# JavaScript Essentials 1 (JSE): Module 3

## Section 2

**String, comparison, and other JS operators**

Topics in this section:

- String operators: concatenation and compound assignment
- Comparison operators
- Other JS operators (typeof, instanceof, delete, and ternary)
- Operator precedence

## String operators

The only operator in this group is the **concatenation** `+`. This operator will convert everything to a **String** if any of the operands is a String type. Finally, it will create a new character string, attaching the right operand at the end of the left operand.

```
let greetings = "Hi";

console.log(greetings + " " + "Alice"); // -> Hi Alice


let sentence = "Happy New Year ";

let newSentence = sentence + 10191;

console.log(newSentence); // -> Happy New Year 10191

console.log(typeof newSentence); // -> string
```

### Compound Assignment Operators

You can probably guess that this operator can also be used in conjunction with the replacement operator. Its operation is so intuitive that we will stop with a simple example:

```
let sentence = "Happy New ";

sentence += "Year ";

sentence += 10191;

console.log(sentence); // -> Happy New Year 10191
```

## Comparison operators

Comparison operators are used to check the equality or inequality of values. **All comparison operators are binary, and all of them return a logical value representing the result of the comparison,** `true` **or** `false`**.**

As with other operators, JavaScript will try to convert the values that are being compared if they are of different types. It makes sense to check equality, or which is greater, using numeric representation, and JavaScript will in most cases convert types to a Number before comparison. There are two

exceptions to this, strings and the **identity (strict equality) operator**. Strings are compared `char` by `char` (precisely Unicode character by Unicode character using their values).

To check if the operands are equal, we can use either the **identity** (strict equality) operator `===` or the **equality** operator `==`.

The first is more restrictive, and in order to return true, the operands must be identical (i.e. they must be equal and of the same type).

```
console.log(10 === 5); // -> false
console.log(10 === 10); // -> true
console.log(10 === 10n); // -> false
console.log(10 === "10"); // -> false
console.log("10" === "10"); // -> true
console.log("Alice" === "Bob"); // -> false
console.log(0 === false); // -> false
console.log(undefined === false); // -> false
```

The equality operator requires that they are only equal, and their types are not compared. So if the operands are of different types, the interpreter will try to convert them to numbers, for example, `false` will convert to `0`, `true` to `1`, `undefined` to `NaN`, `null` to `0`, `10n` to `10` and `"123"` to `123`, etc.

Note that if any of the operands has a `NaN` value (or has been converted to `NaN`, e.g. with undefined), the equality operator will return `false`.

```
console.log(10 == 5); // -> false
console.log(10 == 10); // -> true
console.log(10 == 10n); // -> true
console.log(10 == "10"); // -> true
console.log("10" == "10"); // -> true
console.log("Alice" == "Bob"); // -> false
console.log(0 == false); // -> true
console.log(undefined == false); // -> false
console.log(NaN == NaN); // -> false
```

**Remember! Use the identity operator unless you intentionally allow for a positive comparison between the different types.**

There are also complementary operators to those just demonstrated – the **nonidentity** operator `!==` and the **inequality** operator `!=`. The first returns `true` if the operands are not identical, in other words, they are equal but of different types, or they are simply different. The second returns `true` if the operands are different.

```
console.log(10 !== 5); // -> true
console.log(10 !== 10); // -> false
console.log(10 !== 10n); // -> true
console.log(10 !== "10"); // -> true
console.log("10" !== "10"); // -> false
console.log("Alice" !== "Bob"); // -> true
console.log(0 !== false); // -> true
console.log(undefined !== false); // -> true
console.log(10 != 5); // -> true
```

```
console.log(10 != 10); // -> false
console.log(10 != 10n); // -> false
console.log(10 != "10"); // -> false
console.log("10" != "10"); // -> false
console.log("Alice" != "Bob"); // -> true
console.log(0 !=  false); // -> false
console.log(undefined != false); // -> true
console.log(NaN != NaN); // -> true
```

## Comparison operators - continued

We also have operators that allow us to check if one of the operands is bigger than `>`, smaller than `<`, bigger than or equal to `>=`, and smaller than or equal to `<=`. These operators work on any type of operand, but it makes sense to use them only on numbers or values that will convert correctly to numbers.

```
console.log(10 > 100); // -> false

console.log(101 > 100); // -> true

console.log(101 > "100"); // -> true


console.log(101 < 100); // -> false

console.log(100n < 102); // -> true

console.log("10" < 20n); // -> true


console.log(101 <= 100); // -> false

console.log(10 >= 10n); // -> true

console.log("10" <=  20); // -> true
```

You can also use them to compare strings that do not represent numbers, but the algorithm of this comparison is quite complex, and the comparison itself is not very useful. By way of simplification, single characters of both strings are tested on the same positions. It is assumed that the values of the single characters correspond to their positions in the alphabet (the letter b has a higher value than the letter a). Upper-case letters have lower values than lower-case letters, and digits have even lower values.

```
console.log("b" > "a"); // -> true

console.log("a" > "B"); // -> true

console.log("B" > "A"); // -> true

console.log("A" > "4"); // -> true

console.log("4" > "1"); // -> true


console.log("ab1" < "ab4"); // -> true

console.log("ab4" < "abA"); // -> true

console.log("abB" < "aba"); // -> true

console.log("aba" < "abb"); // -> true


console.log("ab" < "ab4"); // -> true
```

Note: the symbol => exists in JavaScript, but is not an operator – we use it in the construction of arrow functions.

