You got 20 out of 25 correct!

1. `true + false`

Output: 1
You answered: `SyntaxError`

You answered incorrectly.

According to the ECMAScript Language Specification, the two boolean values are type coerced into their numeric counterparts.

```
Number(true); // -> 1
Number(false); // -> 0
1 + 0; // -> 1
```

2. `[,,,].length`

Output: 3
You answered: 3

You got it right!

`[,,,]` outputs an array with three empty slots. The last comma is a trailing comma.

If you don't think this is weird enough yet, then take a look at this:

```
[,] + [,]; // -> ""
[] + [] === [,] + [,]; // -> true
[,,,] + [,,,]; // -> ",,,,"
([,,,] + [,,,]).length === [,,,,].length; // -> true
```

To find resources that explain the addition operator with arrays, take a look at the explanation for question 3, directly below this.

3. [1, 2, 3] + [4, 5, 6]

Output: "1,2,34,5,6"
You answered: "1,2,34,5,6"

You got it right!

The extremely simplified answer is that the arrays are converted to strings and are then concatenated. See Denys Dovhan's explanation for how this happens. To learn more about this behavior, visit this StackOverflow answer for a mid-level explanation or this blog post for a more detailed one.

Adding a trailing comma doesn't change anything, by the way:

```
[1, 2, 3,] + [4, 5, 6]; // -> "1,2,34,5,6"
```

But, I suppose, if you really want to convert your arrays to comma-separated strings and combine them, you could write something stupid like this:

```
[1, 2, 3] + [, 4, 5, 6]; // -> "1,2,3,4,5,6"
```

Or, even dumber, this:

```
[1, 2, 3, ""] + [4, 5, 6]; // -> "1,2,3,4,5,6"
```

Probably best not to use the addition operator together with arrays, though. If you do want to combine two arrays for real, this is a better approach:

```
[...[1, 2, 3], ...[4, 5, 6]];
```

4. 0.2 + 0.1 === 0.3

Output: false
You answered: false

You got it right!

This is a standard dilemma of comparing floating-point values. Instead of comparing two floating points directly, one should compare the floating points with some level of tolerance. This StackOverflow answer explains the problem in greater detail.

```
0.2 + 0.1; // -> 0.30000000000000004;
0.2 + 0.1 > 0.3; // -> true
```

5. 10,2

Output: 2
You answered: 2

You got it right!

The comma operator simply returns the value of the last operand.

```
10, 2; // -> 2
1, 2, 3, 4; // -> 4
42, "pineapple", true; // -> true
```

6. !!""

Output: `false`
You answered: `false`

You got it right!

You can add two exclamation marks before any value to get its corresponding boolean primitive. The technical term for this is double NOT (yes, really, Borat would love it). The conversion is based on the truthyness or falsyness of the value.

```
!!""; // -> false
!!0; // -> false
!!"Pineapple"; // -> true
!!42; // -> true
```

This same conversion can be done by using the `Boolean` function.

```
Boolean(""); // -> false
```

7. +!![]

Output: 1
You answered: 1

You got it right!

Arrays are truthy, hence the double NOT operator will output true. The plus character then converts true to its numeric representation: 1.

```
!![]; // -> true
+true; // -> 1
```

This expression might become clearer if written explicitly.

```
Number(Boolean([])); // -> 1
```

8. parseInt(0.0000005)

Output: 5
You answered: NaN

You answered incorrectly.

Yeah, this is genuinely wild.

```
parseInt(0.5); // -> 0
parseInt(0.05); // -> 0
parseInt(0.005); // -> 0
parseInt(0.0005); // -> 0
parseInt(0.00005); // -> 0
parseInt(0.000005); // -> 0
parseInt(0.0000005); // -> 5
```

The `parseInt` function converts its first argument into a string (if it isn't one already) and *then* to a number. When 0.0000005 is converted into a string, this happens:

```
String(0.0000005); // -> "5e-7"
```

The `parseInt` function then simply takes the first character from that string, namely 5, and parses only that character into a number. Full credit to Dmitri Pavlutin for explaining this behavior. Please check out his website for a more detailed explanation (as well as an alternative function that you could use instead).

