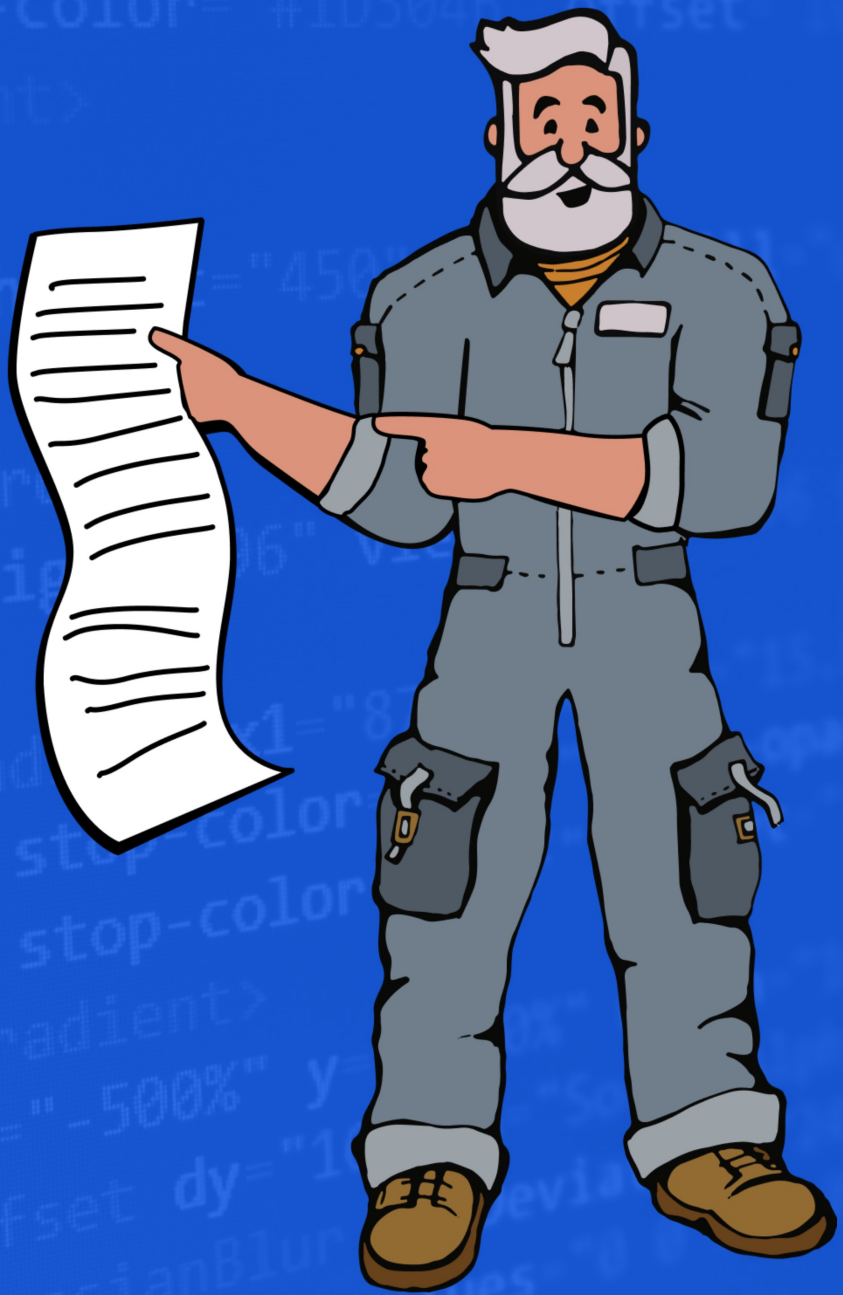
# JavaScript Interview #35



Coderslang Master

# Introduction

Hey there! **Thank you** for downloading the e-book. It's designed to help you **prepare for a technical interview** in **JavaScript**.

JavaScript is tricky and interviewers love to post questions that test the depth of your knowledge.

This e-book is a compilation of the short articles that I write at learn.coderslang.com. I have included only the ones tagged `js-test`, but there's a lot more, so you might want to explore it further for more **programming tutorials**.

Here's what you'll find in it:

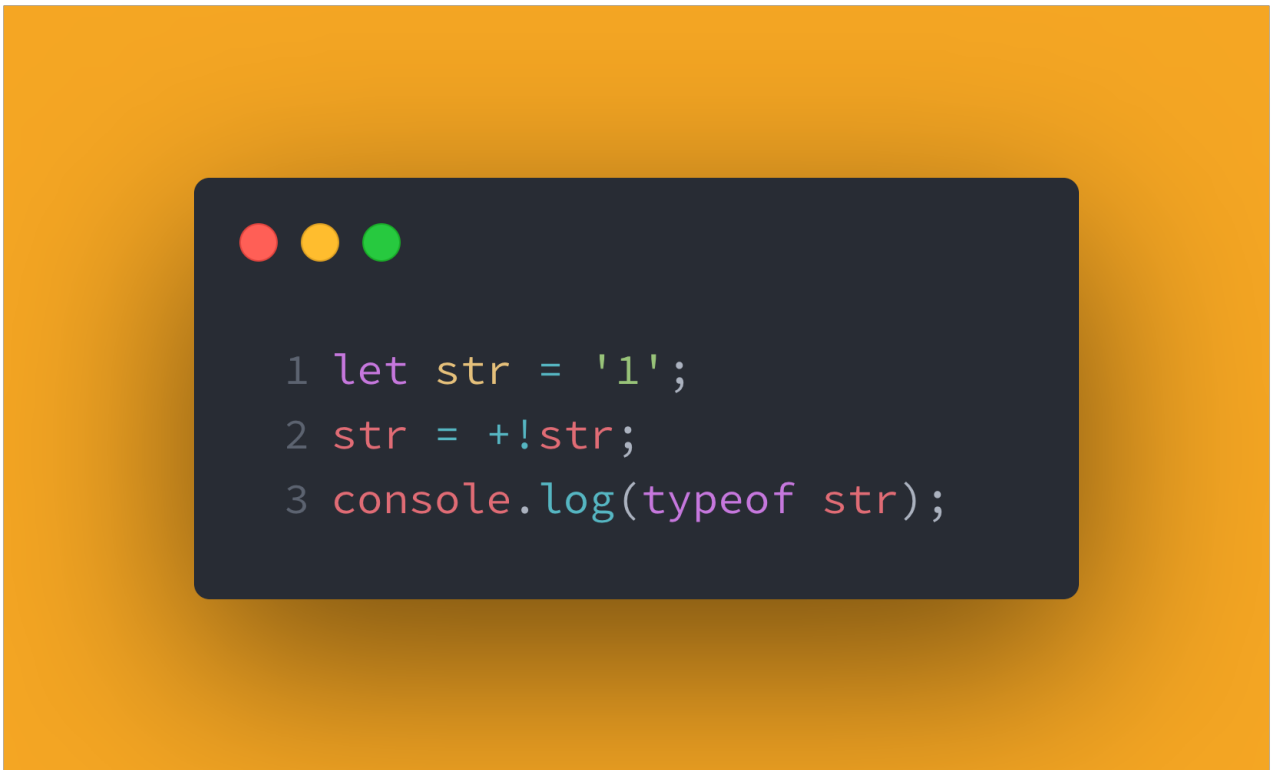
- 35 colorful JS code snippets
- Detailed explanation to every problem
- A surprise on the last page :)

The blank space on the pages with code snippets was left there itentionally. Before looking up the correct answer and an explanation, you should spend a couple of minutes thinking about all the corner cases.

If you're looking to **learn from scratch**, this e-book won't be very helpful. It's written specifically for those who are struggling to land their first or maybe a second job in IT.

But don't despair, as I have a Full Stack JS course that you can start even without any prior knowledge.

# JS Test #1: Type conversion in JavaScript




What's going to be printed to the console?

In the first line, we define the variable `str` and initialize it as a string with the value `1`.

