💯

Coercion

Created	@May 13, 2021 11:50 PM
Class	JS Advanced
Type	Unacademy
Materials	https://262.ecma-international.org/5.1/
Reviewed	☐

What is coercion ?

There are some abstract operations in JS which helps us to do type conversion and in JS type conversion is also called as Coercion.

> # 7 Abstract Operations
>
> These operations are not a part of the ECMAScript language; they are defined here to solely to aid the specification of the semantics of the ECMAScript language. Other, more specialized abstract operations are defined throughout this specification.
>
> ## 7.1 Type Conversion
>
> The ECMAScript language implicitly performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. The conversion abstract operations are polymorphic; they can accept a value of any ECMAScript language type. But no other specification types are used with these operations.

ToPrimitive - the first abstract operation for type conversion

- ToPrimitive takes an *input* argument and an optional *PreferredType*. This operation converts the input into a non-object type. If an object is capable of converting to more than one primitive type the function may use the *PreferredType* argument to resolve it.

> As we said they are abstract operations that means they are conceptual operations and every time when we require them they will be invoked.

- The *PreferredType* is also used when we do an operation and the operations asks for a specific type then this argument helps to tell the abstract function to try to convert the object into this *PreferredType* if possible as it doesn't guarantees a conversion every time.
- Generally the *PreferredType* that operations can send are Number or String

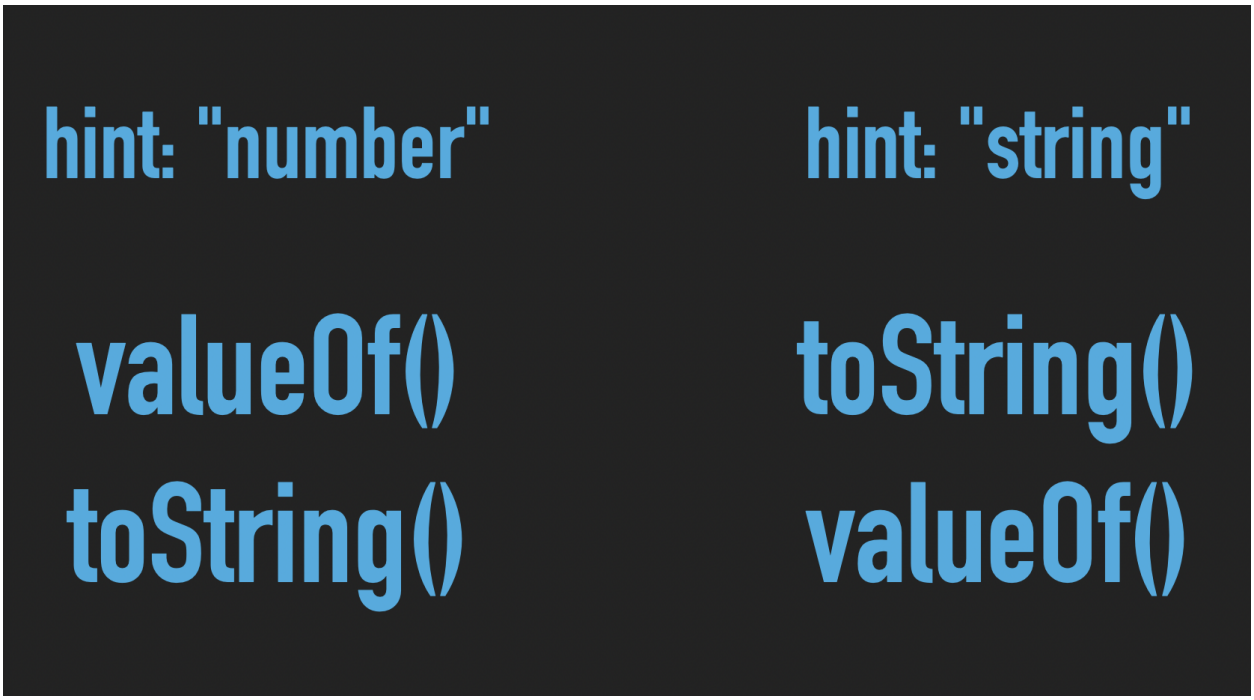
- Also generally algorithms in JS are inherently recursive. If let's say result of ToPrimitive is still an object then it will get recursively invoked again in order to try to convert this object into a primitive.
- If the *hint/PreferredType* is number then the ToPrimitive function will first call `valueOf()` function to get a number if this doesn't gives a primitive then `toString()` function is called. The order reverses to first `toString()` and then `valueOf()` in case of string as the *hint.*

