Type coercion is the process of converting value from one type to another (such as string to number, object to boolean, and so on). Any type, be it primitive or an object, is a valid subject for type coercion. To recall, primitives are: number, string, boolean, null, undefined + Symbol (added in ES6).

As an example of type coercion in practice, look at the JavaScript Comparison Table, which shows how the loose equality `==` operator behaves for different `a` and `b` types. This matrix looks scary due to implicit type coercion that `==` operator does, and it's hardly possible to remember all those combinations. And you don't have to do that — just learn the underlying type coercion principles.

This article goes in-depth on how type coercion works in JavaScript, and will arm you with the essential knowledge, so you can feel confident explaining what following expressions calculate to. By the end of the article I'll show answers and explain them.

```
true + false
12 / "6"
"number" + 15 + 3
15 + 3 + "number"
[1] > null
"foo" + + "bar"
'true' == true
false == 'false'
null == ''
!!"false" == !!"true"
[‘x’] == ‘x’
[] + null + 1
[1,2,3] == [1,2,3]
{}+[]+{}+[1]
!+[]+[]+![]
new Date(0) - 0
new Date(0) + 0
```

Yes, this list is full of pretty silly things you can do as a developer. In 90% of use cases it's better to avoid implicit type coercion. Consider this list as a learning exercise to test your knowledge on how type coercion works. If you're bored, you can find more examples on wtfjs.com.

By the way, sometimes you might face such questions on the interview for a JavaScript developer position. So, keep reading ?

Implicit vs. explicit coercion

Type coercion can be explicit and implicit.

When a developer expresses the intention to convert between types by writing the appropriate code, like `Number(value)`, it's called **explicit type coercion** (or type casting).

Since JavaScript is a weakly-typed language, values can also be converted between different types automatically, and it is called **implicit type coercion**. It usually happens when you apply operators to values of different types, like `1 == null`, `2/'5'`, `null + new Date()`, or it can be triggered by the surrounding context, like with `if (value) {…}`, where `value` is coerced to boolean.

One operator that does not trigger implicit type coercion is `===`, which is called the strict equality operator. The loose equality operator `==` on the other hand does both comparison and type coercion if needed.

Implicit type coercion is a double edge sword: it's a great source of frustration and defects, but also a useful mechanism that allows us to write less code without losing the readability.

Three types of conversion

The first rule to know is there are only three types of conversion in JavaScript:

- to string
- to boolean
- to number

Secondly, conversion logic for primitives and objects works differently, but both primitives and objects can only be converted in those three ways.

Let's start with primitives first.

String conversion

To explicitly convert values to a string apply the `String()` function.
Implicit coercion is triggered by the binary `+` operator, when any operand is a string:

```
String(123) // explicit
123 + ''    // implicit
```

All primitive values are converted to strings naturally as you might expect:

```
String(123)                 // '123'
String(-12.3)               // '-12.3'
String(null)                // 'null'
String(undefined)           // 'undefined'
String(true)                // 'true'
String(false)               // 'false'
```

Symbol conversion is a bit tricky, because it can only be converted explicitly, but not implicitly. Read more on `Symbol` coercion rules.

```
String(Symbol('my symbol'))   // 'Symbol(my symbol)'
'' + Symbol('my symbol')      // TypeError is thrown
```

Boolean conversion

To explicitly convert a value to a boolean apply the `Boolean()` function.
Implicit conversion happens in logical context, or is triggered by logical operators (`||` `&&` `!`).

```
Boolean(2)          // explicit
if (2) { ... }      // implicit due to logical context
!!2                 // implicit due to logical operator
2 || 'hello'        // implicit due to logical operator
```

Note: Logical operators such as `||` and `&&` do boolean conversions internally, but actually return the value of original operands, even if they are not boolean.

```
// returns number 123, instead of returning true
// 'hello' and 123 are still coerced to boolean internally to ca
```

```
let x = 'hello' && 123;   // x === 123
```

As soon as there are only 2 possible results of boolean conversion: `true` or `false`, it's just easier to remember the list of falsy values.

```
Boolean('')          // false
Boolean(0)           // false
Boolean(-0)          // false
Boolean(NaN)         // false
Boolean(null)        // false
Boolean(undefined)   // false
Boolean(false)       // false
```

Any value that is not in the list is converted to `true`, including object, function, `Array`, `Date`, user-defined type, and so on. Symbols are truthy values. Empty object and arrays are truthy values as well:

```
Boolean({})             // true
Boolean([])             // true
Boolean(Symbol())       // true
!!Symbol()              // true
Boolean(function() {})  // true
```

Numeric conversion

For an explicit conversion just apply the `Number()` function, same as you did with `Boolean()` and `String()`.