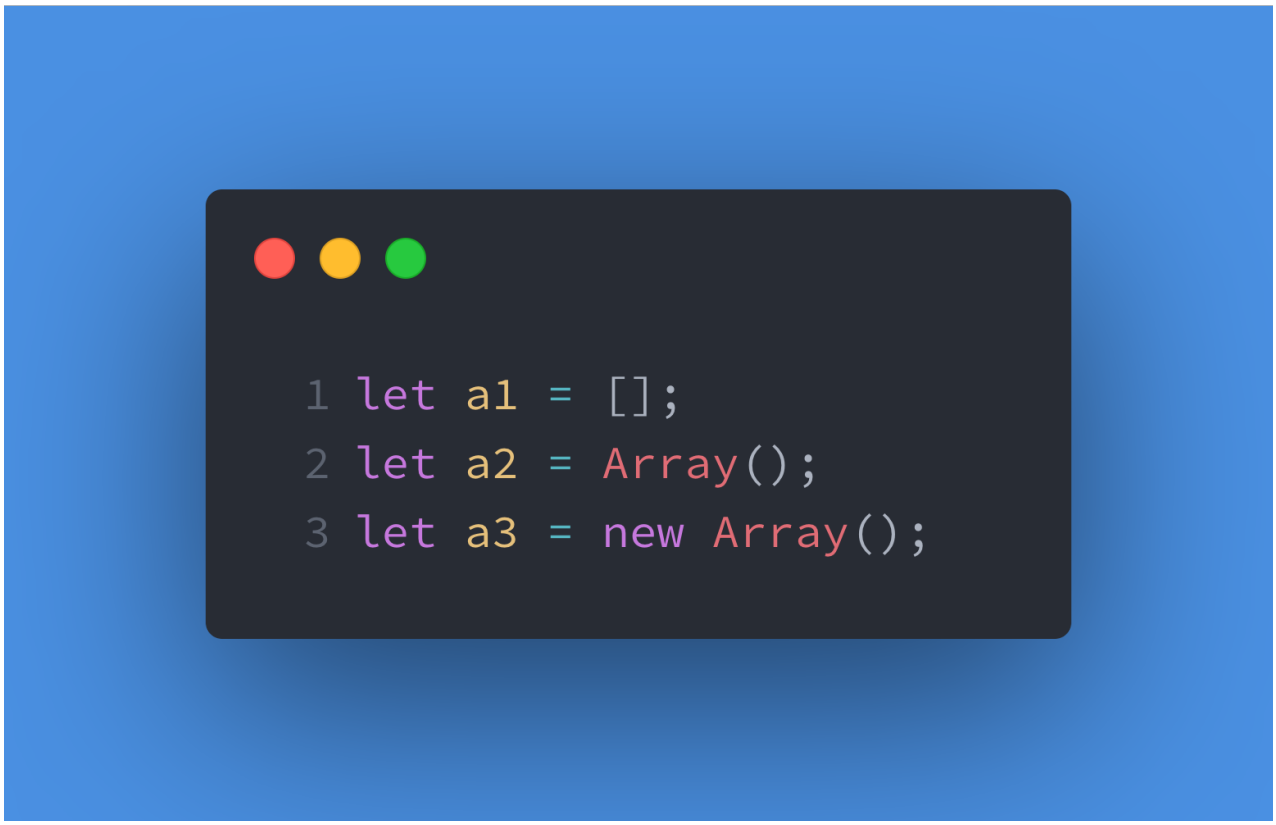
In the second line, there are two typecasts, first `!str` gives us `false` and then `+false` converts boolean into a number `0`.

Eventually, in the third line, the `typeof` operator looks up the current type of `str` which is `number`.

---

**ANSWER**: string `number` will be printed to the console

# JS Test #2: How to create an array in JavaScript




What's the correct way to create an array in JS?

The first array, `a1` is declared using an empty array literal. There are no issues with it. In fact, it's one of the most common ways of declaring an array in JS.

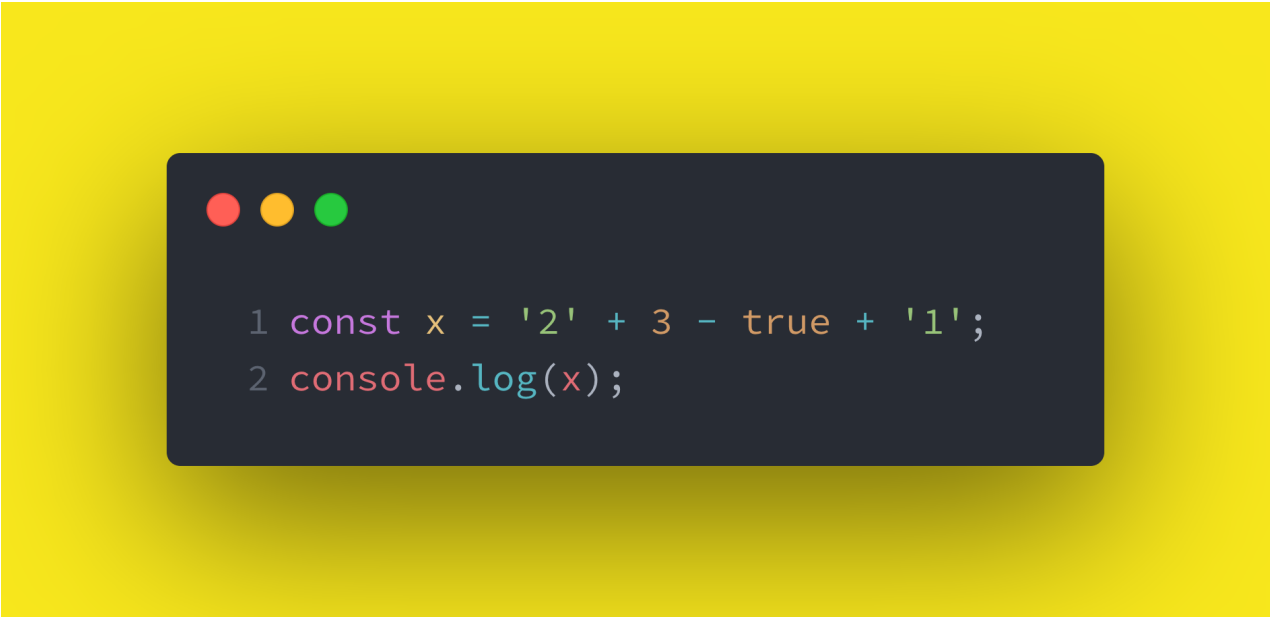
The second one, `a2` is created by calling `Array` as a function. It's fine and it creates an empty array as well.

The third array, `a3` is created using an explicit call to the `Array` constructor as we use the `new` keyword there.

---

**ANSWER**: All arrays are created correctly

# JS Test #3: Adding strings to numbers and booleans




Will we see any output? If yes, then what would it be?

To answer this question correctly, you need to understand the typecast rules in JS.

The arithmetic operations `+` and `-` have the same priority, so the value of `x` will be calculated from left to right without any exceptions.

First, we concatenate the string `'2'` with the number `3`. The result is the string `'23'`.

Second, we try to subtract the boolean value `true` from the string `'23'`. To make this operation possible, both boolean and a string have to be cast to a number. Non-surprisingly `'23'` becomes `23` and `true` is turned to `1`. Eventually, we do the subtraction and get the result, number `22`.

The last step is to add the string `'1'` to the number `22`. Applying the same concatenation that we did on the first step gives us the result - a string `'221'`.

---

**ANSWER**: there are no issues with the expression in line 1. The value of `x` is a string `’221’`, which will be successfully logged to the screen.