ECMAScript® 2018 Language Specification

https://262.ecma-international.org/9.0/#sec-abstract-operations

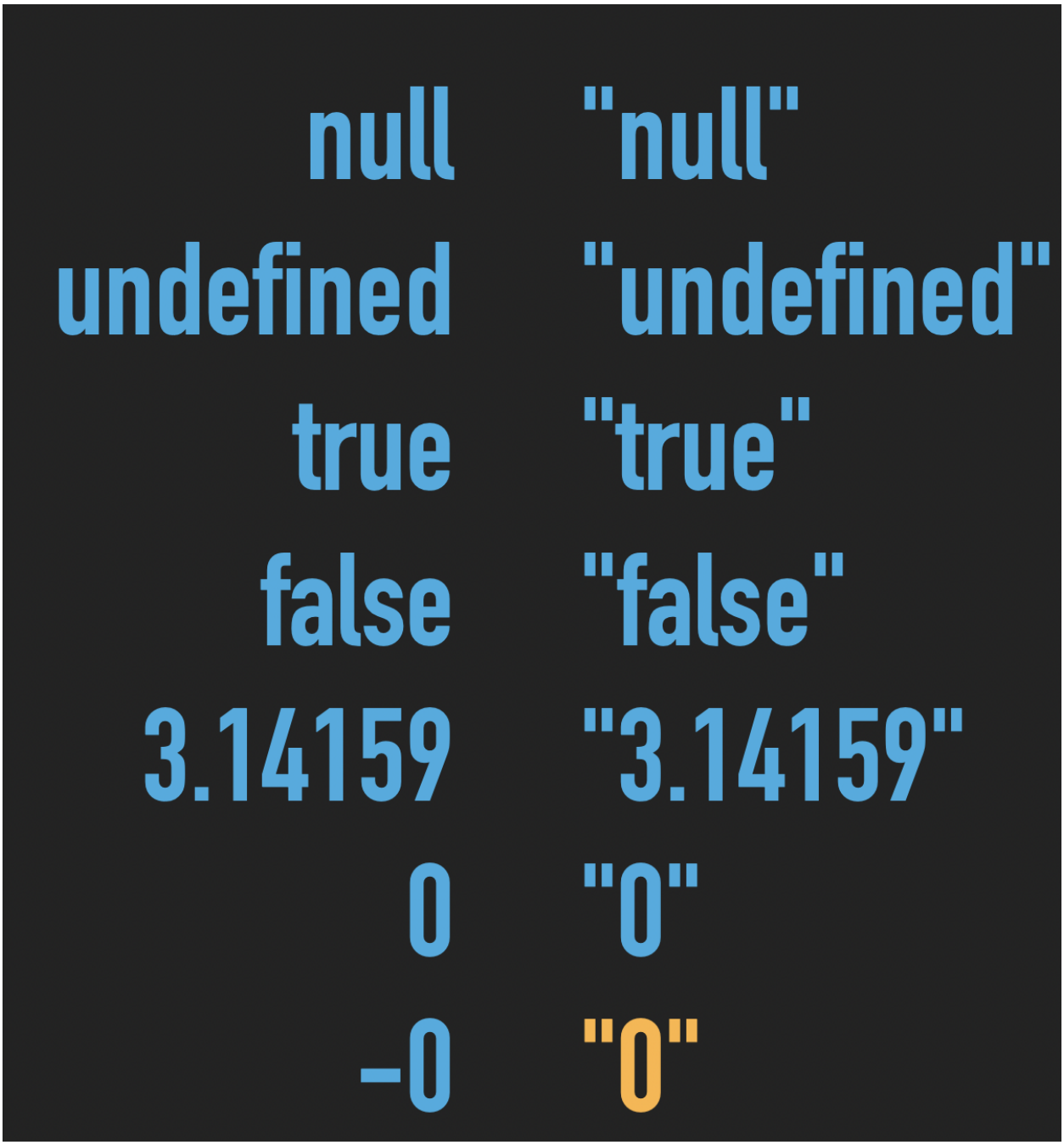
Table 10 — ToPrimitive Conversions

Input Type	Result
Undefined	The result equals the *input* argument (no conversion).
Null	The result equals the *input* argument (no conversion).
Boolean	The result equals the *input* argument (no conversion).
Number	The result equals the *input* argument (no conversion).
String	The result equals the *input* argument (no conversion).
Object	Return a default value for the Object. The default value of an object is retrieved by calling the [[DefaultValue]] internal method of the object, passing the optional hint *PreferredType*. The behaviour of the [[DefaultValue]] internal method is defined by this specification for all native ECMAScript objects in 8.12.8.



ToString

As the name suggests it converts an object to a string.



