Falsy vs false

Falsy values are values with an inherent boolean false, the following are falsy values.

0, '' or "", null, undefined, NaN

```
const zero = 0;
const emptyString = "";
if(!zero){
    console.log("0 is falsy");
}
if(!emptyString){
    console.log("An empty string is falsy")
}
console.log(NaN || 1); // => 1 //value is nothing NaN return a number.
console.log(null || 1); // => 1//null return as object;
console.log(undefined || 1); // => 1// return undefined
```

Truthy vs true

Every value that is not falsy is considered truthy, these include

strings, arrays, objects, functions

```
function somethingIsWrong(){
    console.log("Something went horribly wrong")
}
function callback(){
    console.log("Hello From Callback");
}
const string = "Hello world!"
const array = [1,2,3];

const object = {};

if(string){

   console.log(string) // => "Hello world!"

   const functionToCall = callback || somethingIsWrong

   functionToCall() // => "Hello From Callback"

   console.log(array || "That was not meant to happen")

   console.log(object || "This is strange")

}

null >= 0; --> true // bcz null return as obejct

null > 0; --> false

null == 0; --> false

let s = []==0;//true -> bcz array start from 0;

let s = {}==0;//false -> bcz its an object;

0 == false // => true//bcz false value now 0;

0 === false // => false//bcz its also checking type false type is boolean.

[]==[] or []===[] //false, refer different objects in memory

{}=={} or {}==={} //false, refer different objects in memory
```

Number + Number -> addition;

Boolean + Number -> addition;

Boolean + Boolean -> addition;

Number + String -> concatenation;

String + Boolean -> concatenation;

String + String -> concatenation;

```
27.toString() // > Uncaught SyntaxError: Invalid or unexpected token

(27).toString(); // -> '27'

27..toString(); // -> '27'
```

```
function a(x) {

  arguments[0] = "hello";

  console.log(x);

}

a(); // > undefined

a(0); // > "hello"
```

arguments is an Array-like object that contains the values of the arguments passed to that function. When no arguments are passed, then there's no x to override.

```
{}{}; // -> undefined

{}{}{}; // -> undefined

{}{}{}{}; // -> undefined

{foo: 'bar'}{}; // -> 'bar'

{}{foo: 'bar'}; // -> 'bar'

{}{foo: 'bar'}{}; // -> 'bar'

{a: 'b'}{c:' d'}{}; // -> 'd'

{a: 'b', c: 'd'}{}; // > SyntaxError: Unexpected token ':'

({}{}); // > SyntaxError: Unexpected token '{'
```

When inspecting each {}, they returns undefined. If you inspect {foo: 'bar'}{}, you will find {foo: 'bar'} is 'bar'.

There are two meanings for {}: an object or a block. For example, the {} in () => {} means block. So we need to use () => ({}) to return an object.

Let's use {foo: 'bar'} as a block. Write this snippet in your console:

```
if (true) {

  foo: "bar";

} // -> 'bar'
```

Surprisingly, it behaviors the same! You can guess here that {foo: 'bar'}{} is a block.

Math.max() less than Math.min()

```
Math.min() > Math.max(); // -> true

Math.min() < Math.max(); // -> false

Math.min(); // -> Infinity

Math.max(); // -> -Infinity

Infinity > -Infinity; // -> true
```

Why so? Well, Math.max() is not the same thing as Number.MAX_VALUE. It does not return the largest possible number.

Math.max takes arguments, tries to convert the to numbers, compares each one and then returns the largest remaining. If no arguments are given, the result is −∞. If any value is NaN, the result is NaN.

The opposite is happening for Math.min. Math.min returns ∞, if no arguments are given.

Chaining assignments on object

```
var foo = { n: 1 };

var bar = foo;

foo.x = foo = { n: 2 };

foo.x; // -> undefined

foo; // -> {n: 2}

bar; // -> {n: 1, x: {n: 2}}
```

From right to left, {n: 2} is assigned to foo, and the result of this assignment {n: 2} is assigned to foo.x, that's why bar is {n: 1, x: {n: 2}} as bar is a reference to foo. But why foo.x is undefined while bar.x is not ?

💡 Explanation:

Foo and bar references the same object {n: 1}, and lvalues are resolved before assignations. foo = {n: 2} is creating a new object, and so foo is updated to reference that new object. The trick here is foo in foo.x = ... as a lvalue was resolved beforehand and still reference the old foo = {n: 1} object and update it by adding the x value. After that chain assignments, bar still reference the old foo object, but foo reference the new {n: 2} object, where x is not existing.

