**Executing Methods on Primitive Types**

**Parenthesis And Object Property Access**

The parenthesis operator gives you control over which statement should evaluate first. That’s its primary purpose.

For example statement 5 \* 10 + 2 is not the same as 5 \* (10 + 2).

But sometimes it is used to access a member method or property. Which is demonstrated in the next source code listing. You can execute methods directly on the literal values of primitive types. Which automatically converts them to objects, so that the method can be executed.

In some cases – like with the primitives of type "number" – we must first wrap the literal value in parenthesis, or you’ll freeze your program.

console.log(1.toString) // This will freeze the execution flow

console.log((1).toString());  // "1"

console.log("hello".toUpperCase()); //  "HELLO"

console.log("HELLO WORLD".toLowerCase()); // "hello world"

console.log(new Number(1).toString());   // "1"

**Chaining Method:**

Because in JavaScript functions can return this keyword, or any other value, including functions, it’s possible to chain multiple methods using the dot operator.

// Chaining method

let s = 'HELLO World JavaScript is Awesome'

// s = s.toLowerCase()

// s = s.replace(/ /g, '|')    // globally replace all spaces with |

// s = s.trim()

// console.log(s)

s = s.toLowerCase.replace(/ /g, '|').trim()

console.log(s)

**Type Coercion Madness**

what will happen if we sporadically add up different types of values and stitch them together using the + operator?

console.log(null + {} + true + [] + [5])

**Result:**

null[object Object]true5

When + operator encounters objects of incompatible type, it will attempt to coerce those objects to their values in string format. In this case, leaving us with a new statement: "null[object Object]" + true + [] + [5].

Furthermore, when + operator encounters a string at least on one side of the operator, it will try to coerce the other side to string and perform string addition.

**Calling .toString** on true results in "true". Calling .toString on empty array brackets [] when the other side of operator is also a string evaluates it to "" which is why it appears missing from the result. And finally adding [5] to a string calls [5].toString which results in "5".

More Examples for Type Coercion:

// TYPE COERCION

console.log(null + {} + true + [] + [5]) //null[object Object]true5

console.log(true + 1) // answer = 2

console.log(true + true) //  answer = 2

console.log(true + false) // becomes 1 (true=1, false=0, 0+1 = 1)

console.log("hello"+" "+ "there.") // becomes "hello there"

console.log('Username' + 1532152) // Becomes Username1532152

console.log(1/"string") // becomes NaN(Not a Number)

console.log(NaN == NaN) // becomes false

console.log([1]+[2]) //becomes "12" <string>

console.log([]+[]) // Becomes "" <string>

console.log(Infinity)  // remains infinity <number>

console.log([]+{}) // [object object]

JavaScript will try to come up with best value available if you supply meaningless combinations of types to some of its operators.

After all, what would it mean to ”add” an object literal {} to an array []? Exactly – it doesn’t make any sense. But by evaluating to object [] at least we don’t break the code in that one little odd case where it may happen.

This safety mechanism will prevent the program from breaking. In reality, however, these types of cases will almost never happen. We can treat majority of these cases as examples – not something you should be actually trying to do in code.

// initialize some boolean based on an arguements

console.log( Boolean([])) // true

console.log(Boolean({})) // true

console.log( Boolean(true)) // true

console.log( Boolean(false)) //false

console.log( Boolean(NaN))  //false

console.log( Boolean(null))  //false

console.log( Boolean(undefined)) //false

console.log( Boolean(''))  //false

console.log( Boolean(0))  //false

console.log( Boolean(-0))  //false

In the first two cases we supplied an array literal {} and an object literal [] to Boolean constructor. What does this mean? Not much, but the point is that at least it evaluates to true in this odd case. This is just a safety net to prevent bugs.

Meaningless values still evaluate to either true or false, because these are the only values available for boolean types.

Other built-in data type constructors behave in the same way. JavaScript will try to coerce to an ideal value specific to that type.

Coercion is the process of converting a value from one type into another. For example, number to string, object to string, string to number (if the entire string consists of numeric characters) and so on...

But when values are used together with different operators not all cases are straightforward to the untrained eye.

console.log([]==[]) //false

console.log("[]"=="[]") //true

let a = [];

let b = [];

console.log(a==a)  // Becomes true

console.log(a==b) // becomes false

console.log(a !== b) // true

Let’s say that it is false because two instances of [] are not the same, because JavaScript == operator tests objects by reference and not by value.

But this statement evaluates to true because variable a points to the same instance of the array literal. They refer to the same location in memory.

**Number And String Arithmetics**

Naturally the arithmetic + operator requires two values.

console.log(5+7)  //12

If both values are integers, arithmetic operation is performed. If one of them is a string then coercion happens and string addition is invoked.

If the type of the two values provided to the arithmetic + operator is different, this conflict must be resolved. JavaScript will use type coercion to change one of the values before evaluating the entire statement to a more meaningful result.