- If we call toString() on an object it invokes ToPrimitive with *hint* as string i.e it will first call toString() first and then valueOf()
- Built in toString() on arrays escapes the brackets while converting but they are present in Objects. Why ?? Because it's JS's choice.
- An array of null and undefined also behaves abruptly, because they are not treated the same way null and undefined are separately treated with `toString()` .

- For an object toString() returns a bizarre `[object Object]` they put a lower case `object` string and then a string tag which can be overridden. We can change this `Object` to any custom string of our choice.

```
x = {toString() {return "X"} } // overriding toString()
```

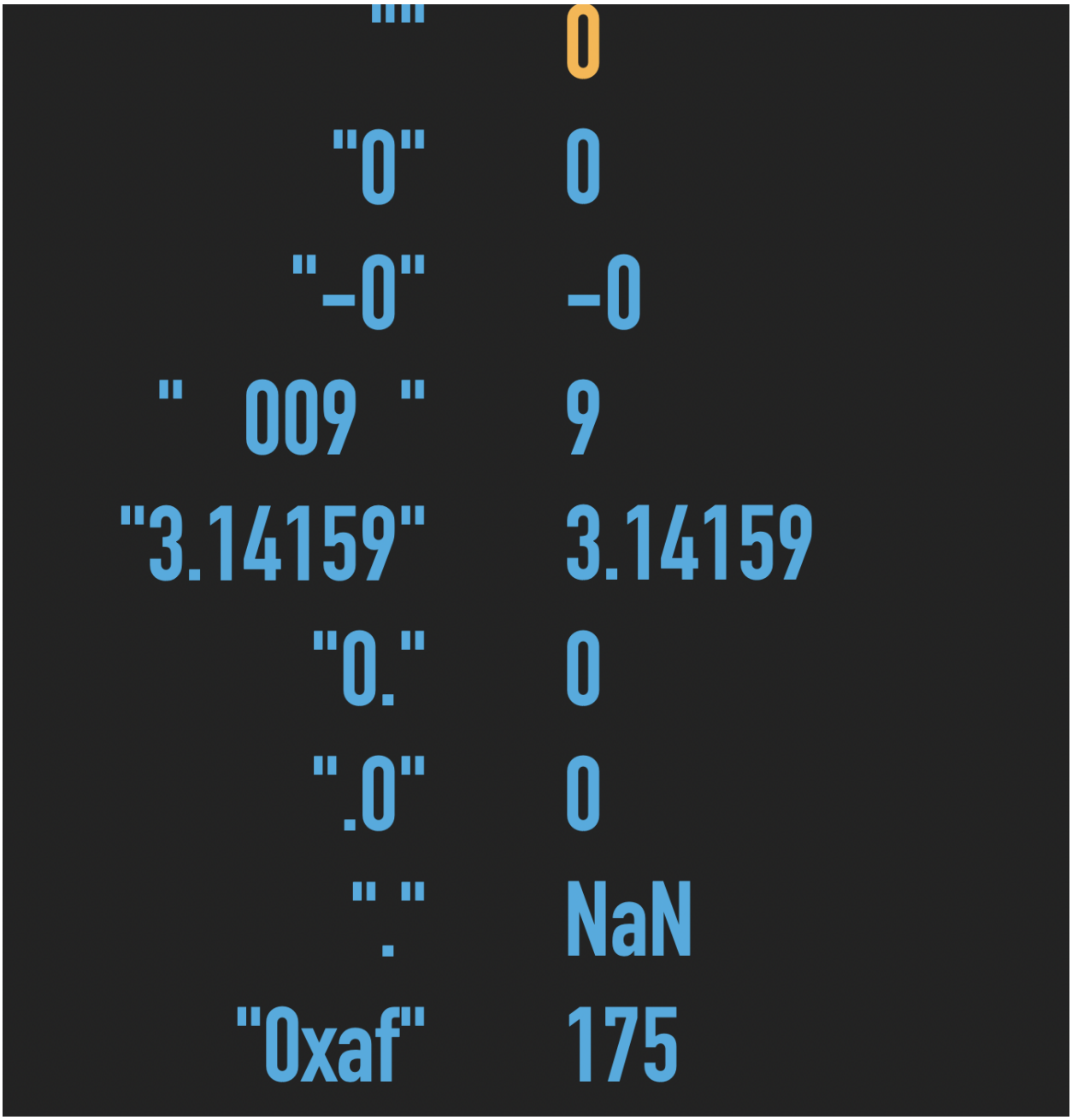
- Why on earth they left square brackets for arrays and brought them back for objects ???
- In case of array of null and undefined the null and defined are not printed but there are spaces available for them. Again a corner case.