# JS Test #4: 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]);
```

How will the `try/catch` blocks behave? What will be logged to the console?

So, we have 2 variables and 2 `try/catch` blocks that supposedly catch errors and put them into `e1` and `e2`.

Then, the content of errors is analyzed, compared and the comparison result is logged to the screen.

First, let's determine what's inside of `e1` and `e2`. To do that, we need to check the code in the `try` blocks. Both trying to get to `null.length` and `undefined.length` will throw an error as neither `undefined` nor `null` have the `length` property.

These errors will be caught in the catch blocks as `e` and then assigned to the variables `e1` and `e2`.

The content of these errors will be a bit different. If we were to log `e.message` to the screen in the catch block, we would see the following:

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

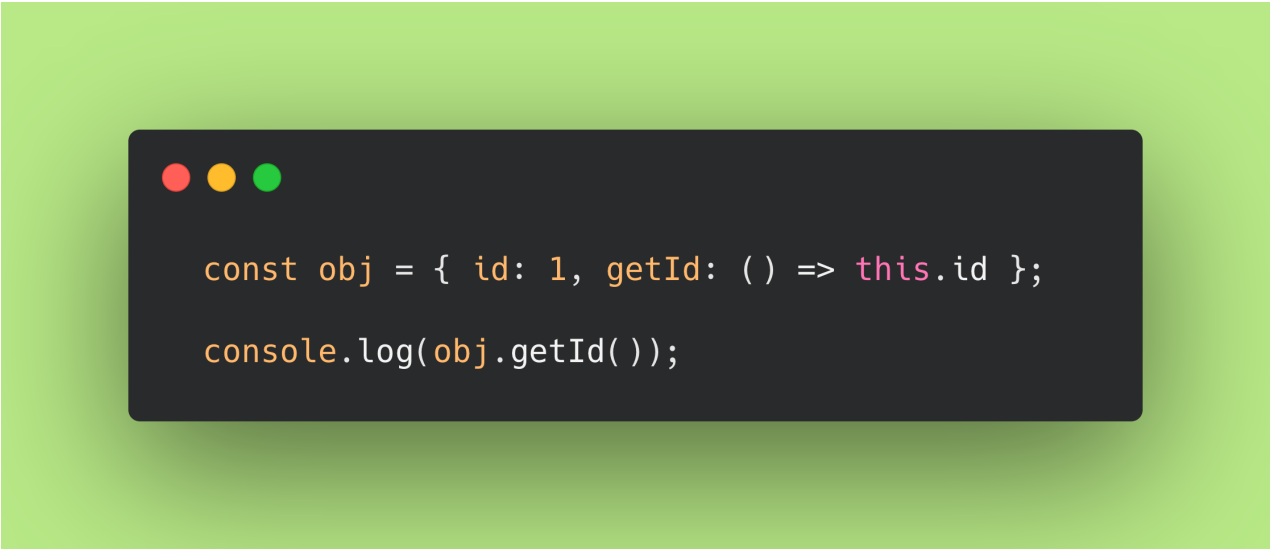
Then, `.split(' ')[0]` gives us the first words of these sentences which is `Cannot` in both cases. So ultimately, the program can be simplified to:

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

---

**ANSWER**: the expression in the `console.log` will be evaluated as `true` and logged to the screen.

# JS Test #5: Can you use an arrow function as a getter?"




Are there any issues with the `getId` function? What will be logged to the screen?

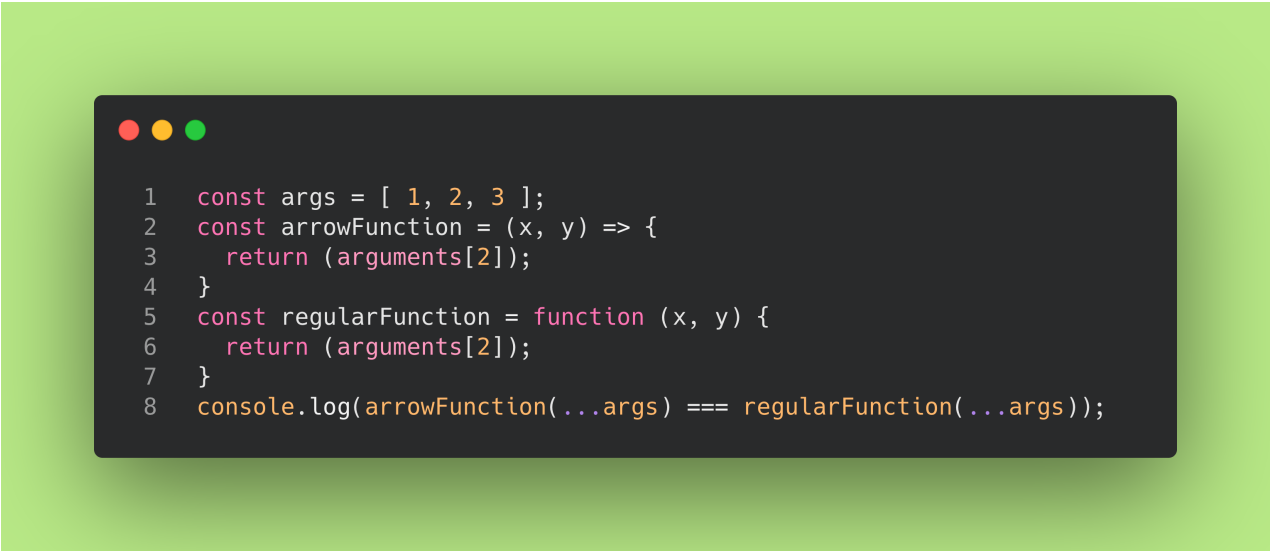
So, `getId` is an arrow function, thus it doesn't have `this` of its own.

It's not bound to `this` of the `obj` object and when we try to get `this.id` it will be evaluated to `undefined` and not `1`.

---

**ANSWER**: `undefined` will be logged to the console.

# JS Test #6: Variable number of arguments in JavaScript



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

`true` or `false`? That is the question...

In JS, all functions have access to the internal `arguments` array that holds all arguments that were passed into the function.

We can access the elements of this array by index, thus expecting that both `regularFunction` and `arrowFunction` will return true.

The only issue is that arrow functions don't have access to the `arguments` array.

There might be two separate outcomes in line 8. Most likely you'll see the message `ReferenceError: arguments is not defined`. However, there also might be a different scenario. For example, if you run this code in Node.js, `arguments[2]` is likely to be evaluated to something like

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

In which case, we'll see `false` logged to the screen as `3` is not equal to the object described above.

---

**ANSWER**: `false` or `ReferenceError` will appear in the console depending on the execution environment

# JS Test #7: Is this an array?

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

Let's find out what's the deal with JavaScript arrays. Is `array` an array?

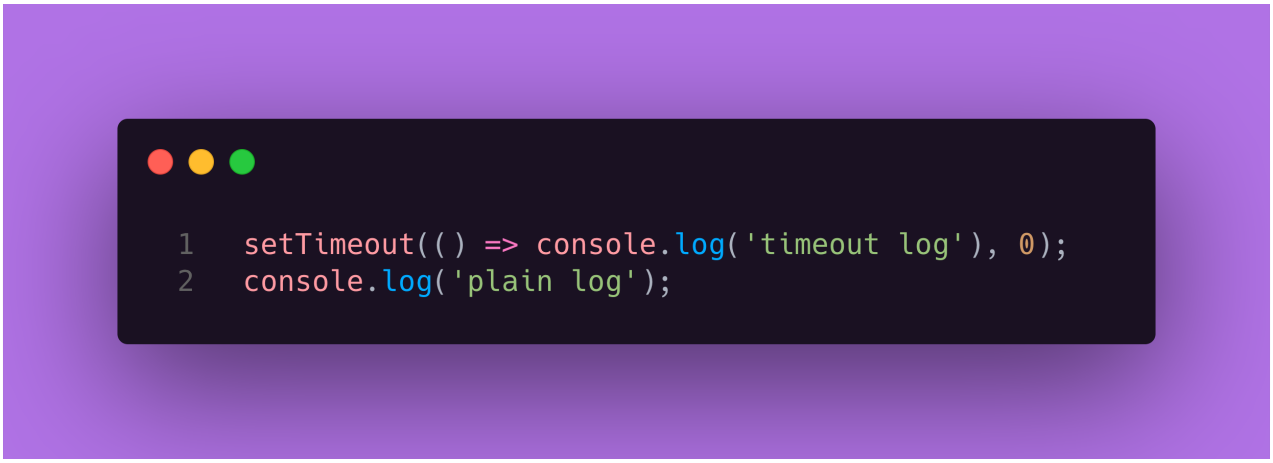
In line one we create an array and bind it with the `array` constant. Then the type of the value of this constant is evaluated by the `typeof` operator.

There's no such type as `array` in JS, so it's impossible to see the message `ARRAY!` on the screen. In fact, all JS arrays are objects, so the execution goes into the `else` branch, and `SOMETHING WEIRD` is printed on the screen.

---

**ANSWER**: `SOMETHING WEIRD` will be logged to the screen as all JS arrays have the type `object`.

# JS Test #8: Zero timeout




What if we call `setTimeout` with 0 delay in JavaScript? Which of the messages will be printed first?

In JS, `setTimeout(func, delay)` takes a function `func` and delays its execution by `delay` milliseconds.

It may seem that if we set the delay to `0`, then the function will be executed immediately, but it's not the case.

The function will be placed in the **message queue** to run asynchronously. This will happen only after the current synchronous execution is done.

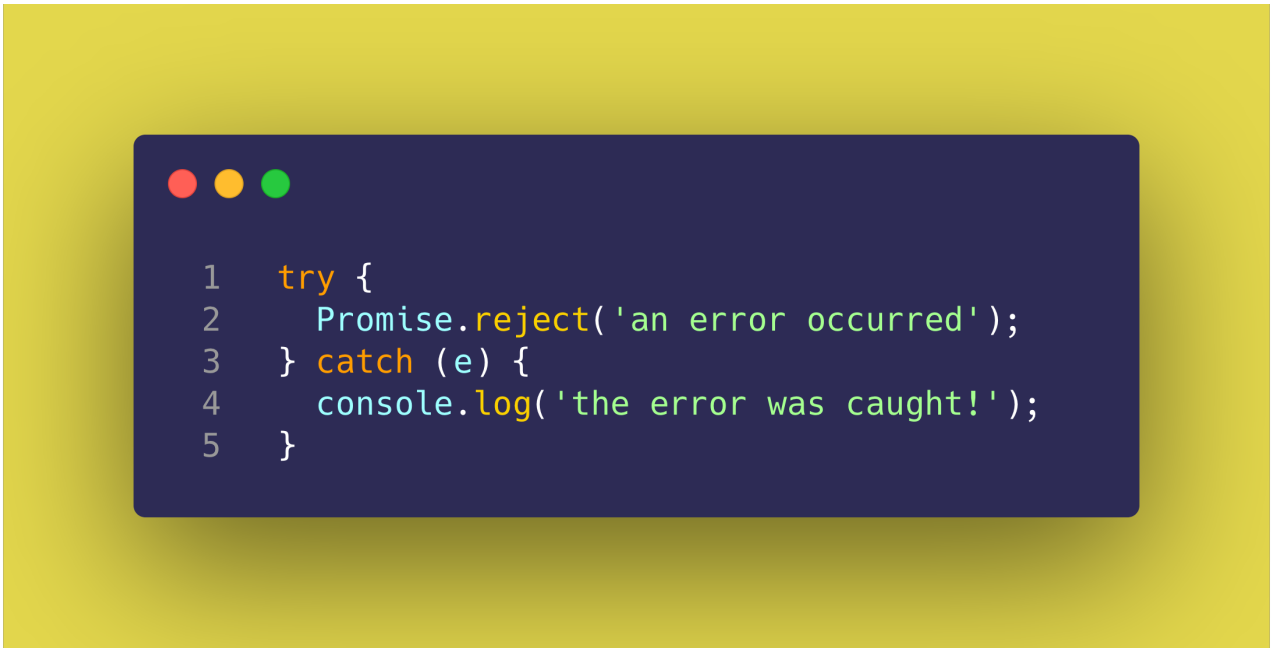
The `console.log` in the second line is a part of the synchronous execution and will run before the `console.log` in the first line.

In most web browsers `setTimeout(f, 0)` has a delay of approximately 3 ms which is determined by the speed of internal processing.

---

**ANSWER**: The message `plain log` will be printed first and then the message `timeout log` will follow.

# JS Test #9: Promise.reject + try/catch




Let's try to reject the promise inside of the JS `try/catch`. Will we catch the error in the `catch` block?

Regular `try/catch` blocks only catch errors that appear in the synchronous code.

As the `Promise` in the second line doesn't have its own asynchronous `.catch` block, the rejection will be left unhandled.

An `UnhandledPromiseRejectionWarning` will be raised and the code inside of the regular `catch` block will not be executed.

---

**ANSWER**: The error will not be caught and the message `the error was caught!` will NOT be logged to the console.

# JS Test #10: null + undefined


What's the difference between `null` and `undefined` in JavaScript? What will be logged to the console?

In the first line, we evaluate `null === null` and the result is `true`.

In the second line, we evaluate `undefined === undefined` and the result is `true` once again.

In the third line, however, we need to understand what the result of `null + undefined` is. For JavaScript, it's hard to make sense of what it should be, so it evaluates this expression as `NaN`.

Now, is `NaN` equal to `NaN`?

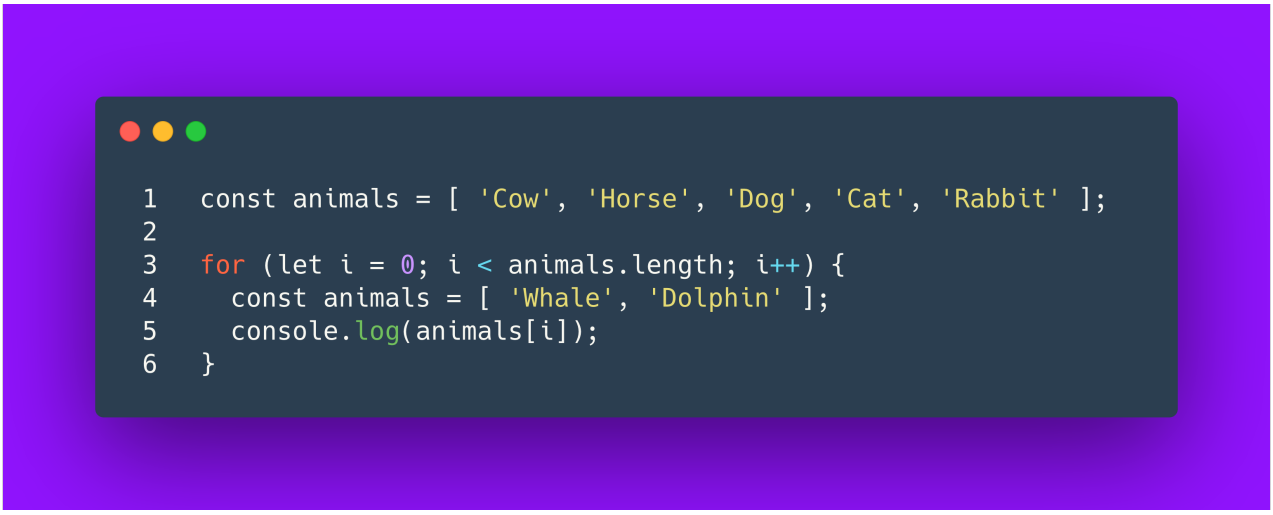
And the answer is - **NO**.

In JS `NaN` is the only value that's not equal to itself.

---

**ANSWER**: The output is going to be `true`, `true`, and `false`.

# JS Test #11: Scope



