Comparison Operators:

console.log (1 < 2 < 3); // the result will be in Boolean — true

**Now if you make changes in the above example.**

console.log ( 3 < 2 < 1 ); // the result will be Boolean — true (Very surprising)

Logically speaking, we all know that 3 is not less than 2 and 2 is not less than 1 but why the browser is showing it true. The answer lies in associativity. In the precedence document you can see that less than < operator works left to right.

So, JavaScript renders the code in the first bit it will see is 3 < 2 (3 is < less than 2) which is false. In JavaScript false is considered as 0 within numbers thus it is doing the comparison with 0. So now it will do the comparison as 0 (zero) is less than 1 (which is true). That is why it is returning true.

For guidance, In your browser's console you can check the value of true & false by inputing the following code.

True & False both are cohesion - meaning True = 1 & False = 0;

Number ( false ); // result will be 0 (in numeric values);

Number ( true ); // result will be 1 (in numeric values);

**More Explanation:**

**console.log (3 < 2 < 1); (It works left to right)**

1. 3 is less than 2 (No, so it will return false);
2. Now, false < 1 (False value is 0 and 0 is less than 1)
3. The answer would be True.

Because of the dynamic typing, there are many advantages but it also introduces some disadvantages. Basically, It attempts to translate the data type to match them. The attempt to convert or translate the type is also called coercion.

.

**Example:**

var a = 0;

var b = false;

if (a==b) {

    console.log("yes, they are equal”); // this will run because of conversion of data type

}

else {

    console.log("No, they are not equal");

}

But to prevent such catastrophe, we use strict equal to comparison operator such as === means “exactly equal to”.

Change from == to ===, now we are using strict equal to will prevent coercion and does strict comparison.

**Example:**

var a = 0;

var b = false;

if (a===b) {

    console.log("yes, they are equal");

}

else {

    console.log("No, they are not equal”); // this will run because of strict comparison of data type.

}

**Boolean & Existence:**

Like Number(), we have Boolean() - When you enter certain values inside Boolean function it will return you the value in true or false i.e

var a; // false

var a = undefined; // false

var a = null; // false

var a = 0; // false

When you put anything inside “If condition” it attempts to coheres the value to a boolean value.

if (a) {

console.log(“Something is here");

} // this code will not work until var a becomes true

but before running the if condition, you assign a value to var a then it will run because inside the if condition the value will become true.

Suppose we want to assign var a the value of 0 then our if condition needs to have a stricter check. Remember in our above code we mentioned var a = 0 and yet is returning false.

Just to make this condition work, we can assign any value to var a i.e.

var a = “hi”;

if (a) {

  console.log(“this will work because if condition will make it true then show the result");

}

New If condition will become:

Now suppose we have different values of var a.

var a = undefined; // false

var a = null; // false

var a = “”; // false

var a = 0; // false but through strict === we can prevent it from becoming false.

if ( a || a === 0) {

  console.log("it is running !");

}

When you check the following code Boolean(0), it will return as false thus to prevent that we used === to check whether that is true or not because the value 0 could be given intentionally by the programmer. We know === has higher precedence than || thus the code with === will be solved first then || condition will be solved.

**Some Basic Console.log practice to check:**

var a = 1;

Boolean(a); // true

var a = “";

Boolean(a); // false

var a = 0;

Boolean(a); // false

var a = undefined;

Boolean(a); // false

var a = null;

Boolean(a); // false

if ( a || a === 0) {

  // above statement means false || true - will be true (check in console);

  console.log("it is running !");

}

false || true // will always be true

true || false // will always be true

**Default Values:**

function greet(name) {

console.log("hello " + name);

};

greet('Ali’);

Now if you remove ‘Ali’ and leave it empty i.e. greet() then it should return “Hello Undefined”.

If you do comparison of anything with true most likely the output will be true

**true || false // true**

**undefined || true // true**

**null || true // true**

**true || “” // true**

**Undefined || any non empty string value or value that can be cohesed will become true**

Run the above example in console and you will see the result.

The operators picks the value and attempts to coheres them for example the numeric value of true is 1 thus it will return true. More examples are:

'hi' || ‘hello’ // hi - because when both values becomes the something after cohesion then it picks up the first value. Remember all the operators are essentially a function which accepts 2 parameters and returns values. If 1 value is defined then it does the cohesion.

Now if we make some changes to our code above to display us the default value, we know why it is displaying us the result.

function greet (name) {

name = name || "<Your name here>";

console.log('hello ' + name);

}

greet();

greet('Ali’);

greet(0); // remember 0 will return false so be careful - Boolean(0); // false

**It will return the value that can be cohesed meaning can be converted into another data type. That is why it will not pick up name because it is undefined but it will take the other value defined.**

**You can cohered non empty string into boolean i.e.**

**Boolean('hello’); // true - which is why in the above example when name is undefined, Javascript picked up the other non empty string value and cohered it into true.**

the “OR” operator has higher precedence than = so it will first check if name is defined then it will display the name else it will pick up the default value defined because it will coerce the value thus it will be returned to true.

If there are 2 values and both can be cohesed then it will pick up the first value and display it i.e. “hi” || “hello”; // “hi"

# Experiment

function calcTest (prop3, prop2, prop1) {

console.log(prop3 < prop2 < prop1);

}

calcTest(3,2,1);

// true