ECMAScript® 2018 Language Specification

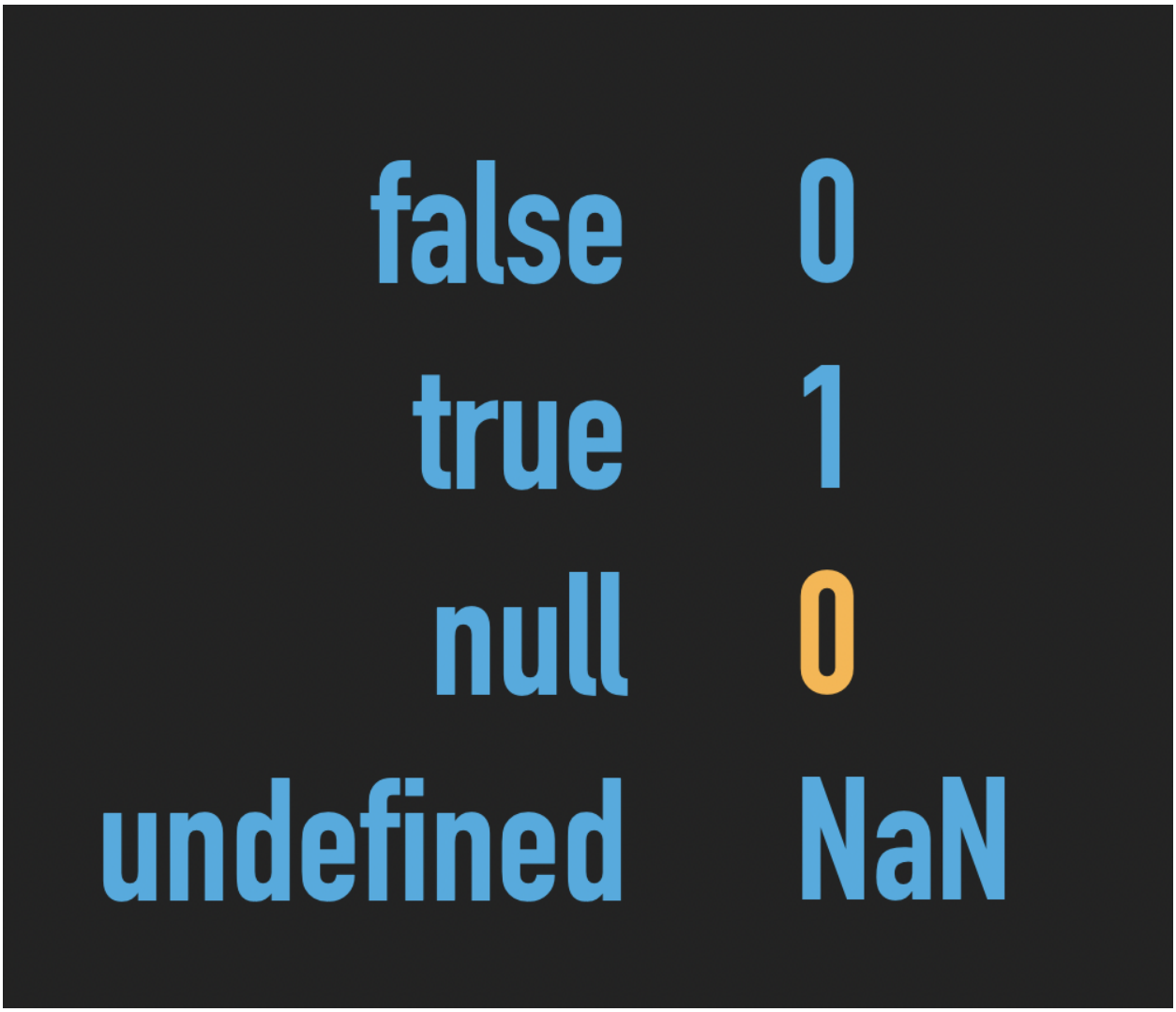
https://262.ecma-international.org/9.0/#sec-tostring

ToNumber

When in an numeric operation we don't have a number toNumber() is called.