```
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   }
```

Variables with the same name in JavaScript? What will be logged to the console?

In the first line, we see an array `animals` that holds 5 strings.

The length of this array is used in the loop condition, so the loop will continue spinning up to the point when `i` becomes equal to `5`.

Inside of the loop, a new array is declared with the same name `animals`. There are no issues with such a declaration and no errors will be thrown.

It's important to remember, though, that the value `animals.length` in the loop condition is attributed to the external array with 5 elements but the `console.log` picks up the inner array, which has only 2 elements in it.

Once we go `out of bounds` there will be no error like in `C++` or `Java`. Instead, we'll get `undefined` as the result of the last 3 iterations of the loop.

---

**ANSWER**: The strings `Whale`, `Dolphin` will be logged to the console, followed by `undefined`, `undefined`, `undefined`.

# JS Test #12: Math.min()


How small is `Math.min()` in JavaScript?

The function `Math.min()` takes the variable number of arguments and returns the lowest number passed into it.

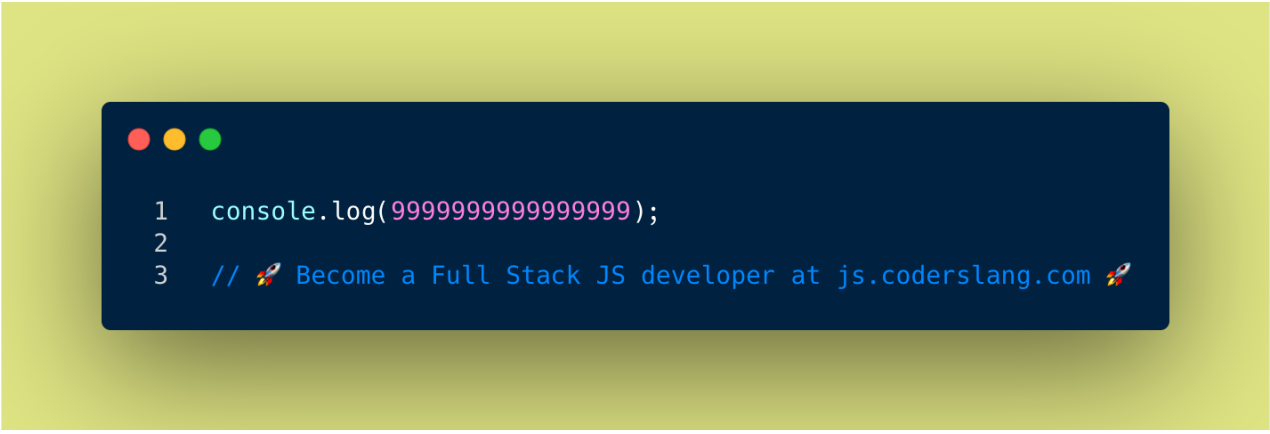
In our case, it's called without any arguments which is a special case.

If `Math.min()` is called without any parameters, it will return `Infinity` which is the opposite of what you might have expected.

---

**ANSWER**: `true` will be printed to the screen as `Infinity` is greater than `0`.

# JS Test #13: Big number




We're just logging a number, what can go wrong here?

Under the hood, there are no integers in JavaScript.

All numbers are represented as `64-bit` floats. This is also known as `double precision`.

`52 bits` are used to store digits, `11 bits` serve to track the position of the decimal point, and `1 bit` holds the sign and determines whether the number is positive or negative.

When there's not enough "space" to store the whole number, then rounding to the nearest possible integer occurs.

It's impossible to store the number `9999999999999999` using `52 bits`, so the rounding gets rid of the least significant digits which leads to the result of `10000000000000000`.

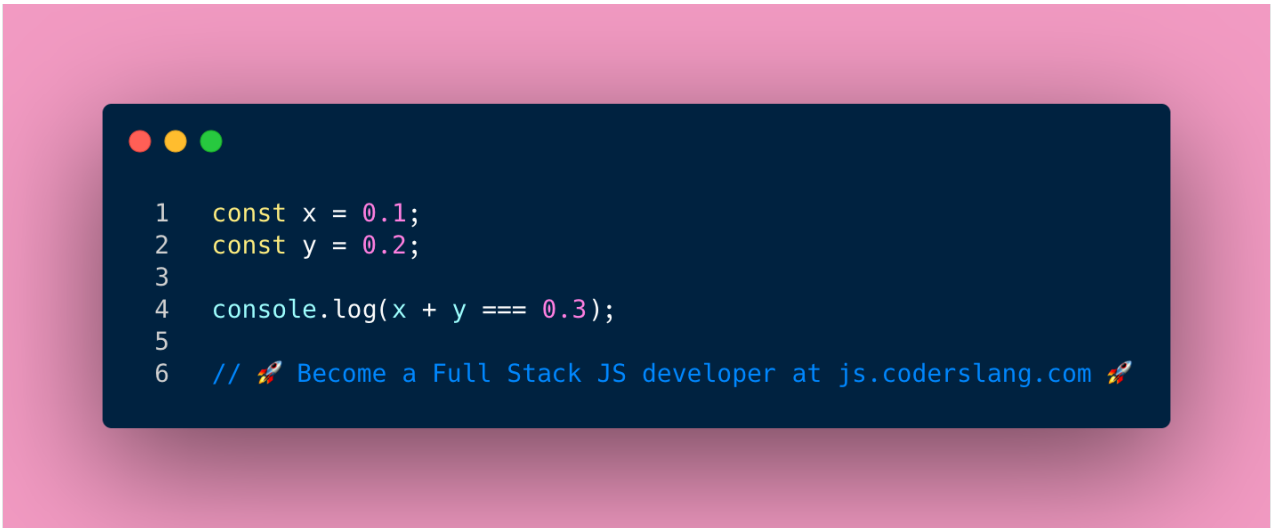
In JavaScript, no error will be thrown in this case.

If you haven't quite understood what's going on here, make sure to read the lecture on **Binary Number System** of my Full Stack JS course CoderslangJS.

---

**ANSWER**: `10000000000000000` will be printed to the screen.

# JS Test #14: 0.1 + 0.2 = ?




JavaScript math is weird. What's the output? True or false?

Inside the computer, all numbers are stored in the **Binary Number System**.

To keep it simple, it's the sequence of `bits` - which are "digits" that can be either `0` or `1`.

The number `0.1` is the same as `1/10` which can be easily represented as a decimal number. In binary, it will result in an endless fraction, similar to what `1/3` is in decimal.

All numbers in JavaScript are stored as `64-bit` signed floating-point values, and when there's not enough space to hold the value, the least significant digits are rounded.

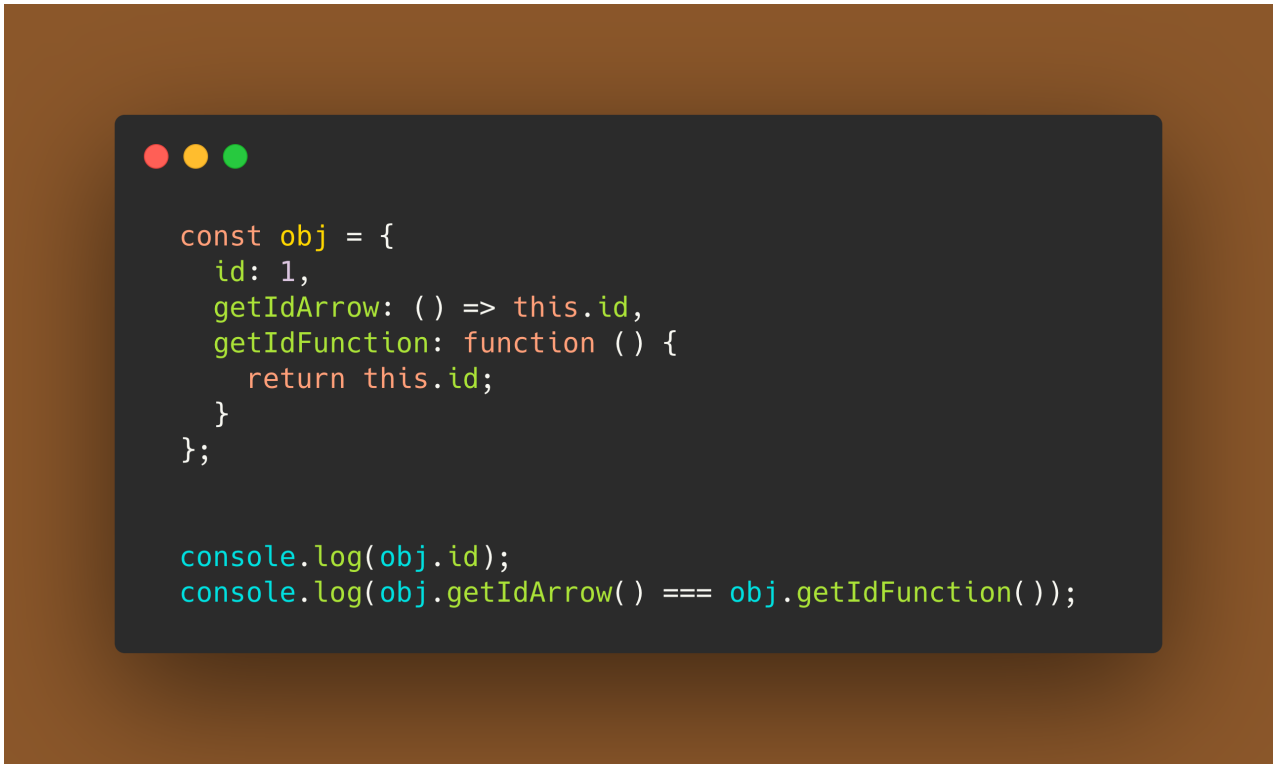
This leads us to the fact that in JavaScript `0.1 + 0.2` render `0.30000000000000004` and not `0.3` like you would have obviously though.

There's a whole lecture dedicated to the **Binary Number System** in my Full Stack JS course CoderslangJS.

---

**ANSWER**: `false` will be printed on the screen.

# JS Test #15: Getter function



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


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

Can we use an arrow function as a getter in JavaScript? What's going on and what will be logged to the console?

So, we have an object with a single field `id` equal to `1` and two functions `getIdArrow` and `getIdFunction` which supposedly do the same thing, return the value of `id`.

Unfortunately, that's not the case. In JavaScript, arrow functions are different from the regular ones. There's no binding between `this` inside of the arrow function and the object `obj`. Thus `this.id` will be evaluated to `undefined`.

In case of the `getIdFunction`, `this` is bound to the `obj` and `this.id` is the same as `obj.id`, which is `1`.

---

**ANSWER**: the first `console.log` will print the number `1` to the console. The second one will print `false`.

# JS Test #16: typeof NaN


What's the type of `Not a Number`?

In JavaScript, `NaN` means `Not a Number`. This is a special value that appears whenever JS can't make sense of the numerical expression.

`NaN` also often appears during typecast. For example, if you try to convert the string into a number, the result will be `NaN`.

It may seem counterintuitive, but `NaN` is just a special number. Thus its type is considered to be `number`.

You can tackle this problem from a different angle and try to answer the question:

> What else `typeof NaN` can be?

---

**ANSWER**: the `typeof NaN` is `number`, which will be logged to the console.

# JS Test #17: Sum of two empty arrays in JavaScript

