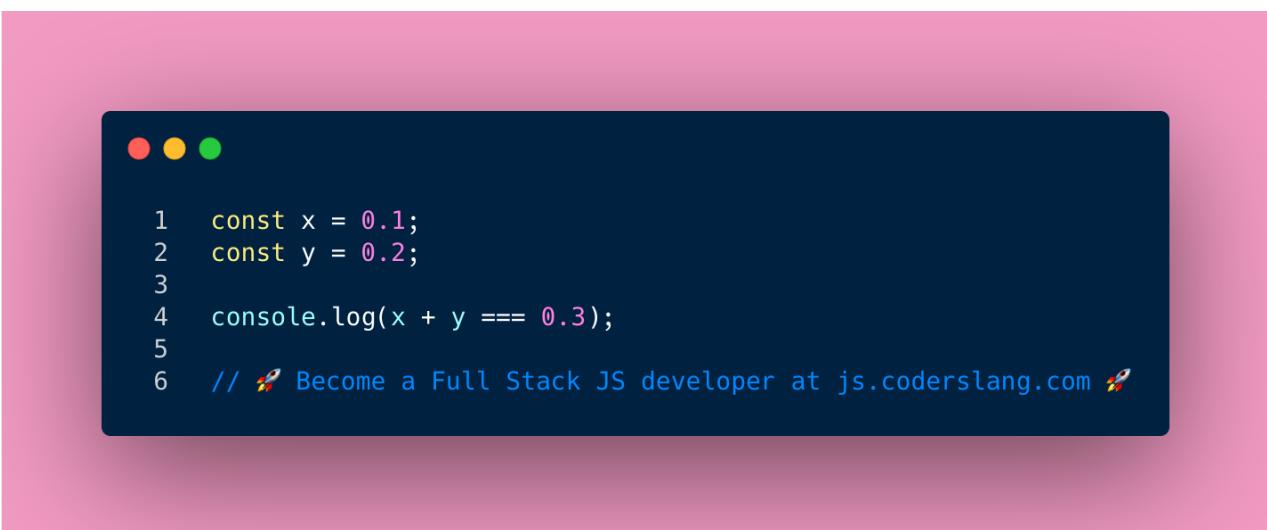
Значащие биты числа 9999999999999999 невозможно сохранить используя только 52 бита, поэтому при округлении мы избавляемся от самого младшего бита, сдвигаем точку и получаем число 1000000000000000.

В JavaScript, такая ситуация считается нормальной и не приводит к ошибке.

Если не совсем понимаешь, что происходит тут, советую изучить лекцию о [двоичной системе](#) курса CoderslangJS.

ОТВЕТ: на экран будет выведено число 1000000000000000

JavaScript собеседование #14: Сколько будет $0.1 + 0.2 = ?$



Хорошо знаешь JavaScript математику? Что будет выведено на экран?

Внутри компьютера все числа представлены в **двоичной системе**.

То есть, там нет привычным нам десятичных чисел, а есть только последовательности **битов**. Бит — это двоичная цифра, он может принять только два значения - **0** или **1**.

Число **0.1** это то же, что и **1/10** и его легко представить как десятичную дробь. А в двоичной системе, "одна десятая" превратится в бесконечную дробь. Почти так же как и **1/3** в десятичной системе.

Все числа в JavaScript хранятся как **64-битные** значения с плавающей точкой. Когда число не помещается в эти 64 бита, оно автоматически округляется.

Это приводит нас к тому, что в JavaScript **0.1 + 0.2** будет равно **0.3000000000000004**, а не **0.3**, как ты мог бы подумать.

ОТВЕТ: на экран будет выведено значение **false**.

JavaScript собеседование #15: Отличия между функциями геттерами



```
const obj = {
  id: 1,
  getIdArrow: () => this.id,
  getIdFunction: function () {
    return this.id;
  }
};

console.log(obj.id);
console.log(obj.getIdArrow() === obj.getIdFunction());
```

В чем отличие между обычной функцией и стрелочной функцией в JavaScript? Что будет выведено на экран?

У нас есть объект `obj` с одним полем `id`, которое равно `1`. Две функции `getIdArrow` и `getIdFunction` которые в теории должны делать одно и то же — вернуть значение поля `id`.