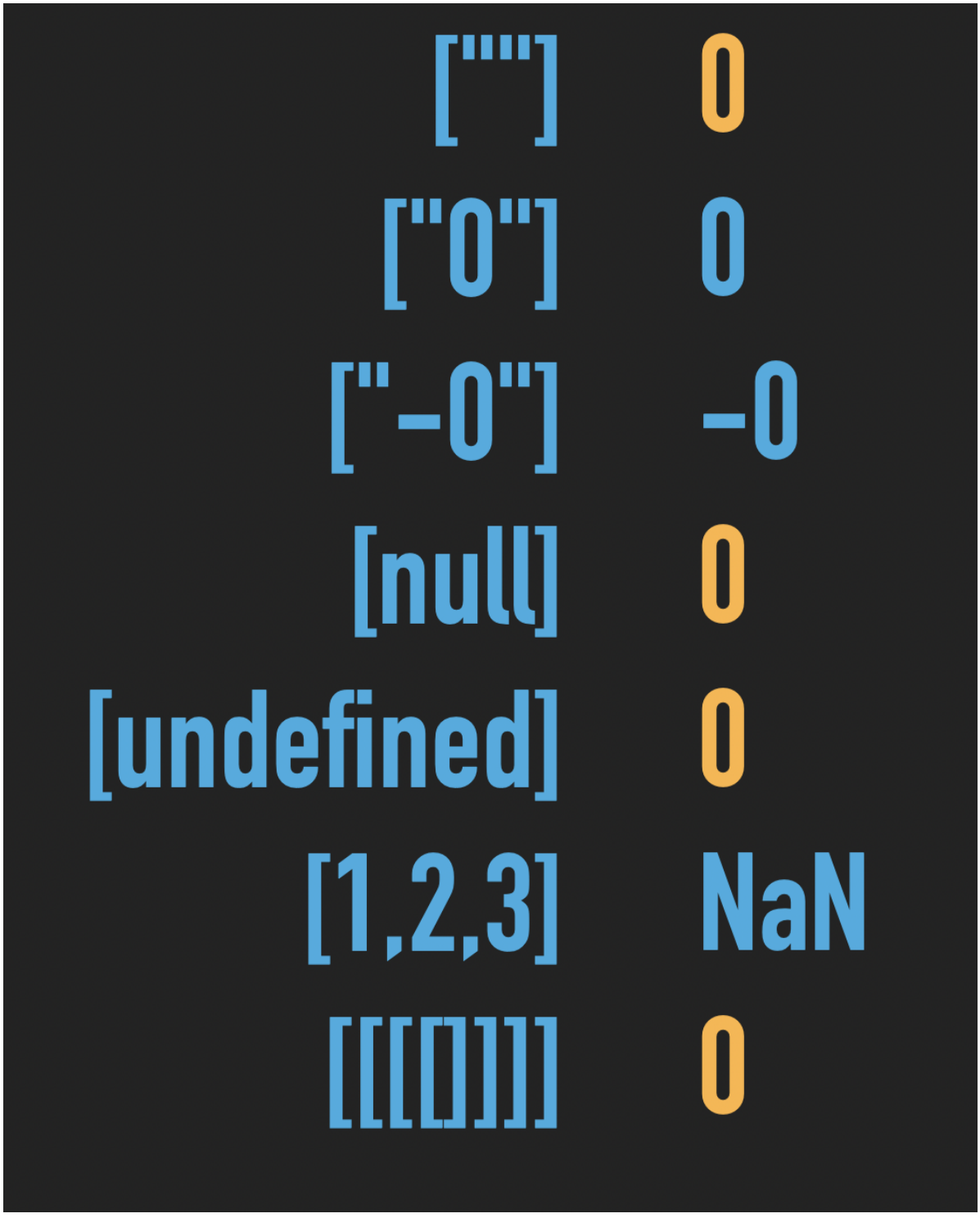
- The main evil here is "" → 0, because they could have returned NaN here.



- Here undefined is NaN but null is not not NaN.
- When we use ToNumber on a non primitive, i.e. object then ToPrimitive is called with Number as *hint*
- For any array and object by default valueOf function returns original array or object only.

  ```
  valueOf{ return this; } // for [] and {}
  ```

  The effect of this is that the valueOf return value is ignored and goes to toSrting directly for [] and {}.



- Here null and undefined are first converted to 0 which in turn converts into empty string

- For an object it returns NaN. It can be changed if you override the valueOf function

ECMAScript® 2018 Language Specification

https://262.ecma-international.org/9.0/#sec-tonumber

ToBoolean

The toBoolean function converts the given type to boolean wherever expected. But in order to do this it doesn't invoke the toPrimitive function rather than it does a lookup on a table of **falsy** values. The values which are not in the table are all true.

ECMAScript® 2018 Language Specification

https://262.ecma-international.org/9.0/#sec-toboolean

Now what the hell is coercion ??

People try to avoid coercion and say we will follow === operator and do the things but at a lot of places they end up doing implicit coercion.

```
var numStudents = 16;

console.log(
    `There are ${numStudents} students.`
);
// "There are 16 students."
```

In the above example the template literal string or string interpolation is using coercion.

```
var msg1 = "There are ";
var numStudents = 16;
var msg2 = " students.";
console.log(msg1 + numStudents + msg2);
// "There are 16 students."
```

In the above example we are adding a number to a string, then why it converts it to a string ? Is it only coercion ? No... This is an example of operator overloading.