```
if ([] + [] == false) {
  console.log('same');
} else {
  console.log('different');
}

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

Is the sum of two arrays equal to `false`?

To analyze this code snippet we need to understand how type conversion works in JS.

When we try to sum two arrays using the `+` operator, the arrays are first converted to strings and then these strings are concatenated.

An empty array `[]` is evaluated as an empty string. The sum of two empty strings is still an empty string.

> Now, is an empty string equal to `false` or not?

The comparison here is done using the `==` operator. This operator is used to check `loose equality` and does implicit type conversion.

In this case, empty string and `false` are considered equal and the condition of the `if` statement will be evaluated to `true`.

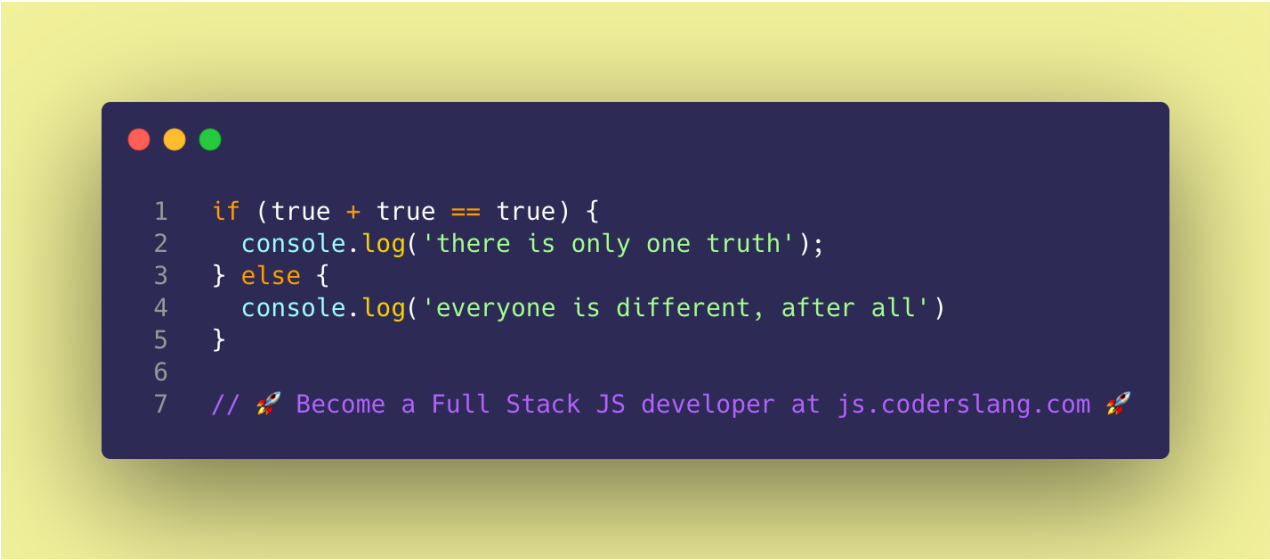
If you want to use a strict comparison which respects the types of values you compare, you should use the strict equality operator `===`.

Here, you can find more information on basic math operations is JavaScript.