It's equivalent to:

```
var foo = { n: 1 };

var bar = foo;

foo = { n: 2 }; // -> {n: 2}

bar.x = foo; // -> {n: 1, x: {n: 2}}

// bar.x point to the address of the new foo object

// it's not equivalent to: bar.x = {n: 2}
```

Tricky return

return statement is also tricky. Consider this:

```
(function() {

  return

  {

    b: 10;

  }

})(); // -> undefined
```

return and the returned expression must be in the same line:

```
(function() {

  return {

    b: 10
  };
})(); // -> { b: 10 }
```

This is because of a concept called Automatic Semicolon Insertion, which automagically inserts semicolons after most newlines. In the first example, there is a semicolon inserted between the return statement and the object literal, so the function returns undefined and the object literal is never evaluated.

arguments and arrow functions

Consider the example below:

```
let f = function() {
  return arguments;
};
f("a"); // -> { '0': 'a' }
```

Now, try do to the same with an arrow function:

```
let f = () => arguments;
f("a"); // -> Uncaught ReferenceError: arguments is not defined
```

💡 Explanation:

Arrow functions are a lightweight version of regular functions with a focus on being short and lexical this. At the same time arrow functions do not provide a binding for the arguments object. As a valid alternative use the rest parameters to achieve the same result:

```
let f = (...args) => args;
f("a");
```

Arrow functions can not be a constructor

Consider the example below:

```
let f = function() {
  this.a = 1;
};
new f(); // -> f { 'a': 1 }
```

Now, try do to the same with an arrow function:

```
let f = () => {
  this.a = 1;
};
new f(); // -> TypeError: f is not a constructor
```

💡 Explanation:

Arrow functions cannot be used as constructors and will throw an error when used with new. Because they have a lexical this, and do not have a prototype property, so it would not make much sense.

Tricky arrow functions

Consider the example below:

```
let f = () => 10;
f(); // -> 10
```

Okay, fine, but what about this:

```
let f = () => {};
f(); // -> undefined
```

💡 Explanation:

You might expect {} instead of undefined. This is because the curly braces are part of the syntax of the arrow functions, so f will return undefined. It is however possible to return the {} object directly from an arrow function, by enclosing the return value with brackets.

```
let f = () => ({});
f(); // -> {}
```

Non-coercible objects

With well-known symbols, there's a way to get rid of type coercion. Take a look:

```
function nonCoercible(val) {
  if (val == null) {
    throw TypeError("nonCoercible should not be called with null or undefined");
  }
  const res = Object(val);
  res[Symbol.toPrimitive] = () => {
    throw TypeError("Trying to coerce non-coercible object");
  };
  return res;
}
```

Now we can use this like this:

```
// objects
const foo = nonCoercible({ foo: "foo" });
foo * 10; // -> TypeError: Trying to coerce non-coercible object
foo + "evil"; // -> TypeError: Trying to coerce non-coercible object
// strings
const bar = nonCoercible("bar");
bar + "1"; // -> TypeError: Trying to coerce non-coercible object
bar.toString() + 1; // -> bar1
bar === "bar"; // -> false
bar.toString() === "bar"; // -> true
bar == "bar"; // -> TypeError: Trying to coerce non-coercible object
// numbers
const baz = nonCoercible(1);
baz == 1; // -> TypeError: Trying to coerce non-coercible object
baz === 1; // -> false
baz.valueOf() === 1; // -> true
```

Insidious try..catch

What will this expression return? 2 or 3?

```
(() => {
  try {
    return 2;
  } finally {
    return 3;
  }
})();
```

The answer is 3. Surprised?

Labels

Not many programmers know about labels in JavaScript. They are kind of interesting:

```
foo: {
  console.log("first");
  break foo;
  console.log("second");
}
// > first
// -> undefined
```

💡 Explanation:

The labeled statement is used with break or continue statements. You can use a label to identify a loop, and then use the break or continue statements to indicate whether a program should interrupt the loop or continue its execution.

In the example above, we identify a label foo. After that console.log('first'); executes and then we interrupt the execution.