The + operator says that while adding anything and one of the operand is a string then do string concatenation.

ECMAScript® 2018 Language Specification

https://262.ecma-international.org/9.0/#sec-addition-operator-plus

12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).
6. Let *rprim* be ? ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   a. Let *lstr* be ? ToString(*lprim*).
   b. Let *rstr* be ? ToString(*rprim*).
   c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? ToNumber(*lprim*).
9. Let *rnum* be ? ToNumber(*rprim*).
10. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below 12.8.5.

This is again internally using coercion.

Now we can also do explicit coercion in a lot of things.

```
1 var numStudents = 16;
2
3 console.log(
4     `There are ${[numStudents].join("")} students.`
5 );
6 // "There are 16 students."
```

In this example we have only one value in the array and there is nothing to be concatenated with to work with join still the join operation says that it will first convert everything to a string and then join them. Not a very useful and best practice way but it can be seen in a lot of places written unknowingly.

The best way to explicitly do coercion for String is to use `String()` as it explicitly converts things. We can also use `.toString()` but the issues is why does a primitive value has got a function associated with it ?????

Anything apart from String?? Yes what about numbers ..

In web applications we take input from user and that comes as a string and we end up doing coercion without even realising.

```
function addAStudent(numStudents) {
    return numStudents + 1;
}

addAStudent(studentsInputElem.value);
// "161"  OOPS!
```

Here when we send the string value to add with one leads to doing concatenation of strings.

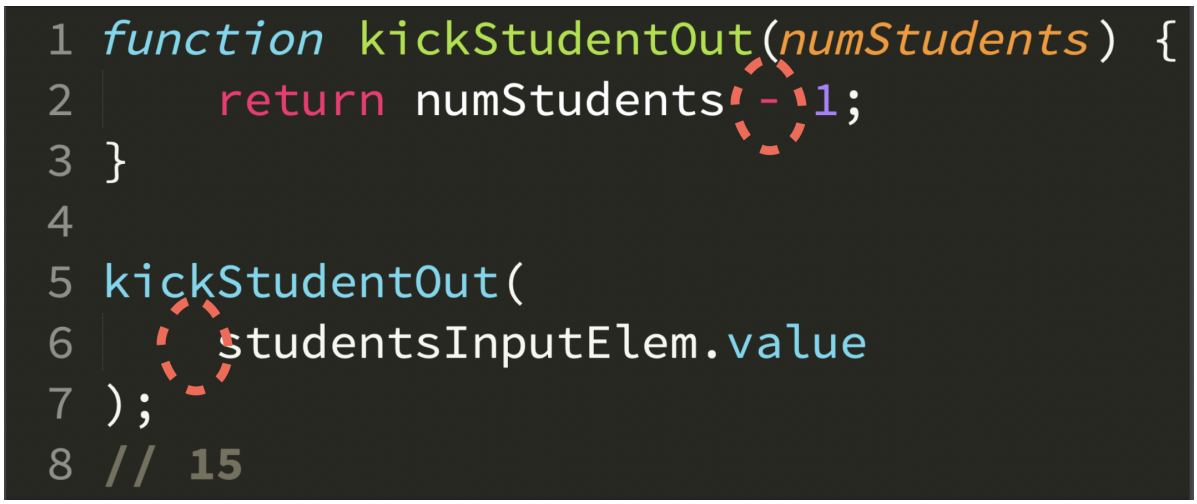
```
function addAStudent(numStudents) {
    return numStudents + 1;
}

addAStudent(
    +studentsInputElem.value
);
// 17
```

Here we are explicitly converting string to number using the unary plus operator. It invokes `toNumber()` abstract operations. So here what if we get an empty string what

it will convert itself into ? → 0

If you want to be more specific in terms of syntax and semantics use `Number()` function. Also we don't have a toNumber available in JS.



A '`-`' operator is only defined for Numbers so it never does coercion to strings.

What about booleans ?

Never mess around coercion with booleans. One should never write `if` statements that use non booleans in the `if` clause. Example:

- Checking a string if it is empty or not inside if. In the below example here we expect an empty string to be false but what about a string with only spaces ? It will be true.

```
if(studentInputElem.value) {
    numStudents = Number(studentInputElem.value);
}
```

- Numeric conversion of a number to zero or non-zero to check if it is false or true. But a corner case here can be a `NaN` ?

```
while(numStudents.length) {
    enrollStudent(newStudents.pop())
}
```

What you should prefer for above cases ?

```
if(!!studentInputElem.value) { // explicit boolean conversion
    numStudents = Number(studentInputElem.value);
}
// or we can use Boolean()
```

```
while(numStudents.length > 0) {
    enrollStudent(newStudents.pop())
}
```

Here explicitness is necessarily better.

```
Boolean("") // false
Boolean(" \n\t") // true Ahh!!
Boolean({}) // true
```

How do we access properties like length() and toString() with primitive types ???