Но, к сожалению, это не так. В JavaScript, стрелочные функции отличаются от обычных. У них нет связи с `this` объекта `obj`. Поэтому внутри стрелочной функции `this.id` будет равно `undefined`.

В случае функции `getIdFunction`, `this` привязано к `obj` и `this.id` — это то же что и `obj.id`, то есть `1`.

ОТВЕТ: первый `console.log` выведет на консоль число `1`. Второй — сообщение `false`.

JavaScript собеседование #16: typeof NaN

```
1 console.log(typeof NaN);
2
3 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

В JS есть специальное значение - `Not a Number`. Чему равен его тип?

В JavaScript, `Nan` значит `Not a Number`. Это особое значение, которое возникает в ситуациях когда JavaScript не может адекватно посчитать числовое выражение.

Еще, значение `Nan` может возникнуть во время преобразования типов. Например, если ты попробуешь сконвертировать строку в число, то получишь `Nan`.

Это не совсем очевидно, но `Nan` - это особое число. Поэтому его типом считается `number`.

Проблему можно попробовать решить от обратного:

Чем еще мог бы быть тип `Nan`?

ОТВЕТ: выражение `typeof Nan` вернет строку `number`, которая и будет выведена на экран.

JavaScript собеседование #17: Сумма двух пустых массивов в JS

```
1  if ([] + [] == false) {
2    console.log('same');
3  } else {
4    console.log('different');
5  }
6
7 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Равна ли сумма двух пустых массивов `false`?

Чтобы проанализировать этот тест, нам нужно понимать как работает преобразование типов в JavaScript.

Когда мы пытаемся сложить два массива используя оператор `+`, массивы сначала становятся строками, а потом складываются.

Пусто массив `[]` становится пустой строкой после первого преобразования. А сумма двух пустых строк — тоже пустая строка.

И, мы приходим к финальному вопросу:

Равна ли в JavaScript пустая строка `false` или нет?

В тесте для сравнения используется оператор `==`. Этот оператор выполняет нестрогое сравнение. Это значит, что под капотом JavaScript сначала выполнит приведение типов operandов, а потом — сравнение.

В нашем случае пустая строка и `false` считаются равными, потому что оба будут приведены к числу `0` и условие внутри `if` будет равно `true`.

Если тебе нужно выполнить строгое сравнение, которое сначала проверит совпадение типов operandов в JavaScript, то используй оператор `====`.

Тут, ты можешь найти больше информации о [базовой математике JavaScript](#).

ОТВЕТ: на экран будет выведена строка `same`.

JavaScript собеседование #18: Чему равна сумма двух boolean?

```
1  if (true + true == true) {  
2      console.log('there is only one truth');  
3  } else {  
4      console.log('everyone is different, after all')  
5  }  
6  
7 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Неужели тут что-то может быть неправильно? Что будет выведено на экран?

Так же как и в [предыдущем тесте](#), мы имеем дело с преобразованием типов и нестрогим сравнением используя оператор `==`.

Когда JavaScript вычисляет выражение `true + true`, он сначала преобразовывает оба значение в числа, и только после этого складывает их. Для переменных логического типа правила такие: `true` становится единицей, а `false` - нулем.

Когда мы пытаемся выполнить сравнение `2 == true`, то приведение типов в JavaScript возникает снова и мы получаем выражение `2 == 1`.

Ответ, очевидно, `false` и мы идем в ветку `else`.

Чтобы понять, как приведение типов работает с оператором `+` и разными типами данных, советую тебе прочесть [этую заметку](#).

ОТВЕТ: на экран будет выведена строка `everyone is different after all.`

JavaScript собеседование #19: Ловим ошибку в JS промисе



```
1  try {
2    Promise.reject(null.length).then(console.log);
3  } catch (e) {
4    console.log('the error was caught!', e.message);
5 }
```

Еще одна необработанная ошибка? Поймаем или нет?

В JS, невозможно поймать асинхронную ошибку используя обычный блок `try/catch`.

Поэтому, если в асинхронном коде возникает исключение, мы видим `UnhandledPromiseRejectionWarning ...` или что-то похожее.

Но в этом тесте мы даже не успеем дойти до асинхронного исключения.

JavaScript попытается получить доступ к `null.length` в синхронном режиме. Так как у `null` нет поля `length`, как нет и других полей, то мы получим синхронную ошибку `Cannot read property 'length' of null`.

Эту ошибку отлично поймает блок `catch` и выведет на экран сообщение о том, что ошибка была поймана.

ОТВЕТ: ошибка будет поймана и в консоли появится сообщение `the error was caught! Cannot read property 'length' of null.`

JavaScript собеседование #20: Можно ли складывать объекты с массивами в JS?