Implicit conversion is tricky, because it's triggered in more cases:

- comparison operators (`>`, `<`, `<=`, `>=`)
- bitwise operators (`|` `&` `^` `~`)
- arithmetic operators (`-` `+` `*` `/` `%`). Note, that binary `+` does not trigger numeric conversion, when any operand is a string.
- unary `+` operator
- loose equality operator `==` (incl. `!=`).
  Note that `==` does not trigger numeric conversion when both operands are strings.

```
Number('123')   // explicit
+'123'          // implicit
123 != '456'    // implicit
4 > '5'         // implicit
5/null          // implicit
true | 0        // implicit
```

Here is how primitive values are converted to numbers:

```
Number(null)              // 0
Number(undefined)         // NaN
Number(true)              // 1
Number(false)             // 0
Number(" 12 ")            // 12
Number("-12.34")          // -12.34
Number("\n")              // 0
Number(" 12s ")           // NaN
Number(123)               // 123
```

When converting a string to a number, the engine first trims leading and trailing whitespace, `\n`, `\t` characters, returning `NaN` if the trimmed string does not represent a valid number. If string is empty, it returns `0`.

`null` and `undefined` are handled differently: `null` becomes `0`, whereas `undefined` becomes `NaN`.

Symbols cannot be converted to a number neither explicitly nor implicitly. Moreover, `TypeError` is thrown, instead of silently converting to `NaN`, like it happens for `undefined`. See more on Symbol conversion rules on MDN.

```
Number(Symbol('my symbol'))    // TypeError is thrown
+Symbol('123')                 // TypeError is thrown
```

There are two **special rules** to remember:

1. When applying `==` to `null` or `undefined`, numeric conversion does not happen. `null` equals only to `null` or `undefined`, and does not equal to anything else.

```
null == 0               // false, null is not converted to 0
null == null            // true
undefined == undefined  // true
null == undefined       // true
```

2. NaN does not equal to anything even itself:

```
if (value !== value) { console.log("we're dealing with NaN here")
```

Type coercion for objects

So far, we've looked at type coercion for primitive values. That's not very exciting.

When it comes to objects and engine encounters expression like `[1] + [2,3]`, first it needs to convert an object to a primitive value, which is then converted to the final type. And still there are only three types of conversion: numeric, string and boolean.

The simplest case is boolean conversion: any non-primitive value is always
coerced to `true`, no matter if an object or an array is empty or not.

Objects are converted to primitives via the internal `[[ToPrimitive]]` method, which is responsible for both numeric and string conversion.

Here is a pseudo implementation of `[[ToPrimitive]]` method:

`[[ToPrimitive]]` is passed with an input value and preferred type of conversion: `Number` or `String`. `preferredType` is optional.

Both numeric and string conversion make use of two methods of the input object: `valueOf` and `toString`. Both methods are declared on `Object.prototype` and thus available for any derived types, such as `Date`, `Array`, etc.

In general the algorithm is as follows:

1. If input is already a primitive, do nothing and return it.
2. Call `input.toString()`, if the result is primitive, return it.
3. Call `input.valueOf()`, if the result is primitive, return it.
4. If neither `input.toString()` nor `input.valueOf()` yields primitive, throw `TypeError`.

Numeric conversion first calls `valueOf` (3) with a fallback to `toString` (2). String conversion does the opposite: `toString` (2) followed by `valueOf` (3).

Most built-in types do not have `valueOf`, or have `valueOf` returning `this` object itself, so it's ignored because it's not a primitive. That's why numeric and string conversion might work the same — both end up calling `toString()`.

Different operators can trigger either numeric or string conversion with a help of `preferredType` parameter. But there are two exceptions: loose equality `==` and binary `+` operators trigger default conversion modes (`preferredType` is not specified, or equals to `default`). In this case, most built-in types assume numeric conversion as a default, except `Date` that does string conversion.

Here is an example of `Date` conversion behavior:

You can override the default `toString()` and `valueOf()` methods to hook into object-to-primitive conversion logic.

Notice how `obj + ''` returns `'101'` as a string. `+` operator triggers a default conversion mode, and as said before `Object` assumes numeric conversion as a default, thus using the `valueOf()` method first instead of `toString()`.

ES6 Symbol.toPrimitive method

In ES5 you can hook into object-to-primitive conversion logic by overriding `toString` and `valueOf` methods.

In ES6 you can go farther and completely replace internal `[[ToPrimitive]]` routine by implementing the `[Symbol.toPrimtive]` method on an object.

Examples

Armed with the theory, now let's get back to our examples:

```
true + false             // 1
12 / "6"                 // 2
"number" + 15 + 3        // 'number153'
15 + 3 + "number"        // '18number'
[1] > null               // true
"foo" + + "bar"          // 'fooNaN'
'true' == true           // false
false == 'false'         // false
null == ''               // false
!!"false" == !!"true"    // true
['x'] == 'x'             // true
[] + null + 1            // 'null1'
[1,2,3] == [1,2,3]       // false
{}+[]+{}+[1]             // '0[object Object]1'
!+[]+[]+![]              // 'truefalse'
new Date(0) - 0          // 0
new Date(0) + 0          // 'Thu Jan 01 1970 02:00:00(EET)0'
```

Below you can find explanation for each the expression.

