

Author: Aditya Sharma

## JavaScript (Part 2)

1. **Dynamic typing:** We never tell the code what type of data a variable holds. The engine figures this by itself.

### Static Typing

```
bool isNew = 'hello'; // an error
```

### Dynamic Typing

```
var isNew = true; // no errors  
isNew = 'yup!';  
isNew = 1;
```

2. **Primitive types:** The type of data that represents a single value. It is not an object.

1. Undefined (lack of existence)
2. Null (lack of existence)
3. Boolean
4. Number (a floating-point number)
5. String
6. Symbol (ES6 only)

3. **Operators:** Special function written differently/syntactically. Generally take two parameters and give one result.

Ex:

Addition operator(+), NOT, NOR, XOR, greater than, less than etc

In phase operator: In  $3 + 4$  the  $+$  symbol is the in-phase operator as it sits between the two numbers. JavaScript has this built-in function like :

```
function +(a,b){  
  sum=a+b;  
  console.log(sum);  
}
```

Operator precedence: Which operator is called first.

Associativity: Left to right or right to left

```
1 var a = 2, b = 3, c = 4;  
2  
3 a = b = c;  
4  
5 console.log(a);  
6 console.log(b);  
7 console.log(c);
```

Console prints out 4, three times on the screen because the associativity of 'assignment' operator is from right to left (refer precedence table using the link below).

[JS precedence table](#)

#### 4. Coercion:

Converting a value from one form to another. Happens in JS because it is dynamically typed.

```
1 var a = 1 + '2';  
2 console.log(a);  
3
```

Output: 12

Reason: Due to coercion type conversion of 1 takes place from a number to a string and then string concatenation.

#### 5. Comparison operator:

Eg:

`console.log(1<2<3)` gives true on the console but  
`console.log(3<2<1)` also gives true.

Reason:

Associativity of less than function is from left to right.

In the case of `console.log(3<2<1)` first `(3<2)` is evaluated which is false. So, we are left with `console.log(false<1)`. Since these two values do not belong to the same category of primitive types, coercion takes place i.e. JS engine tries to convert false to a number.

Numbers corresponding to a string can be found out using following command in the console:

`Number(false)=0`, and `0<1` is true that's why console printed out true.

Thus due to type conversion, `false == 0` is true. However, null and undefined do not behave this way i.e. `null == 0` is false.

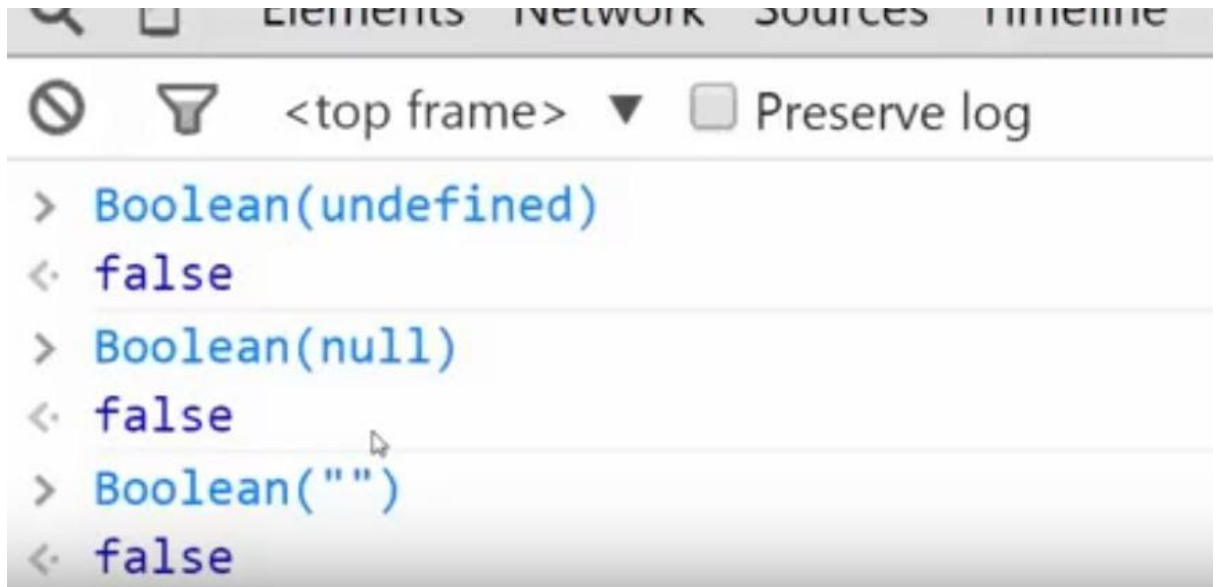
`"" == 0` is true and `"" == false` is also true.

That's why strict equality is used (===). This avoids type coercion. Similarly, !== is not equal and in this case, we can anticipate coercion. However, using !== (strict inequality) this does not happen.

See the comparison table while comparing different types:

[Comparison and sameness](#)

## 6. Boolean and Existence



A screenshot of a web browser's developer console. The top bar shows tabs for 'Elements', 'Network', 'Sources', and 'Timeline'. Below the tabs, there are icons for a disabled state, a filter, and a dropdown menu set to '<top frame>'. To the right is a checkbox labeled 'Preserve log'. The console log shows three entries: a prompt '>' followed by 'Boolean(undefined)' and a response '<' followed by 'false'; a prompt '>' followed by 'Boolean(null)' and a response '<' followed by 'false'; and a prompt '>' followed by 'Boolean("")' and a response '<' followed by 'false'.

```
> Boolean(undefined)
< false
> Boolean(null)
< false
> Boolean("")
< false
```

Even Boolean(0) is false due to coercion with "" and false. This can be avoided by using the "or" operator.

```
1 var a;
2
3 // goes to internet and looks for a
  value
4 a = 0;
5
6 if ([a || a === 0]) {
7   console.log('Something is there.');
```

## 7. Setting default values

ex:

```
function greet(name){  
  console.log('Hello' + name);  
}
```

In this piece of code if we do not initialize name or take it input from the user it would be taken as “undefined” and in this case, Hello undefined would be printed. To avoid this:

```
function greet(name){  
  name = name || '<Your name here>';  
  console.log('Hello' + name);  
}
```



## 8. Exploiting practical use

Note: Whenever different JS files are attached to an HTML, they executed in a single execution context.

Ex:

Consider three different JS libraries that we have and want to include on our website.

1. “lib1.js” has the following code:

```
var libraryName = "lib1";
```

2. "lib2.js" has the following code:

```
var libraryName = "lib2";
```

3. app.js has the following code:

```
console.log("libraryName");
```

4. And our HTML has;

```
<script src="lib1.js"></script>
```

```
<script src="lib2.js"></script>
```

```
<script src="app.js"></script>
```

What will the console output?

To answer this remember all these JS files form the part of the same execution context. This simply means the execution queue looks like:

```
var libraryName = "lib1";
```

```
var libraryName = "lib2";
```

```
console.log("libraryName");
```

Thus, lib2 would be printed out in the console. This can be checked by modifying lib2.js as follows:

```
window.libraryName = window.libraryName || "lib2";
```

In this case, "lib1" would be printed out as 'or' statement just takes the "first true statement".