```
1 const res = [] + {};
2
3 if (res.length > 10) {
4   console.log('wow, this is quite long');
5 } else {
6   console.log('it\'s f**ing empty');
7 }
8
9 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Пустой массив плюс пустой объект. Найдется ли у них поле `length`?

Что будет в консоли?

И снова мы видим пример приведения типов в JavaScript.

И пустой массив, и пустой объект будут преобразованы в строки.

В случае пустого массива — это будет пустая строка.

Но для пустого объекта мы получим строку `[object Object]`!

Так как длина этой строки больше 10, то на экран будет выведено сообщение из основной ветки `if`.

Больше примеров преобразования типов в JavaScript можешь найти [тут](#).

ОТВЕТ: на экран будет выведена строка `wow, this is quite long.`

JavaScript собеседование #21: ISO Date

```
 1 const date = new Date();
 2
 3 console.log(date.toISOString().slice(0, 4));
 4
 5 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Как выглядит ISO Date в JavaScript? Что будет выведено на экран?

В первой строке мы создаем новый объект `Date`.

Он хранит текущую дату и время.

Функция `toISOString` возвращает строковое представление объекта `date` в формате `ISO`. В этом формате дата всегда начинается с года и заканчивается временем. Что-то вроде `2020-12-27T10:35:26.159Z`.

Когда мы применяем `slice(0, 4)` к этой строке, то получим первые четыре символа, в которых хранится текущий год.

ОТВЕТ: на экран будет выведен текущий год.

JavaScript собеседование #22: Как работает `toString` в JS?

```
const toString = Object.prototype.toString;
const arr = [ 1, 2, 3 ];

console.log(toString.call(arr));

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Что будет выведено на экран?

В первой строке, мы сохранили функцию

`Object.prototype.toString` в константу `toString`. Эта функция вызывается тогда, когда нужно преобразовать объект в строку.

У большинства объектов, функция `toString` переопределена.

Например, у массивов, она выведена на экран список всех значений массива через запятую.

Обычное поведение функции `Object.prototype.toString` в браузере — это строка вида `[object "TYPE"]`. Часть "TYPE" заменяется на тип объекта, на котором вызывается `toString`. В нашем случае — это `Array`.

Итого, при вызове `toString.call(arr)` мы обратимся к оригинальной реализации `Object.prototype.toString` и поэтому в консоли не появится перечисление всех элементов массива.

ОТВЕТ: на экран будет выведена строка `[object Array]`.

JavaScript собеседование #23: Как работает Array.splice в JS

```
const arr = [1, 2, 3, 4, 5];
const splicedArr = arr.splice(1, 2);

arr.splice(1, 2, ...splicedArr);
console.log(arr);

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Сколько раз можно сделать `splice` с массивом?

Давай начнем с определения `splice` в JavaScript.

Функция `splice` доступна во всех JavaScript массивах и принимает переменное количество аргументов. Вот 4 важные вещи, которые тебе нужно знать о функции `splice`:

- Первый параметр называется `start` и определяет индекс первого элемента, который будет удален из массива.
- Второй параметр - `deleteCount`. Он определяет количество элементов, которые будут удалены.
- Третий, четвертый и все следующие элементы — это новые значения, которые будут добавлены в массив
- Функция `splice` возвращает массив сформированный из удаленных элементов начального массива

Теперь перейдем к нашему тесту.

Мы начинаем с массивом `arr`. В нем пять элементов - `[1, 2, 3, 4, 5]`.

Первый `splice` удалить два элемента, начиная с `arr[1]`. Мы сразу же сохраняем их в `splicedArr`.

Перед вторым сплайсом у нас такое состояние:

```
[ 1, 4, 5 ] // arr
[ 2, 3 ]     // splicedArr
```

Второй вызов `splice` снова удалит 2 elements из `arr` начиная от `arr[1]`. И мы остаемся с одним элементом — `[1]`.

Дальше мы применяем деструктуризация `...` к `splicedArr` и добавляем новые значения `2` и `3` к финальному состоянию `arr`.

Вот пример кода с двумя дополнительными вызовами `console.log`, который ты можешь запустить, чтобы лучше понять объяснение:

```
const arr = [1, 2, 3, 4, 5];
const splicedArr = arr.splice(1, 2);

console.log(arr);          // [ 1, 4, 5 ]
console.log(splicedArr);  // [ 2, 3 ]

arr.splice(1, 2, ...splicedArr);
console.log(arr);
```

ОТВЕТ: массив `arr` будет содержать значения `[1, 2, 3]`. Они и будут выведены на экран.

JavaScript собеседование #24: Добавляем поле обычной JS строке



```
const s = 'Hello world!'
s.user = 'Jack';

console.log(s.user);

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Как думаешь, получится у нас добавить поле `user` строке `s` или получим ошибку? Если нет, то что выведется на экран?

Ответ на этот вопрос зависит добавлен ли флаг `'use strict'` твоему JS скрипту.

Возможные варианты:

- `undefined` если `'use strict'` не был указан в скрипте
- ошибка, если флаг был добавлен и выполнение происходит в строгом режиме

Почему же это происходит?

Во второй строке, когда ты пытаешься получить доступ к `s.user`, JavaScript под капотом создает объект обертку типа `String`.

И если ты используешь строгий режим, то попытка модификации такого объекта бросит ошибку.

Но без флага `use strict` ошибки не будет и поле `user` без проблем добавится к объекту обертке.

Но, несмотря на то, что при добавлении ошибки уже нет, на следующей строке нет и объекта обертки. Свойство `user` пропало и поэтому на экран выведется строка `undefined`.

ОТВЕТ: В JS нельзя добавлять новые поля примитивным значениям, таким как строки или числа. Результат будет зависеть от того, используется флаг `'use strict'` или нет.

JavaScript собеседование #25: Мгновенный Promise.resolve

```
1  Promise.resolve().then(() => {
2    console.log('resolved');
3  );
4  console.log('end');
```

Как быстро работает `Promise.resolve()`? Какое сообщение будет выведено первым?

В этом teste логика почти та же, что и в [примере с setTimeout](#).

Даже хотя у `Promise.resolve()` нет явной задержки, но код внутри `.then()` выполняется асинхронно, и у него ниже приоритет, чем у синхронного кода.

Поэтому, функция `console.log('resolved')` будет выполнена после `console.log('end')`.

ОТВЕТ: сначала на консоль выведется строка `end`, а после нее - `resolved`.

JavaScript собеседование #26: Равны ли эти даты?

```
 1 const date1 = new Date();
 2 const date2 = new Date(0);
 3
 4 if (date1 === date2) {
 5   console.log('equal');
 6 } else {
 7   console.log('not so much');
 8 }
 9 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

В JavaScript дату можно создать несколькими способами. Но равны ли эти две?

В первых двух строках когда создаются два объекта `Date`.

А на четвертой строке, мы сравниваем их используя оператор строгого равенства `==`. Помни, мы сравниваем разные объекты!

Даже если `date` и `date2` представляли бы одну и ту же дату и время, их строгое равенство всегда вернет `false`.

ОТВЕТ: В консоль будет выведена строка `not so much`, потому что `date1` и `date2` - это разные объекты.

JavaScript собеседование #27: Обработка ошибок внутри JS промисов

```
● ● ●

const f1 = (promise, successHandler, errorHandler) => {
    return promise.then(successHandler, errorHandler);
}

const f2 = (promise, successHandler, errorHandler) => {
    return promise.then(successHandler).catch(errorHandler);
}

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Есть ли какая-то разница между функциями `f1` и `f2`?

В JavaScript обработчик ошибок для промисов можно добавить двумя основными способами.

Первый показан в функции `f1`. Мы передаем обработчик `errorHandler` как второй аргумент для `.then()`.

Второй подход реализован в функции `f2`. Тут, мы добавляем `errorHandler` используя функцию `.catch()`.