Binary `+` operator triggers numeric conversion for `true` and `false`

```
true + false
==> 1 + 0
==> 1
```

Arithmetic division operator `/` triggers numeric conversion for string `'6'`:

```
12 / '6'
==> 12 / 6
==>> 2
```

Operator `+` has left-to-right associativity, so expression `"number" + 15` runs first. Since one operand is a string, `+` operator triggers string conversion for the number `15`. On the second step expression `"number15" + 3` is evaluated similarly.

```
“number” + 15 + 3
==> "number15" + 3
==> "number153"
```

Expression `15 + 3` is evaluated first. No need for coercion at all, since both operands are numbers. On the second step, expression `18 + 'number'` is evaluated, and since one operand is a string, it triggers a string conversion.

```
15 + 3 + "number"
==> 18 + "number"
==> "18number"
```

Comparison operator `&` gt; triggers numeric conversion f `or` [1] a `nd` `n` ull .

```
[1] > null
==> '1' > 0
==> 1 > 0
==> true
```

Unary `+` operator has higher precedence over binary `+` operator. So `+'bar'` expression evaluates first. Unary plus triggers numeric conversion for string `'bar'`. Since the string does not represent a valid number, the result is `NaN`. On the second step, expression `'foo' + NaN` is evaluated.

```
"foo" + + "bar"
==> "foo" + (+"bar")
==> "foo" + NaN
==> "fooNaN"
```

`==` operator triggers numeric conversion, string `'true'` is converted to NaN, boolean `true` is converted to 1.

```
'true' == true
==> NaN == 1
==> false

false == 'false'
==> 0 == NaN
==> false
```

`==` usually triggers numeric conversion, but it’s not the case with `null`. `null` equals to `null` or `undefined` only, and does not equal to anything else.

```
null == ''
==> false
```

`!!` operator converts both `'true'` and `'false'` strings to boolean `true`, since they are non-empty strings. Then, `==` just checks

equality of two boolean `true's` without any coercion.

```
!!"false" == !!"true"
==> true == true
==> true
```

`==` operator triggers a numeric conversion for an array. Array's `valueOf()` method returns the array itself, and is ignored because it's not a primitive. Array's `toString()` converts `['x']` to just `'x'` string.

```
['x'] == 'x'
==> 'x' == 'x'
==>  true
```

`+` operator triggers numeric conversion for `[]`. Array's `valueOf()` method is ignored, because it returns array itself, which is non-primitive. Array's `toString` returns an empty string.

On the the second step expression `'' + null + 1` is evaluated.

```
[] + null + 1
==>  '' + null + 1
==>  'null' + 1
==> 'null1'
```

Logical `||` and `&&` operators coerce operands to boolean, but return original operands (not booleans). `0` is falsy, whereas `'0'` is truthy, because it's a non-empty string. `{}` empty object is truthy as well.

```
0 || "0" && {}
==>  (0 || "0") && {}
==> (false || true) && true  // internally
==> "0" && {}
==> true && true             // internally
==> {}
```

No coercion is needed because both operands have same type. Since `==` checks for object identity (and not for object equality) and the two arrays are two different instances, the result is `false`.

```
[1,2,3] == [1,2,3]
==>  false
```

All operands are non-primitive values, so `+` starts with the leftmost triggering numeric conversion. Both `Object's` and `Array's` `valueOf` method returns the object itself, so it's ignored. `toString()` is used as a fallback. The trick here is that first `{}` is not considered as an object literal, but rather as a block declaration statement, so it's ignored. Evaluation starts with next `+[]` expression, which is converted to an empty string via `toString()` method and then to `0`.

```
{}+[]+{}+[1]
==> +[]+{}+[1]
==> 0 + {} + [1]
==> 0 + '[object Object]' + [1]
==> '0[object Object]' + [1]
==> '0[object Object]' + '1'
==> '0[object Object]1'
```

This one is better explained step by step according to operator precedence.

```
!+[]+[]+![]
==> (!+[]) + [] + (![])
==> !0 + [] + false
==> true + [] + false
==> true + '' + false
==> 'truefalse'
```

`-` operator triggers numeric conversion for `Date`. `Date.valueOf()` returns number of milliseconds since Unix epoch.

```
new Date(0) - 0
==> 0 - 0
==> 0
```

`+` operator triggers default conversion. Date assumes string conversion as a default one, so `toString()` method is used, rather than `valueOf()`.

```
new Date(0) + 0
==> 'Thu Jan 01 1970 02:00:00 GMT+0200 (EET)' + 0
==> 'Thu Jan 01 1970 02:00:00 GMT+0200 (EET)0'
```