This concept is called as **Boxing**. This happens when something is not an object but you're trying to use it as an object, JS will help you with this and convert it to an object. In other languages they might throw an exception but JS will implicitly coerce it for you.

It is a coercion of primitive to object. This gives the notion of everything is an object, actually everything is not an object but when required JS will make them.

Corner cases of coercion

People use corner cases of weapon against JS when we know every language has got some corner cases, more or less but every language has it.

```
Number( "" );                  // 0       OOPS!
Number( "  \t\n" );            // 0       OOPS!
Number( null );                // 0       OOPS!
Number( undefined );           // NaN
Number( [] );                  // 0       OOPS!
Number( [1,2,3] );             // NaN
Number( [null] );              // 0       OOPS!
Number( [undefined] );         // 0       OOPS!
Number( {} );                  // NaN

String( -0 );                  // "0"     OOPS!
String( null );                // "null"
String( undefined );           // "undefined"
String( [null] );              // ""      OOPS!
String( [undefined] );         // ""      OOPS!

Boolean( new Boolean(false) ); // true    OOPS!
```

The Root Of All (Coercion) Evil

```
studentsInput.value = "";

// ..

Number(studentsInput.value);      // 0
```

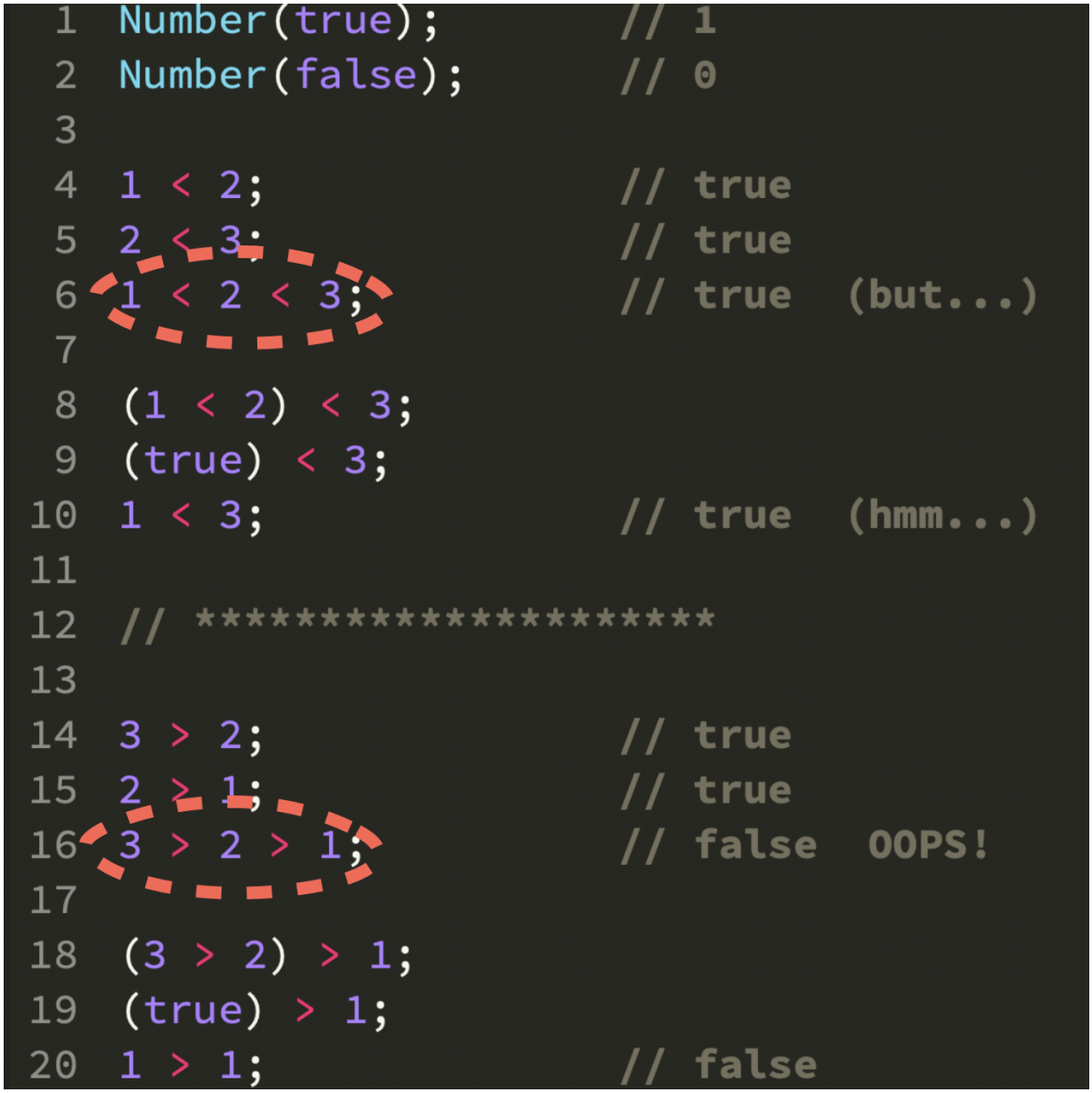
```
studentsInput.value = "  \t\n";

// ..

Number(studentsInput.value);      // 0
```