## Other operators

The list of operators in JavaScript is much longer, but many of them would not be particularly useful at this stage of learning, such as bitwise operators, which operate on single bits of operands. However, it is worth mentioning a few more operators, some of which have already appeared in earlier examples.

### typeof

We already introduced the `typeof` operator when discussing data types. It is a unary operator, which checks the type of operand (it can be a variable or a literal). The operator returns a string with the type name, such as "boolean" or "number".

If you want to refresh your knowledge of this operator, go back to the section about data types.

```
let year = 10191;

console.log(typeof year); // -> number

console.log(typeof false); // -> boolean
```

### instanceof

The `instanceof` operator appeared while discussing arrays. It is a binary operator that checks whether an object (left operand) is of some type (right operand). Depending on the test result, it returns true or false.

During this course, the usefulness of this operator is limited to testing whether a variable contains an array.

```
let names = ["Patti", "Bob"];

let name = names[0];

console.log(names instanceof Array); // -> true

console.log(name instanceof Array); // -> false
```

### delete

The unary `delete` operator was introduced while discussing objects. It allows you to delete a selected field of the object whose name is indicated with an operand.

```
let user = {
  name: "Alice",
  age: 38
};

console.log(user.age); // -> 38
```

```
delete user.age;

console.log(user.age); // -> undefined
```

### ternary

The last of the operators discussed is quite unusual, because it is the only operator using three operands. It is a conditional operator. Based on the value of the first operand (true or false), the value of the second or third operand, respectively, is returned. This operator is most often used to place one of the two values in the variable depending on a certain condition. We will come back to the operator when discussing the conditional if, but here we'll provide only a simple example of its use. The three operands are separated from each other by ? (the first from the second) and : (the second from the third).

```
console.log(true ? "Alice" : "Bob"); // -> Alice

console.log(false ? "Alice" : "Bob"); // -> Bob
```

Each of these operands can be an expression that must be calculated. In the following example, the first operand is a comparison of two numbers using a comparison operator. The result of the comparison will be false, which will be used by the conditional (ternary) operator. Here we come to an important problem about operator precedence and order of execution. In a moment, we will say a few more words about it.

```
let name = 1 > 2 ? "Alice" : "Bob";

console.log(name); // -> Bob
```

## Precedence

Practically in all the examples where we presented the operation of successive operators, we followed instructions in which one operator was used. In reality, usually multiple operators are used simultaneously. At this point, a quite important question arises: in what order will the interpreter perform them? This will of course affect the final result of the operators, so it is worth taking this into account when writing the instructions.

```
let a = 10;

let b = a + 2 * 3;

let c = a + 2  < 20 - 15;

console.log(a); // -> 10

console.log(b); // -> 16

console.log(c); // -> false
```

In the second line of the example (variable b declaration), the operators are executed in the order we know from mathematics. First, multiplication is performed, then addition, and at the end the resulting value is assigned to the variable. In the third line (declaration of variable c) the matter gets a little more complicated. First, the sum of variable a and number 2 is calculated, then the sum of numbers 20 and 15, and both results are compared with the logical operator (less than) and the result is placed in variable c.

The JavaScript interpreter uses two operator properties to determine the sequence of operations: precedence and

associativity. Precedence can be treated as a priority, with some operators having the same precedence (e.g. addition and subtraction). Associativity allows you to specify the order of execution if there are several operators with the same priorities next to each other.