9. `true == "true"`

Output: `false`
You answered: `true`

You answered incorrectly.

According to the rules of abstract equality comparison, both of these values are converted to numbers when compared.

```
Number(true); // -> 1
Number("true"); // -> NaN
1 == NaN; // -> false
```

10. `010 - 03`

Output: 5
You answered: `NaN`

You answered incorrectly.

010 is treated as an octal number by JavaScript. Thus, its value is in base 8. The MDN Web Docs provides a brief explanation of octal numbers.

```
010; // -> 8
03; // -> 3
8 - 3; // -> 5
```

You can go all out with octal numbers, if you'd like:

```
01111111111111111; // -> 40210710958665
```

By the way, the number of leading zeroes doesn't matter:

```
010 === 0000000010; // -> true
```

11. "" - - ""

Output: 0
You answered: 0

You got it right!

These two empty strings are both converted to 0.

```
Number(""); // -> 0
0 - - 0; // -> 0
```

The expression might become a bit clearer if I write it like this:

```
+"" - -"";
+0 - -0;
```

Please note that, while I put the space between the minus sign and the empty string simply to attempt to confuse you, the space between the minus signs themselves is important:

```
- -""; // -> 0
--""; // -> SyntaxError
```

12. null + 0

Output: 0
You answered: 0

You got it right!

Null converts to its numeric representation: 0.

```
Number(null); // -> 0
0 + 0; // -> 0
```

This also means that while...

```
null === false; // -> false
```

... this is true:

```
+null === +false; // -> true
```

13. 0/0

Output: `NaN`
You answered: `NaN`

You got it right!

As there is no meaningful numerical answer to the equation 0/0, the output is simply `NaN`.

```
isNaN(0/0); // -> true
```

14. 1/0 > 10 ** 1000

Output: `false`
You answered: `true`

You answered incorrectly.

JavaScript treats both of these values as infinite, and infinity is equal to infinity. Learn more about infinities.

```
1/0; // -> Infinity
10 ** 1000; // -> Infinity
Infinity > Infinity; // -> false
```

The exponentiation operator `**` is basically the same thing as the `Math.pow` function.

```
10 ** 1000; // -> Infinity
```

```
Math.pow(10, 1000); // -> Infinity
```

15. true++

Output: SyntaxError
You answered: SyntaxError

You got it right!

Our first and only syntax error. I put SyntaxError as an option on a lot of the questions, hoping that some users would find some syntax so bizarre that it could not possibly be allowed. So, I felt that I had to add at least one expression that actually does result in a SyntaxError.

Some global variables and functions won't throw syntax errors, by the way:

```
true++; // -> SyntaxError
1++; // -> SyntaxError
"x"++; // -> SyntaxError
null++; // -> SyntaxError
undefined++; // -> NaN
$++; // -> NaN
console.log++; // -> NaN
NaN++; // -> NaN
```

And, of course, just to be completely clear, this is valid syntax:

```
let _true = true;
_true++;
_true; // -> 2
```

16. "" - 1

Output: -1
You answered: -1

You got it right!

While the addition operator (+) is used both for numbers and strings, the subtraction operator (-) has no use for strings, so JavaScript interprets this as an operation between numbers. An empty string is type coerced into 0.

```
Number(""); // -> 0
0 - 1; // -> -1;
```

This would still be true even if the string had a space (or more) inside of it:

```
" " - 1; // -> -1;
```

However, if we use the addition operator, then string concatenation takes priority:

```
"" + 1; // -> "1";
```

17. (null - 0) + "0"

Output: `"00"`
You answered: `"00"`

You got it right!

`null` is coerced into 0.

```
Number(null); // -> 0
0 - 0; // -> 0
0 + "0"; // -> "00"
```

If the question had used only the subtraction operator, the result would have been different:

```
(null - 0) - "0"; // -> 0
```

18. true + ("true" - 0)

Output: NaN
You answered: NaN

You got it right!

You might suspect that JS is so bananas that it would convert the string literal "true" into the numeric representation of the boolean true. It's not quite that bananas, however. What actually happens is that it tries to convert the string to a number and fails.

```
Number("true"); // -> NaN
```

19. !5 + !5

Output: `0`
You answered: `0`

You got it right!