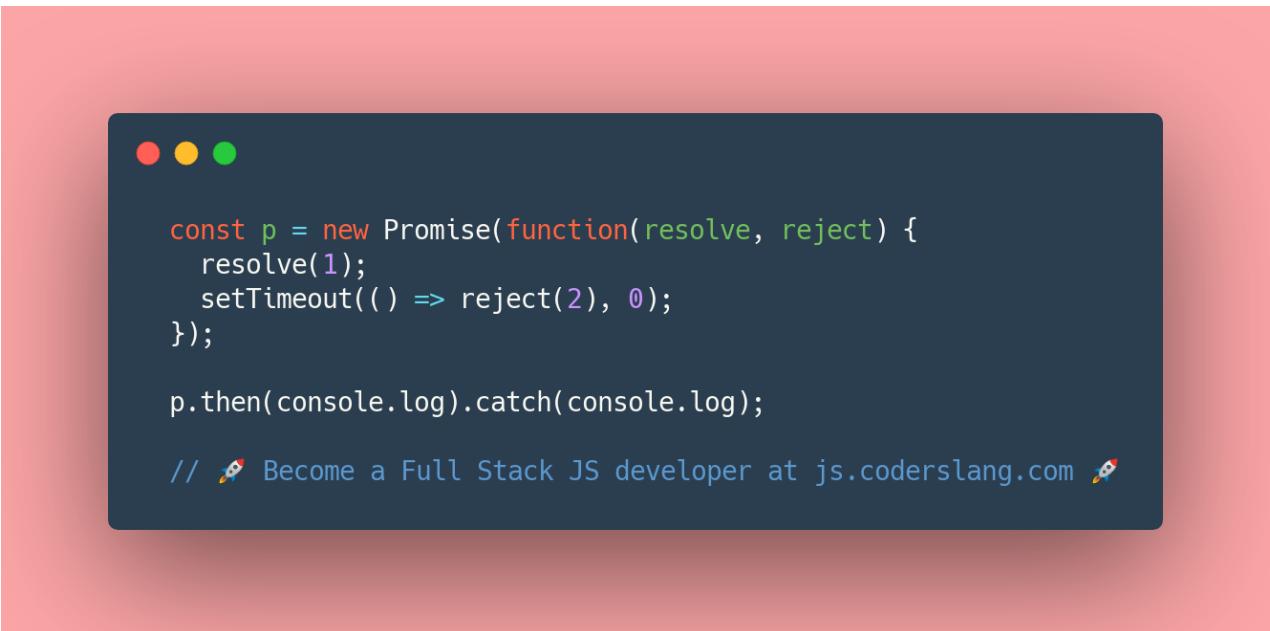
В обоих случаях обработчик `errorHandler` будет вызван если ошибка произойдет внутри `promise`.

Если же `promise` завершится успешно, то выполнение продолжится внутри `successHandler`. И если `successHandler` бросит ошибку, то она будет обработана только в `f2`, но не в `f1`.

Это происходит из-за того, как реализован блок `.catch()` для JavaScript промисов. Он обрабатывает все ошибки в цепочке промисов, даже те, которые возникли внутри функций переданных в `.then()`.

ОТВЕТ: Да, есть большая разница между `f1` и `f2`. Первая функция не обработает ошибку внутри `successHandler` (если она возникнет), а вторая справится с ошибкой без проблем.

JavaScript собеседование #28: Одновременный `resolve` и `reject`



```
const p = new Promise(function(resolve, reject) {
    resolve(1);
    setTimeout(() => reject(2), 0);
});

p.then(console.log).catch(console.log);

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Можно ли сделать `resolve` и `reject` для одного и того же JavaScript промиса? Что будет напечатано в консоль?

В JavaScript, промисы не могут быть одновременно переведены завершиться и успешно и неуспешно. А именно это подразумевают вызовы `resolve()` и `reject()`.

Выполнение никогда не дойдет до `setTimeout` и команды `reject(2)` внутри него.

Следовательно, только число `1` будет выведено в консоль.

ОТВЕТ: Одна строка будет выведена в консоль. После того, как промис успешно завершается со значением `1`, выполнение прекращается и `setTimeout` не будет вызван.

JavaScript собеседование #29: Рубим и режем



```
const arr = [1, 2, 3, 4, 5];
const slicedArr = arr.slice(1, 2);

arr.splice(1, 2, ...slicedArr);
console.log(arr);

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Какая разница между `slice` и `splice` в JavaScript? Что произойдет с массивом `arr`?