The `ToNumber()` operation strips all the leading and trailing whitespaced of the string before coercion that's why a string of spaces becomes empty and gets converted to 0.

> ECMAScript® 2018 Language Specification
>
> https://262.ecma-international.org/9.0/#sec-tonumber



The above example is to demonstrate why true should not evaluate to 1 and false to 0.

In `1 < 2 < 3` first we resolve `1 < 2` which evaluates to true then true < 3 coerces true to 1 and makes it `1 < 3` which is true, but in the second example when `(3 > 2)` is resolved to true and true is resolved to 1 then `1 > 1` becomes false.

Assignment

Working With Coercion

In this exercise, you will define some validation functions that check user inputs (such as from DOM elements). You'll need to properly handle the coercions of the various value types.

Instructions

1. Define an `isValidName(..)` validator that takes one parameter, `name`. The validator returns `true` if all the following match the parameter (`false` otherwise):
    - must be a string
    - must be non-empty
    - must contain non-whitespace of at least 3 characters
2. Define an `hoursAttended(..)` validator that takes two parameters, `attended` and `length`. The validator returns `true` if all the following match the two parameters (`false` otherwise):
    - either parameter may only be a string or number
    - both parameters should be treated as numbers
    - both numbers must be 0 or higher
    - both numbers must be whole numbers
    - `attended` must be less than or equal to `length`

Solution:

```
function isValidName(name) {
  if (
    typeof name == "string" &&
    name.trim().length >= 3
  ) {
    return true;
  }

  return false;
}

function hoursAttended(attended,length) {
  if (
    typeof attended == "string" &&
    attended.trim() != ""
  ) {
    attended = Number(attended);
  }
  if (
    typeof length == "string" &&
    length.trim() != ""
  ) {
    length = Number(length);
  }
  if (
    typeof attended == "number" &&
    typeof length == "number" &&
    attended <= length &&
    attended >= 0 &&
    length >= 0 &&
    Number.isInteger(attended) &&
    Number.isInteger(length)
  ) {
    return true;
  }

  return false;
}


// tests:
console.log(isValidName("Frank") === true);
console.log(hoursAttended(6,10) === true);
console.log(hoursAttended(6,"10") === true);
console.log(hoursAttended("6",10) === true);
console.log(hoursAttended("6","10") === true);

console.log(isValidName(false) === false);
console.log(isValidName(null) === false);
```

```
console.log(isValidName(undefined) === false);
console.log(isValidName("") === false);
console.log(isValidName("  \t\n") === false);
console.log(isValidName("X") === false);
console.log(hoursAttended("",6) === false);
console.log(hoursAttended(6,"") === false);
console.log(hoursAttended("","") === false);
console.log(hoursAttended("foo",6) === false);
console.log(hoursAttended(6,"foo") === false);
console.log(hoursAttended("foo","bar") === false);
console.log(hoursAttended(null,null) === false);
console.log(hoursAttended(null,undefined) === false);
console.log(hoursAttended(undefined,null) === false);
console.log(hoursAttended(undefined,undefined) === false);
console.log(hoursAttended(false,false) === false);
console.log(hoursAttended(false,true) === false);
console.log(hoursAttended(true,false) === false);
console.log(hoursAttended(true,true) === false);
console.log(hoursAttended(10,6) === false);
console.log(hoursAttended(10,"6") === false);
console.log(hoursAttended("10",6) === false);
console.log(hoursAttended("10","6") === false);
console.log(hoursAttended(6,10.1) === false);
console.log(hoursAttended(6.1,10) === false);
console.log(hoursAttended(6,"10.1") === false);
console.log(hoursAttended("6.1",10) === false);
console.log(hoursAttended("6.1","10.1") === false);
```

JavaScript type coercion explained

by Alexey Samoshkin JavaScript type coercion explainedKnow your enginesWeird things can happen in JavaScript[Edit 2/5/2018]: This post is now available in Russian. Claps to Serj

https://www.freecodecamp.org/news/js-type-coercion-explained-27ba3d9a2839/