---

**ANSWER**: the string `same` will be logged to the console.

# JS Test #18: What's the sum of two booleans in JavaScript?



```
if (true + true == true) {
  console.log('there is only one truth');
} else {
  console.log('everyone is different, after all')
}

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

Can you add booleans in JS? Is something `false` here? What will be logged to the screen?

Just as in the previous test, we're dealing here with `type conversion` and `loose equality` using the `==` operator.

When JavaScript evaluates the expression `true + true` it first converts booleans to numbers, which is `1` for `true` and `0` for `false`.

When we try to do calculate the value of `2 == true`, the typecast happens again and we arrive at the final condition `2 == 1`.

The result is obviously false, so we go into the `else` branch.

To understand how type conversion works with the `+` operator and different data types, you can read this article.

---

**ANSWER**: the string `everyone is different after all` will be logged to the console.

# JS Test #19: Catching the rejected Promise


Can you catch the Promise rejection in JS? Another unhandled rejection?

In JS, it's impossible to catch the unhandled promise rejection using the regular `try/catch` blocks.

So, if the rejection does take place, then we'll likely see a message like `UnhandledPromiseRejectionWarning …` or something along these lines.

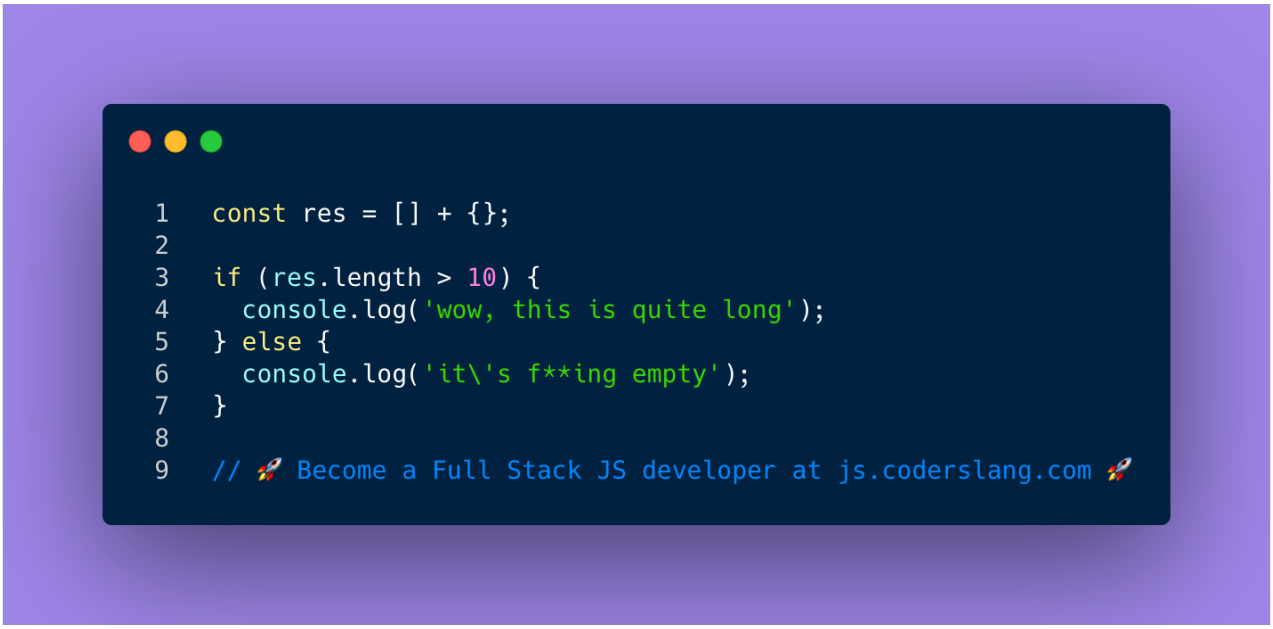
Here, though, we don't get to reject the promise properly.

JavaScript tries to evaluate the result of `null.length` which happens synchronously. An error `Cannot read property 'length' of null` will be thrown and caught in the `catch` block.

---

**ANSWER**: the error will be caught and the string `the error was caught! Cannot read property 'length' of null` will be logged to the screen.

# JS Test #20: Can you sum arrays with objects in JavaScript?




What's the sum of an empty object and an empty array in JS? Is there any `length` to be found?

Once again, we're dealing with type conversion in JavaScript.

Both an empty array and an empty object will be converted to strings.

In the case of an empty array, it will be an empty string.

But for the empty object, we get the string `[object Object]`!

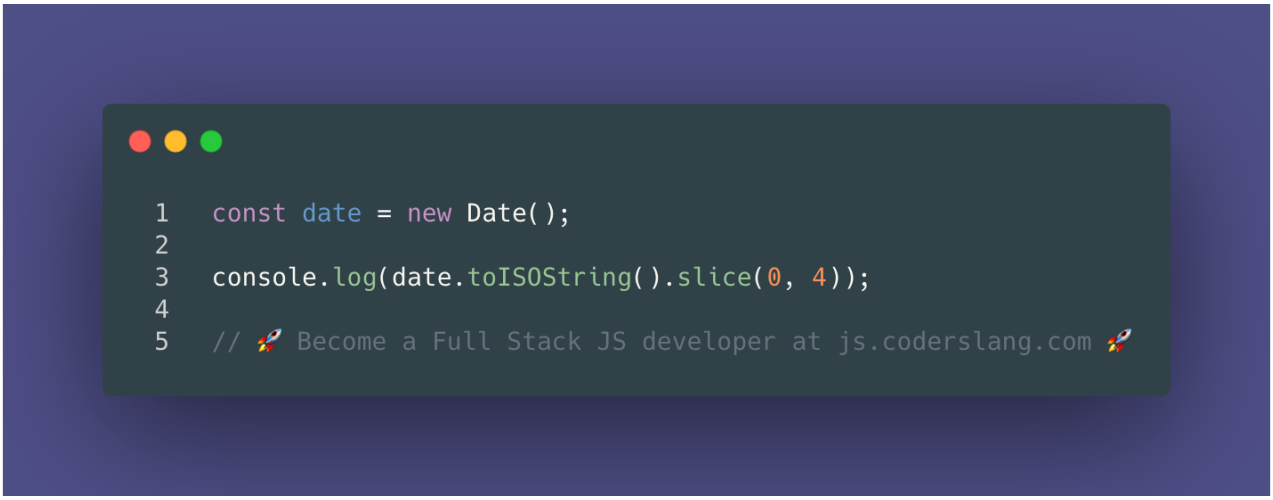
The length of this string is greater than 10, so we'll see the first message logged to the console.

More examples of JavaScript typecast are covered here.

---

**ANSWER**: the string `wow, this is quite long` will appear on the screen.

# JS Test #21: ISO Date




How does the ISO date look in JavaScript? What will be logged to the console?

In the first line, we create a new `Date` object.

It holds the current date and time.

The function `toISOString` returns the string representation of the `date` object in the `ISO` format. It starts with the year and ends with time. Something like `2020-12-27T10:35:26.159Z`.

When we're slicing this string with `slice(0, 4)` we'll get the first four characters, which represent the year.

---

**ANSWER**: the current year will be logged to the console.

# JS Test #22: How `toString` works in JavaScript?


Let's try to apply a generic `toString` function to a regular JavaScript array. What's the output?

In the first line, we've saved the function `Object.prototype.toString` into the constant `toString`. This function is called whenever the object has to be converted to a string.

Most objects provide an overridden implementation of the `toString` function. For example, an array will look like a comma-separated list of all values it holds.

The default behavior of `Object.prototype.toString` is to return a string of the format `[object "TYPE"]`. The "TYPE" is substituted with the actual type of the object. In our case, it's `Array`.

So, with `toString.call(arr)` we call the original implementation of `Object.prototype.toString`.

---

**ANSWER**: the string `[object Array]` will be printed to the console.

# JS Test #23: Array.splice"


How many times can you `splice` the array in JavaScript?

Let's start with the definition of `splice`.

The function `splice` is available in all JavaScript arrays and accepts the variable number of parameters. Here are 4 important things you should know about `splice`:

- The first parameter is called `start` and represents the index of the first element that will be removed from the array.
- The second argument is `deleteCount`. It determines the number of array elements that will be removed from the array
- The third, the fourth argument, and so on, are the new elements that will be added to the array.
- The function `splice` returns the array formed by deleted elements.

Now, we start the array `arr` with 5 elements `[1, 2, 3, 4, 5]`.

The first `splice` extracts 2 elements starting from `arr[1]`. We immediately save them into the `splicedArr`.

Before the final splice we have the following state:

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

The second `splice` once again removes 2 elements from `arr` starting at `arr[1]`. This leaves us with a single element — `1`.

Then, we apply the destructuring with `...` to the `splicedArr` and add elements `2` and `3` to the initial array `arr`.

Here's the code snippet with 2 additional calls to `console.log` to help you understand the explanation better:

```
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);
```

---

**ANSWER**: the array will eventually hold values `[ 1, 2, 3 ]` which will be logged to the console.

# JS Test #24: Adding properties to strings in JavaScript


Can you add a custom field to a regular JS string? What's the output?

The answer to this problem will depend on whether you've added the `'use strict'` flag at the beginning of your script.

The result will be:

- `undefined` if `'use strict'` wasn't specified
- an error will be thrown if you're using the strict mode

So what's the deal?

In the second line, when you try to access `s.user`, JS creates the wrapper object under the hood.

If you're using the strict mode, any modification attempt will throw an error.

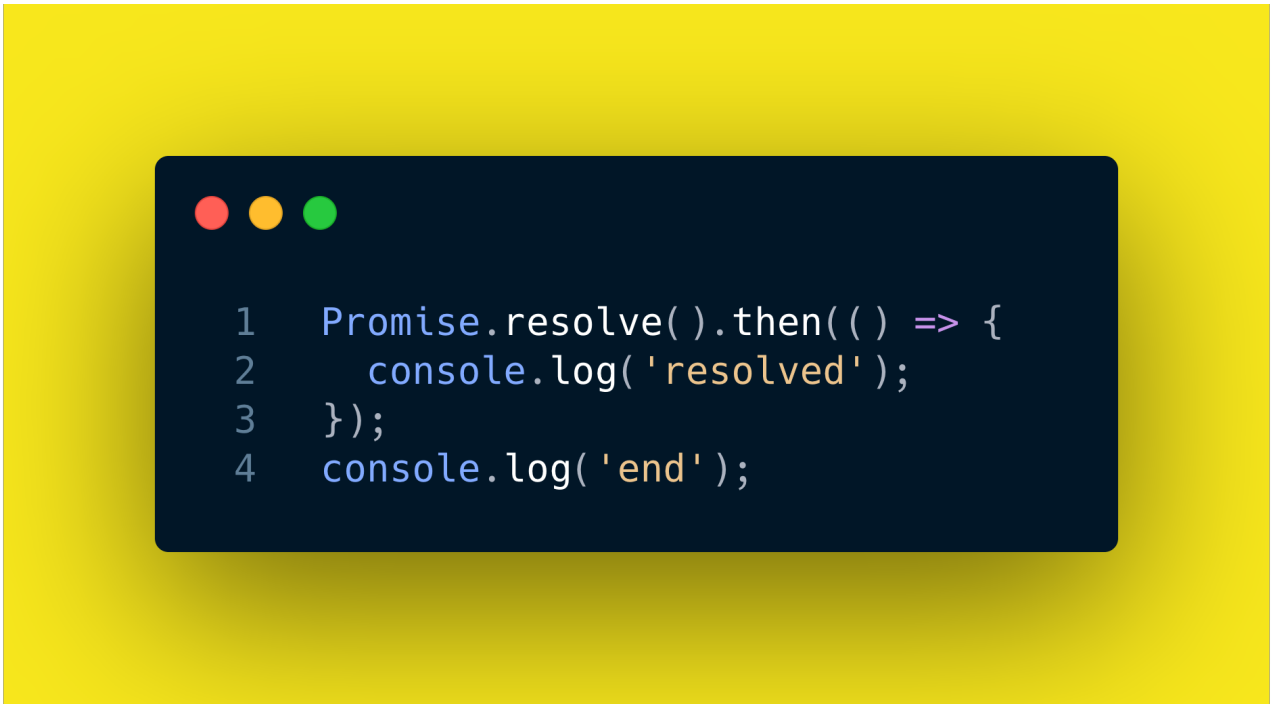
If you're not using the strict mode, then the execution will continue and the new property `user` will be added to the wrapper object.

However, once we're done with the second line of code, the wrapper object is disposed and the `user` property is gone, so `undefined` is logged to the console.

---

**ANSWER**: You can't add properties to primitive values in JS. The result will depend on the presence of the `'use strict'` flag.

# JS Test #25: Immediate Promise.resolve




How fast is `Promise.resolve()` in JavaScript? Which of the messages will be logged first?

The logic is almost the same as in this setTimeout example.

Even though `Promise.resolve()` doesn't have any explicit delay, the code inside of `.then()` is executed asynchronously and has a lower priority than the synchronous code.

So, the `console.log('resolved')` will be executed after the `console.log('end')`.

**ANSWER**: the string `end` will be logged first, followed up by `resolved`.

# JS Test #26: Are these dates equal?