`Array.slice` в JavaScript возвращает поверхностную копию массива. Первыми двумя параметрами ты должен передать индексы `start` и `end`. Элемент массива на позиции `arr[start]` включается в результат, а элемент `arr[end]` - нет. Также, `Array.splice` удалит все элементы между `start` и `end` из первоначального массива.

В отличие от `Array.splice`, массив не будет модифицирован после вызова `Array.slice`.

Таким образом, после выполнения первых двух строк, у нас будет такое состояние:

```
[ 1, 2, 3, 4, 5]      // arr
[ 2 ]                  // slicedArr
```

Потом, мы делаем два действия с помощью `arr.splice`:

- удаляем два элемента из массива `arr` начиная с `arr[1]`. В массиве остаются элементы `[1, 4, 5]` после этого действия.
- мы делаем деструктуризацию `...slicedArr` и добавляем его элементы в `arr` начиная с `arr[1]`. Так, мы приходим к финальному состоянию `[1, 2, 4, 5]` внутри массива `arr`.

Пример кода, с дополнительным выводом на экран для самопроверки:

```
const arr = [1, 2, 3, 4, 5];
const slicedArr = arr.slice(1, 2);

console.log(arr);          // [ 1, 2, 3, 4, 5]
console.log(slicedArr);   // [ 2 ]

arr.splice(1, 2, ...slicedArr);
console.log(arr);          // [ 1, 2, 4, 5]
```

ОТВЕТ: Массив `arr` будет изменен и будет содержать элементы `[
1, 2, 4, 5]`.

JavaScript собеседование #30: reject промиса внутри resolve

```
● ○ ●
```

```
1  try {
2    Promise.resolve(Promise.reject(-1)).then(console.log);
3  } catch (e) {
4    console.log('the error was caught!', e.message);
5  } finally {
6    console.log('finally');
7 }
```

Что произойдет, если ты попробуешь сделать `reject` JavaScript промиса внутри `resolve`? Будет ли выполнен блок `finally`?

Чтобы проанализировать эту проблему, я начну с вещей, которые можно сказать наверняка:

- логирование внутри функции `.then(console.log)` не будет выполнено, потому что `Promise.resolve()` завершится с ошибкой
- блок `catch` не ловит ошибки в асинхронном коде

Итак, у нас остался блок `finally`. В нем есть один вызов `console.log` и это первая строка, которая будет выведена на экран.

После этого, возникнет необработанная ошибка в асинхронном коде, потому что мы не добавили обработчик ошибок к промису в строке 2.

ОТВЕТ: На экран будет сначала выведена строка `finally`, а после нее — сообщение об асинхронной ошибке

`UnhandledPromiseRejectionWarning: -1.`

JavaScript собеседование #31: Больше или меньше

```
 1  if (Math.max() > 0) {
 2    console.log('MAX ALWAYS WINS!');
 3  } else {
 4    console.log('ZERO!')
 5  }
 6
 7 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Что больше, `Math.max()` или `0`? Что будет выведено на экран?

Итак, у нас есть условный оператор `if` и условие внутри него. Получается, что нам нужно проанализировать только выражение `Math.max() > 0`, чтобы понять, каким будет ответ.

Если твоей первой догадкой было то, что `Math.max()` должно вернуть какое-то очень большое число, которое будет точно больше чем `0`, то ты ошибаешься.

В JavaScript `Math.max()` принимает несколько аргументов и возвращает самый большой из них. Сравнение начинается с самого низа, от `-Infinity`, потому, что меньше число не существует.

Поэтому, если вызвать `Math.max()` без аргументов, то вернется `-Infinity`.

А так как `-Infinity` меньше чем `0`, мы попадаем в ветку `else` условного оператора.

ОТВЕТ: на экран будет выведена строка `ZERO!`.

JavaScript собеседование #32: 0.1 + 0.1 + 0.1 === 0.3

```
1 const x = 0.1;
2 const y = 0.1;
3 const z = 0.1;
4
5 console.log(x + y + z === 0.3);
6
7 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Как у тебя с математикой в JavaScript? Знаешь, что будет выведено на экран?

