JavaScript For Web

Week 1, Lecture 4 - JavaScript Fundamentals: More operators and Data Types

Instructor: Jason Xu

Today's Overview

- JavaScript Fundamentals: More operators and Data Types
- Tutorial: No assignment exercise today, we will do some readings and reviews.

JavaScript Operators Review: Maths

- Addition `+`,
- Subtraction `-`,
- Multiplication `*`,
- Division `/`,
- Remainder(Modulus/Mod) `%`,
- Exponentiation `**`.

String concatenation with `+`

Let's meet the features of JavaScript operators that are beyond school arithmetics.

- Usually, the plus operator `+` sums numbers. But, if the binary + is applied to strings, it merges (concatenates) them:

```
let s = "my" + "string"
console.log(s) // "mystring"
```

- if any of the operands is a string, then the other one is converted to a string too:

```
console.log( "1" + 2 ) // "12"
console.log( 2 + "1" ) // "21"
```

String concatenation with `+` continued

- More complex example

```
console.log(2 + 2 + "1" ) // "41" and not "221"
console.log("1" + 2 + 2) // "122" and not "14"
```

- Other arithmetic operators work only with number and always convert their operands to numbers.

```
console.log( 6 - "2" ) // 4, converts '2' to a number
console.log( "6" / "2" ) // 3, converts both operands to numbers
```

My suggestion: You don't have to memorize all of this rules, **try it at the console of the web inspector** if you are not sure about the result.

```
console.log(1 + "1" + 11 + "1")
console.log(typeof (1 + "1" + 11 + "1"))
```

Numeric conversion with +

The unary plus or, in other words, the plus operator + applied to a single value, doesn't do anything to numbers. But if the operand is not a number, the unary plus converts it into a number. For example:

```
// No effect on numbers
let x = 1
console.log( +x ) // 1

let y = -2;
console.log( +y ) // -2

// Converts non-numbers
console.log( +true ) // 1
console.log( +"" )   // 0
```

Numeric conversion

The need to convert strings to numbers arises very often. For example, if we are getting values from HTML form fields, they are usually strings. We can use unary `+` to do this:

```
let apples = "2"
let oranges = "3"

console.log( apples + oranges ) // "23", the binary plus concatenates strings

console.log( +apples + +oranges ) // 5, both values converted to numbers before the binary plus
```

We can also use `Number(...)` to convert string to number:

```
console.log( Number(apples) + Number(oranges) ); // 5
```

How about convert a number to string?

Assignment Operator

`=` assign the value to the variable.

```
let num = 5
```

Modify-in-place

- We often need to apply an operator to a variable and store the new result in that same variable. For example:

```
let num = 2
num = num + 5 // now n = 7
num = num * 2 // now n = 14
```

- This notation can be shortened using the operators `+=` and `*=`:

```
let num = 2
num += 5 // now n = 7 (same as num = num + 5 )
num *= 2 // now n = 14 (same as num = num * 2)
```

Short “Modify-and-assign” operators exist for all arithmetical and bitwise operators.

Increment/Decrement Operator

Increasing or decreasing a number by one is among the most common numerical operations.

- **Increment** `++` increases a variable by 1:

```
let counter = 2
counter++ // works the same as counter = counter + 1, but is shorter
console.log(counter) // 3
```

- **Decrement** `++` decreases a variable by 1:

```
let counter = 2
counter--  // counter = 1, works the same as counter = counter + 1, but is shorter
console.log(counter) // 1
```

Increment/Decrement Operator continued

The operators `++` and `--` can be placed either before or after a variable.

- When the operator goes after the variable, it is in "postfix form": `counter++`.
- The "prefix form" is when the operator goes before the variable: `++counter`.

Both of these statement do the same thing: increase `counter` by `1`, we can only see the difference if we use the returned value of `++/--`.

```
let counter = 1
let a = ++counter

console.log(a) // 2
```

The `++counter` increments `counter` and returns the new value, `2`. So, the output is `2`.

```
let counter = 1
let a = counter++

console.log(a) // 2
```

The `counter++` also increments `counter` but returns the old value (prior to increment). So, the output `1`.

Summary: if the result of increment/decrement is not used, there is no difference in which form to use.

Operator precedence

If an expression has more than one operator, the execution order is defined by their **precedence**, or, in other words, the default priority order of operators.

- From school, we all know that the multiplication in the expression `1 + 2 * 2` should be calculated before the addition. That's exactly the precedence thing. The multiplication is said to have a **higher precedence** than the addition.
- Parentheses override any precedence, so if we're not satisfied with the default order, we can use them to change it. For example, write `(1 + 2) * 2`.
- The "unary plus" has a priority of 14 which is higher than the 11 of "addition" (binary plus). That's why, in the expression `+apples + +oranges`, unary pluses work before the addition.
- The assignment operator has lower precedence than arithmetic operators. For example, the calculations are done first and then the `=` is evaluated, storing the result in `num`:

```
let num = 2 * 2 + 1
```

Check the operator precedence table on MDN.