Precedence can be presented as a numerical value – the higher the value, the higher the priority of the operation. If, for example, an ***OP₁*** operator has a smaller precedence than ***OP₂***, then the instruction:

*a* ***OP₁*** *b* ***OP₂*** *c*

will be executed as follows: first, **OP₂** will be executed on operands *b* and *c*, then ***OP₁*** will be executed on the left operand a and the right operand, resulting from ***OP₂***. So the instruction could be presented in the form:

*a* ***OP₁*** ( *b* ***OP₂*** *c*)

If we perform the same operations (or different operations but with the same precedence), the interpreter uses associativity to determine the order of operations. Operators may have a specified left-associativity (left to right order) or right-associativity (right to left order). Let's assume that in our example, the operator OP1 has left-associativity:

*a* ***OP₁*** *b* ***OP₂*** *c*

In such a situation, the OP1 operation on operands a and b will be performed first, followed by a second OP1 operation on the received result and operand c. Bearing in mind that we are dealing with left-associativity, we could write the instruction in the following form:

(*a* ***OP₁*** *b*) ***OP₂*** *c*

It follows that it would be appropriate to know not only the precedence of all operators, but also their associativity. This may seem a bit overwhelming, taking into account the number of operators. Fortunately, it is usually enough to know the properties of the most basic operators and use brackets in doubtful situations. The brackets allow you to impose the order of operations, just like in mathematics. Keep this in mind when viewing the table below. It contains a list of operators we already know with their precedence and associativity, so it is quite large. You absolutely do not have to remember everything if you can use brackets to group operations.

## Precedence - continued

| Precedence | Operator | Associativity | Symbol |
|---|---|---|---|
| 14 | Grouping | n/a | ( … ) |
| 13 | Field access | ⇒ | … . … |
| 12 | Function call | ⇒ | … ( … ) |
| 11 | Postfix increment | n/a | … ++ |
| | Postfix decrement | n/a | … -- |
| 11 | Logical NOT | ⇐ | ! … |
| | Unary plus | ⇐ | + … |
| | Unary negation | ⇐ | - … |

| Precedence | Operator | Associativity | Symbol |
|---|---|---|---|
| | Prefix increment | ⇐ | |
| | Prefix decrement | ⇐ | |
| | typeof | ⇐ | |
| | delete | ⇐ | |
| 9 | Exponentiation | ⇐ | |
| | Multiplication | ⇒ | |
| 8 | Division | ⇒ | |
| | Remainder | ⇒ | |
| | Addition | ⇒ | … + … |
| 7 | Subtraction | ⇒ | … - … |
| | Less than | ⇒ | … < … |
| | Less than or equal | ⇒ | … <= … |
| 6 | Greater than | ⇒ | … > … |
| | Greater than or equal | ⇒ | … >= … |
| | instanceof | ⇒ | … instanceof … |
| | Equality | ⇒ | … == … |
| 5 | Inequality | ⇒ | … != … |
| | Strict Equality | ⇒ | … === … |
| | Strict Inequality | ⇒ | … !== … |
| 4 | Logical AND | ⇒ | … && … |
| 3 | Logical OR | ⇒ | … \|\| … |
| 2 | Conditional (ternary) | ⇐ | … ? … : … |
| 1 | Assignment | ⇐ | … = …<br>… += …<br>… *= … |
| | … and other assignment o | | |

A few words of explanation about the table.

An arrow in the associativity column facing the right side means left-to-right associativity, while facing the opposite side means right-to-left associativity.

The abbreviation n/a means not applicable, because in some operators the term associativity does not make sense.

At the very beginning of the table, there are three operators that may need further explanation:

- **grouping** is simply using brackets. They take precedence over the other operators, so we can use them to force the execution of operations to take priority;

- **field access (member access)** is the operator used in dot notation, which is when accessing a selected object field. It takes precedence over other operators (except for brackets), so for example the instruction:

  `let x = myObject.test + 10;` means that the value of the test field of the `myObject` object will be fetched first, then we will add a value of 10 to it, and the result will go to the x variable;

- **function call** precedence tells us that if we call a function, this action will take priority over other operations, except for grouping in brackets and the field access operator (dots). So in the example:

  `let y = 10 + myFunction() ** 2;` `myFunction` will be called first, the result returned by it will be raised to power 2, and only then we will add 10 to the total and save the result to variable y.

Remember, however, if you have any doubts, just use brackets to order the precedence of the operators used. They allow you to organize even the most confusing instructions that come to mind.

```
let a, b;

b = (a = (20 + 20) * 2) > (3 ** 2);

console.log(a); // -> 60

console.log(b); // -> true
```

The use of parentheses not only forces the order of actions, but also increases the readability of the code (the person reading it does not have to wonder what and in what order it will be done).

The full list of operators and properties can be found on the **MDN pages**.

## Section Summary

In this chapter, we have introduced a few new operators (e.g. logical operators), and we have also solidified our knowledge about a few that we already know (e.g. assignment operators). Together with the operators, new terms describing their properties – **precedence** and **associativity** – have appeared.

It is likely that the amount of new information that has emerged, especially concerning operator precedence, may seem a bit daunting. Don't worry, it is quite normal. In fact, there are probably not very many JavaScript programmers who would be able to correctly set all operators according to their priorities.

Fortunately, we have parentheses to help, which makes it easier for us to force the priority of operations.

The operators will appear in all the examples until the end of the course, so you will gradually consolidate the knowledge you've gained, which will surely become a logical and coherent whole over time.