На первый взгляд, кажется что ответ должен быть `true`, потому что `0.1 + 0.1 + 0.1` очевидно равняется `0.3`.

Но это правильно только до тех пор, пока мы не разобрались как числа представлены в JavaScript.

Если ты попробуешь выполнить команду `console.log(0.1 + 0.2)` в JS, то получишь число `0.3000000000000004`.

Это происходит потому, что в JavaScript, как и во многих других языках программирования, некоторые десятичные числа не могут быть представлены точно.

Например, `0.1` в двоичной системе счисления будет представлена как бесконечная дробь. Похожая на то, как `1/3` становится `0.333(3)` в десятичной системе.

ОТВЕТ: `false` будет выведено на экран.

JavaScript собеседование #33: Складываем два пустых массива и проверяем тип

```
1  console.log(typeof ([] + []));
2
3  // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Чему равен тип суммы двух пустых массивов? Массив? Объект? Undefined? Что будет выведено на экран?

В JavaScript, оператор `+` не выполняет конкатенацию (склеивание) массивов.

Вместо этого, он преобразовывает массивы в строки и склеивает их.

Два пустых массива становятся двумя пустыми строками, а их сумма — тоже пустая строка!

Что важно для нас, так это результат, который вернет оператор `typeof`. Для любой строки, даже пустой, это будет - `string`.

ОТВЕТ: на экран будет выведена строка `string`.

JavaScript собеседование #34: Разные способы получить текущую дату в JS



```
1 console.log(new Date() == Date.now());
2
3 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Как ты сохраняешь текущую дату в JS? Есть ли разница между `new Date()` и `Date.now()`? Чем отличаются эти методы?

Как `new Date()`, так и `Date.now()` в JavaScript вернут текущую дату и время.

Разница между ними в том, что `new Date()` вернет объект `Date`, а `Date.now()` - количество миллисекунд, прошедшее после первого января 1970.

Если тебе нужно сравнить две даты, сохраненные в разных форматах, то ты всегда можешь перевести объект `Date` в миллисекунды используя встроенную функцию `getTime()`.

Иначе, у тебя не получится адекватно сравнить число с объектом.

ОТВЕТ: на экран будет выведено `false`.

JavaScript собеседование #35: Что работает быстрее, нулевой таймаут или мгновенный resolve?

```
1  setTimeout(() => console.log(1), 0);
2  Promise.resolve(2).then(console.log);
3
4  // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

В каком порядке сообщения будут выведены в консоль?

И `setTimeout`, и `Promise.resolve` позволяют выполнить асинхронный код. Поэтому вызовы `console.log` произойдут после минимальной задержки.

Разница между ними в том, что `Promise.resolve` планирует микро-задачу, а `setTimeout` - макро-задачу.

У микро-задач приоритет выше, поэтому `Promise.resolve` сработает быстрее и первым сообщением на экране мы увидим `2`.

ОТВЕТ: Сначала будет выведена цифра `2`, а после нее - `1`.

JavaScript собеседование #36: Как добавить новое свойство JavaScript массиву.



```
const arr = [1, 2, 3, 4];
arr.greeting = 'Hello, world!';

console.log(arr.length);
console.log(arr.greeting);
```

// 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀

Получится ли у нас добавить поле `greeting` массиву `arr`?

Изменится ли длина массива? Что будет выведено на экран?

Массивы в JavaScript используются, чтобы хранить упорядоченные данные. Обычно, чтобы добавить новый элемент, используют функцию `push`.

Но, в JS, все массивы — это в первую очередь, объекты. Значит, мы можем добавлять им новые поля!

Присваивание `arr.greeting = 'Hello, world!'` не вызовет никаких ошибок и будет обработано корректно.

Как только мы его выполним, в массиве появится новое поле `greeting`.

Длина массива не изменится, потому что `Hello, world!` не считается одним из элементов массива.

ОТВЕТ: длина массива не изменится и останется равной 4. Мы убедимся в этом после первого `console.log`. Второй `console.log` выведет на экран строку `Hello, world!`.

JavaScript собеседование #37: Чему равен тип аргумента функции `split`?

```
● ● ●

1  function split(...args) {
2    console.log(typeof args);
3  }
4
5  split('hello');
6
7 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Что будет выведено на экран?