JavaScript Data Types

There are eight basic data types in JavaScript

Seven primitive data types:

- `number` for numbers of any kind: integer or floating-point, integers are limited by ±(2 -1).
- `bigint` for integer numbers of arbitrary length.
- `string` for strings. A string may have zero or more characters, there's no separate single-character type.
- `boolean` for true/false.
- `null` for unknown values – a standalone type that has a single value null.
- `undefined` for unassigned values – a standalone type that has a single value undefined.
- `symbol` for unique identifiers.

And one non-primitive data type:

- `object` for more complex data structures.

Today we will discuss all data types except `object` and `symbol`. For `object`, we will discuss it very soon, which is the most important data type/concepts in JavaScript Web Development.

`typeof`

The typeof operator allows us to see which type is stored in a variable.

```
let x = 0;

console.log(typeof x)
//OR
console.log(typeof(x))
```

number

```
let n = 123
n = 12.345
```

The number type represents both integer and floating point numbers.
There are many operations for numbers, e.g. multiplication `*`, division `/`, addition `+`, subtraction `-`, and so on.
Besides regular numbers, there are so-called "special numeric values" which also belong to this data type: `Infinity`, `-Infinity` and `NaN`.

- `Infinity` represents the mathematical Infinity ∞. It is a special value that's greater than any number. We can get it as a result of division by zero:

```
console.log(1 / 0)
console.log(-Infinity)
```

- `NaN` represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

```
console.log(NaN + 2)
console.log(3 * NaN)
console.log("not a number" / 2)
```

bigint

In JavaScript, the “number” type cannot safely represent integer values larger than 2 -1 (that’s 9007199254740991), or less than -(2 -1) for negatives.

To be really precise, the “number” type can store larger integers (up to 1.7976931348623157 * 10). But outside of the safe integer range ±(2 -1), there’ll be a precision error, because not all digits fit into the fixed 64-bit storage. So an “approximate” value may be stored.

```
console.log(9007199254740991 + 1) // 9007199254740992
console.log(9007199254740991 + 2) // 9007199254740992
```

A `bigint` value is created by appending `n` to the end of an integer:

```
// the "n" at the end means it's a bigint
const bigInt1 = 1n
const bigInt2 = 12345678901234567890123456789012345678901234567890n
```

string

A string in JavaScript must be surrounded by quotes. In JavaScript, there are 3 types of quotes:

- Double quotes: `"Hello"`.
- Single quotes: `'Hello'`.
- Backticks: `` `Hello` ``.

```
let username = "Jason"
let userId = '56498725'
let phrase = `Hello, my name is ${username} and my ID is ${userId}.`
```

Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in `${…}`, for example:

```
let name = "John"

// embed a variable
console.log( `Hello, ${name}!` ) // Hello, John!

// embed an expression
console.log( `the result is ${1 + 2}` ) // the result is 3
```

boolean

The boolean type has only two values: `true` and `false`.

```
let nameFieldChecked = true // yes, name field is checked
let ageFieldChecked = false // no, age field is not checked
```

Boolean values also come as a result of comparisons

```
let isGreater = 4 > 1

console.log( isGreater ) // true (the comparison result is "yes")
```

Common comparison operators:

- Greater/less than: `a > b`, `a < b`.
- Greater/less than or equals: `a >== b`, `a <== b`.
- Equals: `a === b`, please note the triple equality sign `===` means the equality test, while a single one `a = b` means an assignment.
- Not equals: In maths the notation is ≠, but in JavaScript it's written as a `!==` b.

boolean continued

comparison operators `==` vs `===`

- The double equality sign `==` mean the the equality test.

```
console.log( 2 >= 1 )  // true
console.log( 2 == 1 ) // false
console.log( 2 != 1 ) // true
```

- A regular equality check `==` has a problem. It cannot differentiate data types because operands of different types are converted to numbers by the equality operator `==`:

```
console.log( 0 == false )  // true
console.log( '' == false ) // true
```

- A **strict equality operator** `===` checks the equality without type conversion.

```
console.log( 0 == false )  // false
```

My suggestion: use **strict equality** check all the time.

`null` and `undefined`

The `null` value forms a separate type of its own which contains only the `null` value. It's just a special value which represents “nothing”, “empty” or “value unknown”.

```
let age = null
```

The `undefined` also makes a type of its own, just like `null`. It means “value is not assigned”.

```
let age
```

It is possible to explicitly assign undefined to a variable but we don’t recommend doing that. Normally, we use `null` to assign an “empty” or “unknown” value to a variable, while `undefined` is reserved as a default initial value for unassigned things.

Summary

- String concatenation
- Numberic conversion
- Type conversion
- Asssignment operators
- Incremental/Decremental operators
- Operator precedence
- Commonly used primitive data types in JavaScript

```
typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof null // "object"  (2)

typeof undefined // "undefined"
```