Comparison of three numbers

```
1 < 2 < 3; // -> true

3 > 2 > 1; // -> false
```

Explanation:

Why does this work that way? Well, the problem is in the first part of an expression. Here's how it works:

```
1 < 2 < 3; // 1 < 2 -> true

true < 3; // true -> 1

1 < 3; // -> true

3 > 2 > 1; // 3 > 2 -> true

true > 1; // true -> 1

1 > 1; // -> false
```

We can fix this with Greater than or equal operator (>=):

```
3 > 2 >= 1; // true
```

[] and null are objects

```
typeof []; // -> 'object'

typeof null; // -> 'object'

// however

null instanceof Object; // false
```

NaN is not a number

Type of NaN is a 'number':

```
typeof NaN; // -> 'number
```

Math with true and false

```
true + true; // -> 2

(true + true) * (true + true) - true; // -> 3
```

undefined and Number

If we don't pass any arguments into the Number constructor, we'll get 0. The value undefined is assigned to formal arguments when there are no actual arguments, so you might expect that Number without arguments takes undefined as a value of its parameter. However, when we pass undefined, we will get NaN.

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

Number(undefined); // -> NaN
```

Array equality is a monster

Array equality is a monster in JS, as you can see below:

```
[] == ''   // -> true

[] == 0    // -> true

[''] == '' // -> true

[0] == 0   // -> true

[0] == ''  // -> false

[''] == 0  // -> true


[null] == ''      // true

[null] == 0       // true

[undefined] == '' // true

[undefined] == 0  // true


[[]] == 0  // true

[[]] == '' // true


[[[[[[]]]]]] == '' // true

[[[[[[]]]]]] == 0  // true


[[[[[[ null ]]]]]] == 0  // true
```

```
[[[[[[ null ]]]]]] == '' // true
```

```
[[[[[[ undefined ]]]]]] == 0  // true

[[[[[[ undefined ]]]]]] == '' // true
```

Trailing commas in array

You've created an array with 4 empty elements. Despite all, you'll get an array with three elements, because of trailing commas:

```
let a = [, , ,];

a.length; // -> 3

a.toString(); // -> ',,'
```

null is falsy, but not false

Despite the fact that null is a falsy value, it's not equal to false.

```
!!null; // -> false

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

At the same time, other falsy values, like 0 or '' are equal to false.

```
0 == false; // -> true

"" == false; // -> true
```

[] is truthy, but not true

An array is a truthy value, however, it's not equal to true.

```
!![]      // -> true

[] == true // -> false
```

Object.is() and === weird cases

Object.is() determines if two values have the same value or not. It works similar to the === operator but there are a few weird cases:

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

NaN === NaN; // -> false
```

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

-0 === 0; // -> true

Object.is(NaN, 0 / 0); // -> true

NaN === 0 / 0; // -> false
```

In JavaScript lingo, NaN and NaN are the same value but they're not strictly equal. NaN === NaN being false is apparently due to historical reasons so it would probably be better to accept it as it is.

Similarly, -0 and 0 are strictly equal, but they're not the same value.

For more details about NaN === NaN, see the above case.

NaN is not a NaN

```
NaN === NaN; // -> false
```

The specification strictly defines the logic behind this behavior:

If Type(x) is different from Type(y), return false.

If Type(x) is Number, then

If x is NaN, return false.

If y is NaN, return false.

… … …

true is false

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

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

Explanation:

Consider this step-by-step:

```
// true is 'truthy' and represented by value 1 (number), 'true' in string form is NaN.

true == "true"; // -> false

false == "false"; // -> false

// 'false' is not the empty string, so it's a truthy value
```

```
!!"false"; // -> true

!!"true"; // -> true
```

baNaNa

```
"b" + "a" + +"a" + "a"; // -> 'baNaNa'
```

This is an old-school joke in JavaScript, but remastered. Here's the original one:

```
"foo" + +"bar"; // -> 'fooNaN'
```

true is not equal ![], but not equal [] too

Array is not equal true, but not Array is not equal true too; Array is equal false, not Array is equal false too:

```
true == []; // -> false

true == ![]; // -> false

false == []; // -> true

false == ![]; // -> true
```

Explanation:

```
true == []; // -> false

true == ![]; // -> false

// According to the specification

true == []; // -> false

toNumber(true); // -> 1

toNumber([]); // -> 0

1 == 0; // -> false

true == ![]; // -> false

![]; // -> false

true == false; // -> false

false == []; // -> true
false == ![]; // -> true

// According to the specification

false == []; // -> true

toNumber(false); // -> 0

toNumber([]); // -> 0

0 == 0; // -> true

false == ![]; // -> true

![]; // -> false

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

[] is equal ![]

```
[] == ![]; // -> true

+[] == +![];

0 == +false;

0 == 0;

true;
```

https://github.com/denysdovhan/wtfjs