В этом примере кода, все аргументы, которые отправятся в функцию `split` будут собраны в массиве `args`.

То, что мы предоставляем в качестве аргумента только строку `hello`, никак не меняет ожидаемое поведение.

Все массивы в JavaScript - это объекты. Поэтому `typeof args` вернет строку `object`. Она и будет выведена в консоль.

ОТВЕТ: на экране появится строка `object`.

JavaScript собеседование #38: Складываем JS массивы

```
1  function add(x, y, z) {  
2      return x + y + z;  
3  }  
4  
5  const sum = add([ 1, 2 ], [ 3, 4 ], [ 5, 6 ]);  
6  
7  console.log(sum);  
8  
9  // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Можно ли складывать массивы в JavaScript с помощью `+`? Что будет выведено на экран?

Функция `add(x, y, z)` применяет оператор `+` к параметрам `x`, `y` и `z`. Или, другими словами, складывает их.

В пятой строке, мы передаем в `add` три массива.

В JavaScript, всегда, когда ты попробуешь сложить несколько массивов с помощью `+`, они будут сначала конвертированы в строки. Элементы в массивах будут разделены запятой и пробелом.

В нашем случае, три массива станут такими тремя строками:

- `1, 2`
- `3, 4`
- `5, 6`

Потом произойдет конкатенация (склеивание) строк и мы придем к нашему финальному результату.

ОТВЕТ: на экран будет выведена строка `1, 23, 45, 6`.

JavaScript собеседование #39: Как работает `setTimeout` внутри цикла `for`?



```
1  for (var i = 0; i < 5; i++) {  
2      setTimeout(() => console.log(i), 0);  
3  }  
4  
5 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Что будет выведено на консоль?

Прежде чем мы начнем анализировать пример кода, давай попробуем его упростить. Избавимся на минутку от `setTimeout`.

Если мы просто оставим `console.log` внутри цикла `for`, то он выведет на экран все значения `i` от `0` до `4`. На каждой итерации цикла на экране будет появляться новая цифра.

Но, когда мы добавляем `setTimeout`, то все меняется. Вызов функции `console.log` произойдет уже после того, как цикл завершится. Значение `i` будет равно `5`.

А так как `i` объявлена с помощью ключевого слова `var`, у нее глобальная область видимости. В отличие от `let`, не получится сохранить отдельные значения для `i` для каждой итерации цикла внутри замыкания `() => console.log(i)`.

ОТВЕТ: цифра `5` будет выведена на консоль 5 раз. Каждый раз с новой строки.

JavaScript собеседование #40: Чему равен тип `undefined` в JS?

```
● ● ●
```

```
1 console.log(typeof undefined === undefined)
2
3 // 🚀 Become a Full Stack JS developer at js.coderslang.com 🚀
```

Что будет выведено на экран?

В JavaScript, оператор `typeof` всегда возвращает строку.

Поэтому, хотя и `typeof undefined` возвращает `'undefined'` - это уже строка, а не примитивное значение `undefined`.

Строка `'undefined'` не равна `undefined`.

ОТВЕТ: в консоли появится сообщение `false`

Вместо заключения

Спасибо! Я уверен, что тебе удалось выучить несколько новых JS трюков, которые помогут тебе на следующем собеседовании.

Если какие-то из примеров оказались слишком сложными, заходи на learn.coderslang.com. Там я пишу статьи по JavaScript для начинающих, чтобы **улучшить твои знания**.

Вот несколько вещей, которые можно сделать **дальше**:

- скачать приложение Coderslang для [iOS или Android](#) чтобы **подготовиться к собеседованию** не только по JS, но и Java, Node, React, HTML, CSS, QA, и C#
- отправить эту книжку друзьям, рассказать о ней в своем блоге или соц. сетях
- получить больше тестов и вопросов на собеседовании по JavaScript [тут](#)

Я очень ценю время, которое ты уделяешь обучению, поэтому держи код на **скидку 15%** на мой курс [Full Stack JS](#).

```
const DISCOUNT_CODE = 'surprise-js';
```

Если у тебя есть вопросы, можешь найти меня на [Twitter](#), [Telegram](#) или просто отправить письмо на welcome@coderslang.com