Putting a solemn exclamation mark, also known as the logical NOT operator, before a non-Boolean value will convert it to a boolean value opposite of what the `Boolean` function would convert it into.

5 is truthy:

```
Boolean(5); // -> true
!!5; // -> true
```

The opposite of true is, of course, false, which in turn is coerced into 0:

```
!5; // -> false
+false; // -> 0
0 + 0; // -> 0
```

20. [] + []

Output: `" "`
You answered: `" "`

You got it right!

This question is closely tied to question 3. Again, the extremely simplified answer is that JavaScript converts the arrays to strings. Scroll up to question 3 to find resources that explain this behavior.

```
[].toString(); // -> ""
"" + ""; // -> ""
```

Also, like I mentioned in the explanation for question 2, these expressions are equal, due to trailing commas:

```
[] + [] === [,] + [,]; // -> true
```

Even though these arrays are different, they are both converted to empty strings:

```
[].length; // -> 0
[,].length; // -> 1
[].toString() === [,].toString(); // -> true
```

Of course, this is also true:

```
Number([]) === Number([,]); // -> true
```

21. 1 + 2 + "3"

Output: "33"
You answered: "33"

You got it right!

JavaScript will execute these operations from left to right. String concatenation will take priority when the number three is added with the string three.

```
1 + 2; // -> 3
3 + "3"; // -> "33"
```

Since the operations are executed from left to right, the expressions will give a signficantly different output if it was the first or second operation that had contained a string:

```
"1" + 2 + 3; // -> "123"
1 + "2" + 3; // -> "123"
```

22. typeof NaN

Output: `"number"`
You answered: `"number"`

You got it right!

The ECMAScript Language Specification explains NaN as a number value that is a IEEE 754 “Not-a-Number” value. It might seem strange, but this is a common computer science principle.

There are some odd issues surrounding NaN in JavaScript, however. For instance, this is one of the two only instances where the `Object.is` function disagrees with triple equals.

```
NaN === NaN; // -> false
Object.is(NaN, NaN); // -> true
```

If you want to see if a value is NaN, then you should always use the `isNaN` function.

```
isNaN(NaN); // -> true
```

Unrelated, but in case you are curious about the other case where the `Object.is` function disagrees with triple equals, it's 0 and -0.

```
0 === -0; // -> true
Object.is(0, -0); // -> false
```

23. undefined + false

Output: `NaN`
You answered: `NaN`

You got it right!

While false can be converted to a number, undefined cannot.

```
Number(undefined); // -> NaN
Number(false); // -> 0
NaN + 0; // -> NaN
```

However, undefined *is* represented by false:

```
!!undefined === false; // -> true
```

Which means that we can add undefined and false like so:

```
!!undefined + false; // -> 0
```

24. "" && -0

Output: ""
You answered: ""

You got it right!

Chances are that you've only ever used the logical AND operator in if-statements, but when used by itself it actually returns the value of one of the operands. If the first expression is truthy, then it returns the second. Otherwise, it returns the first. An empty string is falsy, hence it returns the first operand.

```
"" && -0; // -> ""
-0 && ""; // -> -0
5 && 3; // -> 3
0 && 3; // -> 0
```

25. +!!NaN * "" - - [,]

Output: 0
You answered: 0

You got it right!

The finale combines some of the bizarre syntax that I've covered in this quiz. I've explained all of this behavior already in the explanations above, except for the multiplication operator (*), which will coerce the empty string into its numeric equivalent: 0.

```
+!!NaN; // -> 0
+""; // -> 0
-[,]; // -> -0
```

Add it all together:

```
0 * 0 - -0; // -> 0
```

I hope you thought this little quiz was fun, and, hopefully, you even learned something new. Thank you so much for all the insightful thoughts, funny comments, and amazing feedback that I've gotten on Reddit (especially r/javascript), Hacker News, and by e-mail!

This quiz was made by Jacob Bergdahl. Learn more about me on my website, say hi on LinkedIn, and check out my book while you're at it.

Start over