Here + is treated as a string addition operator. The right value is converted to "1" via String(1) and then the statement is evaluated as follows:

console.log("1"+1) // becomes 11

In JavaScript there are actually three + operators: unary, arithmetic and string. Here JavaScript treats + not as the unary addition operator, but as the arithmetic addition operator instead. But... when it sees that one of the values is a string, it invokes the string addition operator. It makes no difference whether the string is on the left or right side. The statement still evaluates to a string:

Operators follow specific associativity rules. Like + and most other operators, the arithmetic addition operator (+) is evaluated from left to right: console.log(1 + "o1") //  1o1

But the assignment operator is evaluated in right to left order: For example while N is assigned value of 2, the statement itself evaluates to undefined.

N = 2;

**Operators:**

An operator is capable of manipulating a certain value or operand. Operators are used to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands. In JavaScript operators are used for compare values, perform arithmetic operations etc. There are various operators supported by JavaScript:

* **Arithmetic Operators**
* **Assignment Operators**
* **Comparison Operators**
* **Logical Operators**
* **Ternary Operators**
* **typeof Operator**

**Arithmetic Operators**

● +, // Addition

● -, // Subtraction

● \*, // Multiplication

● /, // Division

● %, // Modulus

● ++, // Post and Pre-Increment

● -- // Post and Pre-Decrement

**+ + (Increment) :**   
‘+ +’ operator increases an integer value by one. 

let **A = 10** and **Y = A + +** then A = 11, Y=10

if  **A = 10** and **Y = + + A** then A = 11, Y=11

**– – (Decrement) :**   
‘- -‘ operator decreases an integer value by one. 

let **A = 10** and **Y = A - -** then A = 9, Y=10

if  **A = 10** and **Y = - - A** then A = 9, Y=9

**Assignment operator**

The Assignment operator is  **equal (=)** which assigns the value of right-hand operand to its left-hand operand.

That is if a = b assigns the value of b to a.

The simple assignment operator is used to assigning a value to a variable. The assignment operation evaluates to the assigned value. Chaining the assignment operator is possible in order to assign a single value to multiple variables.

var x=10

var y=20

x=y // Here, x is equal to 20

y=x // Here, y is equal to 10

**There are so many assignments operator:**

let counter = 0;

counter = counter + 1;

When evaluating the second statement, JavaScript evaluates the expression on the right hand first (counter + 1) and assigns the result to the counter variable. After the second assignment, the counter variable is 1.

To make the code more concise, you can use the += operator like this:

let counter = 0;

counter += 1;

In this syntax, you don’t have to repeat the counter variable twice in the assignment.

The following table illustrates assignment operators that are shorthand for another operator and the assignment:

| **Operator** | **Meaning** | **Description** |
| --- | --- | --- |
| a = b | a = b | Assigns the value of b to a. |
| a += b | a = a + b | Assigns the result of a plus b to a. |
| a -= b | a = a - b | Assigns the result of a minus b to a. |
| a \*= b | a = a \* b | Assigns the result of a times b to a. |
| a /= b | a = a / b | Assigns the result of a divided by b to a. |
| a %= b | a = a % b | Assigns the result of a modulo b to a. |
| a &=b | a = a & b | Assigns the result of a AND b to a. |

**Comparison operators**

The **Comparison operators** are mainly used to perform the logical operations that determine the equality or difference between the values.

There are various comparison operators supported by JavaScript:

* **Equality Operators**
* **Relational Operators**

**Equality (==):**This operator is used to compare the equality of two operands. If equal then the condition is true otherwise false.

**Syntax:**

x == y

let val1 = 5;

let val2 = '5';

let val3 = 6

// Checking of operands

console.log(val1 == 5);  // true

console.log(val2 == 5);   // true

console.log(val1 == val1);  //true

console.log(val1==val3)   // false

**Inequality(!=):**This operator is used to compare the inequality of two operands. If equal then the condition is false otherwise true.

**Strict equality(===):**This operator is used to compare the equality of two operands with type. If both value and type are equal then the condition is true otherwise false.

console.log(val1 === 6); //false

console.log(val2 === '5'); // true

console.log(val1 === val1); // true

**Strict inequality(!==):**This operator is used to compare the inequality of two operands with type. If both value and type are not equal then the condition is true otherwise false.

**Relational Operators:**

**Greater than operator(>):**This operator is used to check whether the left-side value is greater than the right-side value. If the value is greater then the condition is true otherwise false.

**Greater than or equal operator(>=):**This operator is used to check whether the left side operand is greater than or equal to the right side operand. If the value is greater than or equal then the condition is true otherwise false.

**Less than operator(<):**This operator is used to check whether the left-side value is less than the right-side value. If yes then the condition is true otherwise false.

**Less than or equal operator(<=):**This operator is used to check whether the left side operand value is less than or equal to the right side operand value. If yes then the condition is true otherwise false.