```
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 🚀
```

Are these dates equal?

In the first two lines we create new `Date` objects.

In line 4, we're using the strict equality operator `===`. Remember that we're comparing different objects!

Even if `date` and `date2` represented the same date, the check would have returned `false`.

---

**ANSWER**: The string `not so much` will be printed to the console, as `date1` and `date2` are different objects.

# JS Test #27: Handling errors in JavaScript Promise chains

```
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 🚀
```

Are there any differences between f1 and f2?

There are two ways of providing error handlers to the JavaScript Promises.

The first one is shown in the function `f1`. We pass the `errorHandler` as a second argument to `.then()`.

The second approach is implemented in `f2`. Here, we add the `errorHandler` using the `.catch()` function.

In both cases `errorHandler` will be called if the original `promise` is rejected.

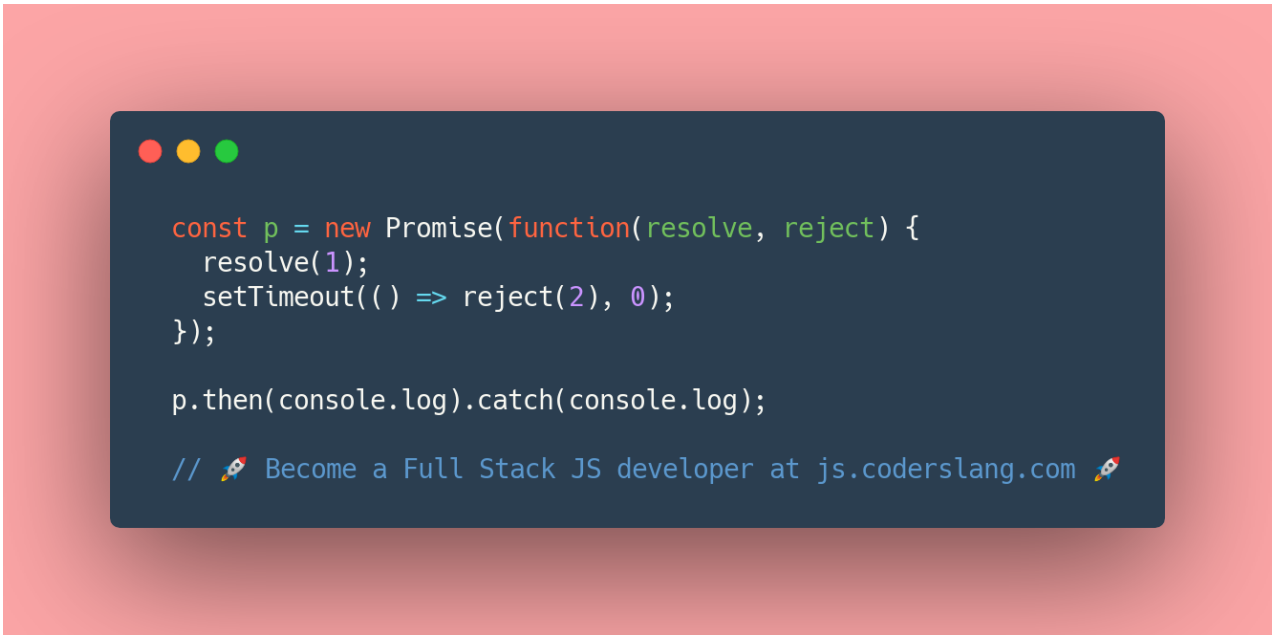
If `promise` resolves successfully, then the execution continues in `successHandler`. And if `successHandler` throws the error, then it will only be handled by `f2` and not `f1`.

This happens because of the internal implementation of `.catch()`. It handles all errors in the promise chain, including the ones inside of the `.then()` handlers.

---

**ANSWER**: Yes, there's a big difference between `f1` and `f2`. The former doesn't handle the error in `successHandler` (if it appears) and the latter does.

# JS Test #28: Resolve and reject at the same time



```
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 🚀
```

Can you resolve and reject a Promise at the same time? What will be printed to the console?

In JavaScript, promises can't be resolved and rejected at the same time.

The execution will never reach the call to `setTimeout` and thus `reject(2)`, inside of it.

Thus only the number `1` will be printed on the screen.

---

**ANSWER**: A single message will be logged to the console. After the promise is resolved with `1` the execution stops and the `setTimeout` won't be called.

# JS Test #29: Slice and dice

```
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 🚀
```

What's happened to `arr`?

`Array.slice` in JavaScript returns the shallow copy of the array. The `start` and `end` indices should be supplied to it as the first 2 parameters. The element at `arr[start]` is included in the copy, and the element at `arr[end]` is not.

Contrary to `Array.splice`, the original array won't be modified when we use `Array.slice`.

So, after the first 2 lines of code, we'll get the following state:

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

Then, we do two actions under `arr.splice`:

- we remove 2 elements from `arr` starting at `arr[1]`. So the original array becomes is `[ 1, 4, 5 ]` at this point.
- we destructure `...slicedArr` and insert its elements into `arr` starting at `arr[1]`. This way we'll get to our final state `[ 1, 2, 4, 5]` in `arr`.

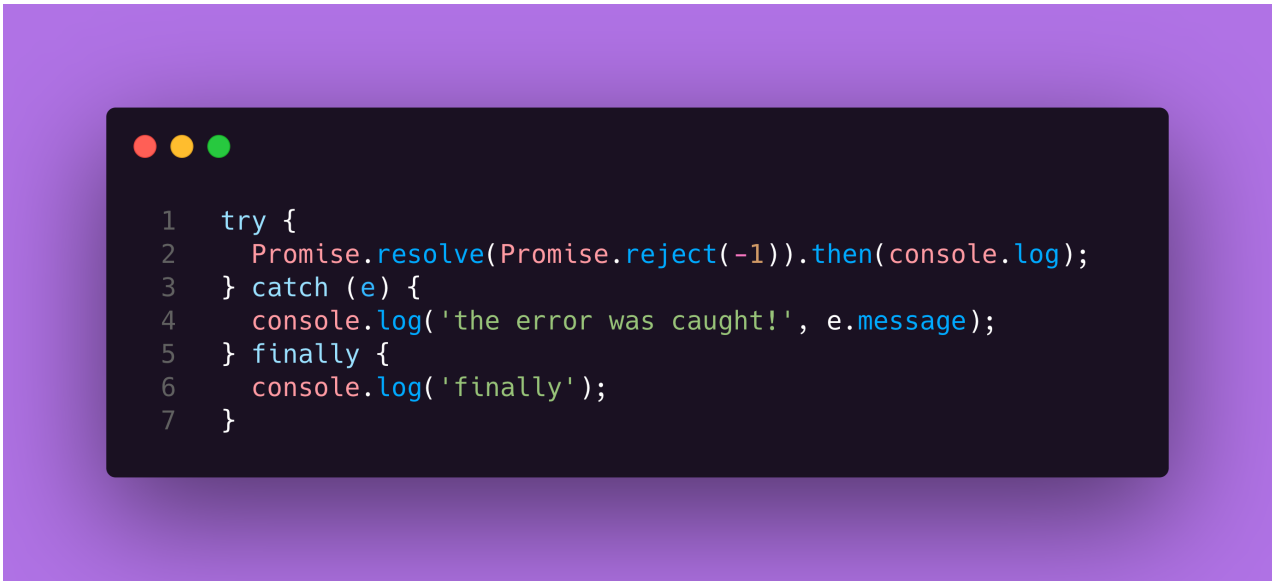
Here's a code snippet with additional logging:

```
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]
```

---

**ANSWER**: The original array `arr` will be modified and hold values `[ 1, 2, 4, 5]`.

# JS Test #30: Reject inside resolve



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

What will be logged to the console? Will the finally block be executed?

To analyze this code snippet, I'll start with the things that are clear:

- the `.then(console.log)` function will not be executed and there's a rejection inside of the `Promise.resolve()`
- the `catch` block won't be able to catch the rejection as it happens asynchronously

So, we're left with the `finally` block. There's a single call to the `console.log` and it's the first message the will be printed on the screen.

Then, the unhandled rejection will happen as we haven't provided the error handler to the promise chain in line 2.

---

**ANSWER**: The string `finally` will be logged to the console followed by the `UnhandledPromiseRejectionWarning: -1`.

# JS Test #31: Big or small


What's the output?

So, there's an `if` statement and it's condition `Math.max() > 0` is all we need to analyze.

If your first guess was that `Math.max()` should return some big number that's for sure bigger than `0`, then you're wrong.

In JavaScript `Math.max()` takes a variable number of arguments and returns the biggest one. The comparison starts at the very bottom, which in JS is `-Infinity` because it's smaller than all other numbers.

This is why if no arguments are provided to the `Math.max()`, it will return `-Infinity`.

As `-Infinity` is smaller than `0`, we'll go into the `else` branch of the conditional statement.

---

**ANSWER**: string `ZERO!` will be logged to the console.

# JS Test #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 🚀
```

What will be logged to the console?

At a first glance, the answer is `true` as `0.1 + 0.1 + 0.1` is obviously equal to `0.3` .

But that's only before we get into the details of how the numbers are represented in JavaScript.

If you try to execute the statement `console.log(0.1 + 0.2)` in JS, you'll get a number `0.30000000000000004`.

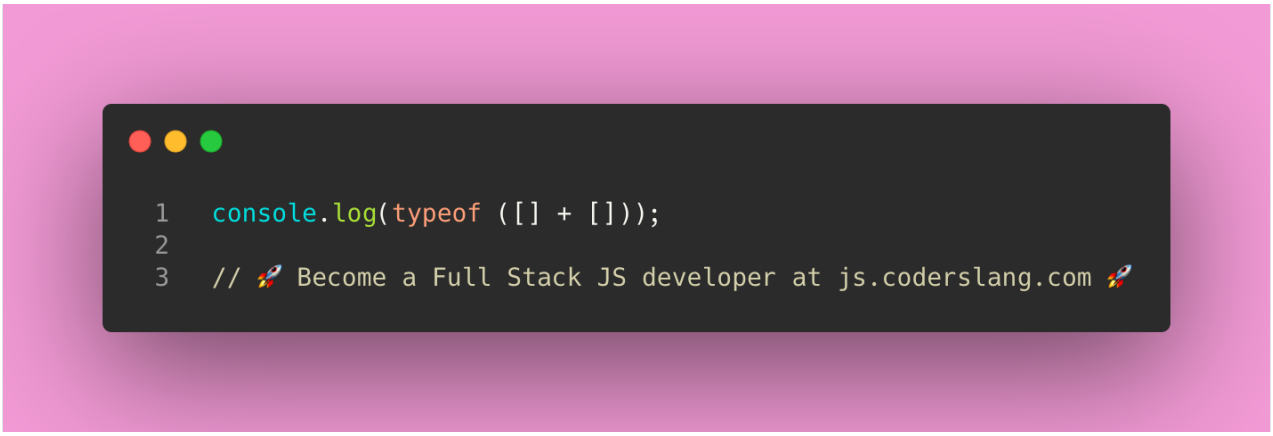
This happens because in JavaScript and quite a few other programming languages some decimal numbers can't be represented exactly as they are.

For example `0.1` in binary will result in an endless fraction, the same way as `1/3` becomes `0.333(3)` in the decimal number system.

---

**ANSWER**: `false` will be logged to the console.

# JS Test #33: Add two empty arrays and check the type




Array? Object? Undefined? What's the output?

In JavaScript, the `+` operator doesn't do the concatenation of arrays.

Instead, it transforms them into strings and then does the string concatenation.

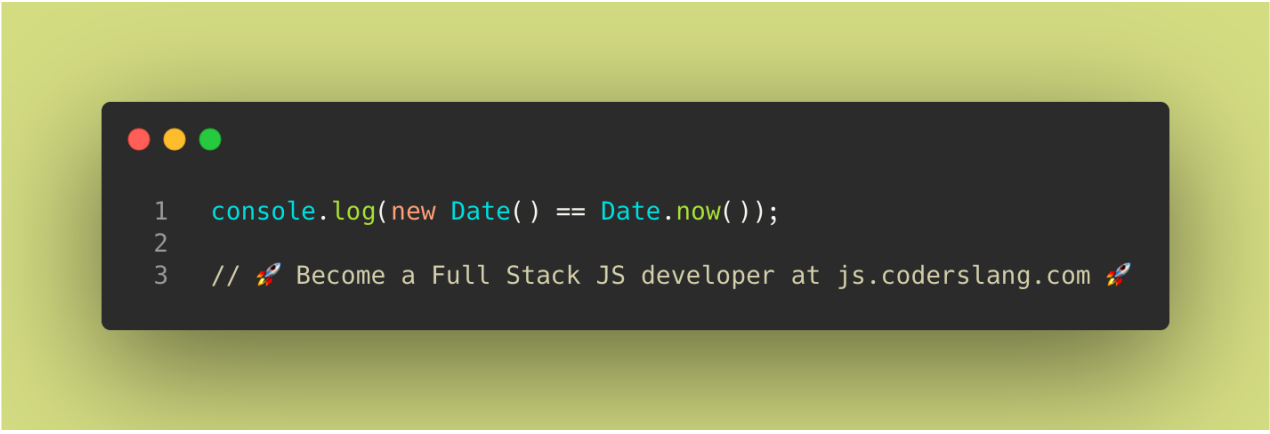
Two empty arrays become two empty strings, and their sum unsurprisingly is still an empty string.

What matters to us is the `typeof` that will return `string` in our case.

---

**ANSWER**: the output will be `string`.

# JS Test #34: Different ways to get current date in JavaScript




How do you prefer to get the current date in JS? What will be logged to the console?

Both `new Date()` and `Date.now()` in JavaScript return us the current date and time.

The difference between them is that `new Date()` returns the `Date` object and `Date.now()` returns the number of milliseconds elapsed from the midnight of Jan 1, 1970.

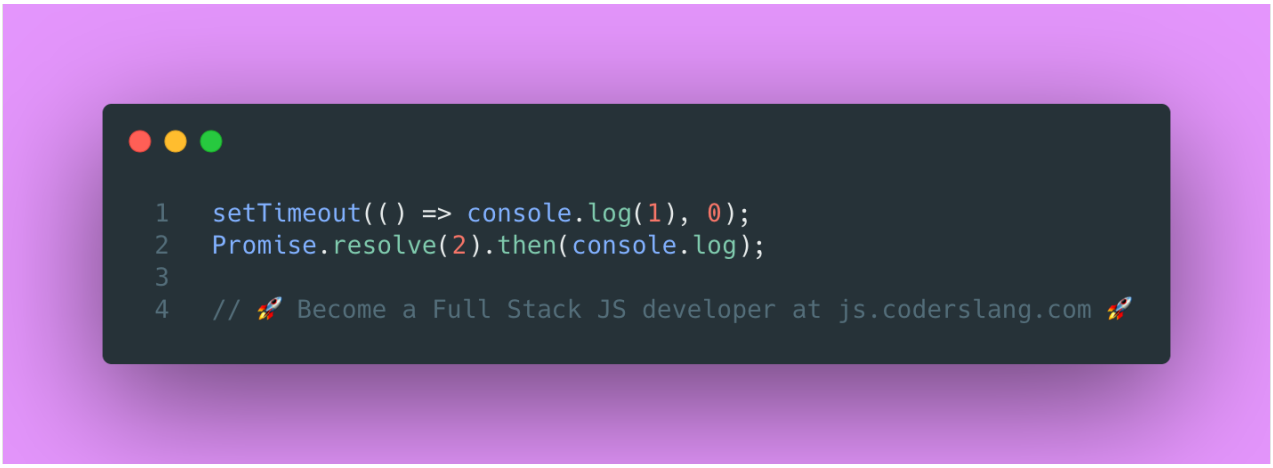
If you need to compare two dates in different formats, you can always get the number of milliseconds from any `Date` object using the built-in `getTime()` function.

Otherwise, you won't have any luck comparing a `number` to an `object`.

---

**ANSWER**: `false` will be logged to the console.

# JS Test #35




What's the order of the output?

Both `setTimeout` and `Promise.resolve` are asynchronous actions, which means that the inner `console.log` statements will be evaluated after some delay.

The difference is that `Promise.resolve` schedules the microtask, and `setTimeout` schedules the macrotask. Micro tasks have higher priority than macrotasks, thus `Promise.resolve` will be evaluated faster and the first output will be `2`.

---

**ANSWER**: `2` will be printed on the first line, followed by `1`.

# Conclusion

**Thank you** for staying with me! I'm sure you've learned at least a couple of new JS tricks to help you ace the next interview.

If some of the problems were not very clear to you, do go ahead and visit learn.coderslang.com. There are multiple JavaScript articles and tutorials to broaden your knowledge and **make you a better dev**.

Here are a couple of things you might want to consider **next**:

- get the app Coderlslang on iOS or Android to **prepare for the tech interview** not only in JS, but also in Java, Node, React, HTML, CSS, QA, and C#
- share this e-book with a friend or write a post about it
- get more tests like these here

And last, but not least! I really appreciate your time reading this book, so here's a **15% discount** code for my Full Stack JS course.

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

If you have any questions, feel free to find me on Twitter, Telegram or just send a plain old email to welcome@coderslang.com