JS | Data Types in JavaScript - number & string

Learning Goals

After this lesson you will:

- Know what are two main kinds of data based on its value in JavaScript
- Be able to use number as a data type
- Be able to use string as a data type and get familiar with some string methods

Two Main Kinds of Data

There are two kinds of data in JavaScript:

1. **primitives** or **primitive values** and
2. **objects** or **non-primitive values**.

According to MDN, a primitive (a.k.a. primitive value or primitive data type) is **any data that is not an object and has no methods**.

This being said, in JavaScript, there are 6 primitive data types:

- number,
- string,
- boolean,
- null,
- undefined,
- symbol (latest added in ECMAScript2015)

We will come back to the immutability but for now, keep in mind that **all primitive data types are immutable**.

Let's talk a bit about numbers as data types 🔢

A number as data type

Using numbers, we can represent **integers** and **floating-point numbers** in JavaScript.

```
const age = 34;
const price = 12.99;
```

Number as data type also support a **special numeric values**:

`NaN` and `Infinity`. We really don't have to go in details here but `NaN` is something that you'll see throughout this course so let's explain a bit.

`Nan` stands for **Not a Number** and it represents a **computational error**. It is a result of an incorrect mathematical operation, such as:

```
const name = "Sandra"; // <== string data type
const whatIsThis = name/2;

console.log(whatIsThis); // ==> NaN
```

Obviously *NaN* is not a number like any other, it just belongs to this data type. It is very important to keep in mind that if you get *NaN* and you expected to get a number after some mathematical operation, there's a good chance you are trying to apply incorrect math operation on top of string or some other data type (that's not a nymber).

Number expressions

If you're familiar with math or other sciences, the term `operator` is well known to you. When we're doing basic addition, in the example `2 + 2`, `+` is the `operator`, and the operation exectuted here is `addition`.

Let's recap some basic math operations:

- `+` addition
- `-` subtraction
- `*` multiplication
- `/` division

Everyone is familiar with these operators, but in case you want to play a bit with them, here's a codepen:

```
console.log(2 + 2);
console.log(4 - 2);
console.log(3 * 2);
console.log(6 / 2);
```

Advanced Operators

Exponentiation

In math, there is a very useful concept called exponentiation. Exponentiation is the process of taking a quantity `b` (the base) to the power of another quantity `e` (the exponent).

In JavaScript, we can easily use exponentiation by using the `**` (exponentiation) operator:

```
console.log(2**3);
// => 8
```

Modulo

Modulo (`%`) is the remainder operator. Think of this as saying *If I divide the first number by the second, what is the remainder?*

This is very handy for finding multiples of a particular number, and many other use cases:

```
// 4 / 2 = 2
console.log(4 / 2);
//With a remainder of 0
console.log(4 % 2);

// 7 / 2 = 3.5
console.log(7 / 2);
//With a remainder of 1
console.log(7 % 2);

// If a number modulus other number is equal to 0
// it is a multiple of "other number"

// 8 is indeed a multiple of 2!
console.log(8 % 2 === 0);
// 9 is NOT a multiple of 2!
```

```
console.log(9 % 2 === 0);
```

Assigment Operators

Previously we learned how to assign values to variables. We use `=` sign to do this. To make sure we are all on the same page:

> The basic assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x. (source: Assignment operators)

Very commonly used assignment operator is `+=` and here is example how to use it:

```
let myAge = 25;

myAge += 1;
console.log(myAge);
```

`+=` is the equivalent of saying `myAge = myAge + 1`. Adding `myAge` and `1` on its own *does not* change the value of myAge, it simply adds the two together and `returns` you a value computated. (**Remember this when we talk about immutability a bit later in the lesson.**)

Basic Assignment Operators Table

These are the most used assignment operators:

Name	Operator	Equivalent
Assignment	x = y	N / A
Addition assignment	x += y	x = x + y
Subtraction assignment	x -= y	x = x - y
Multiplication assignment	x *= y	x = x * y
Division assignment	x /= y	x = x / y

Remainder assignment	x %= y	x = x % y
Exponentiation assignment	x **= y	x = x ** y

To see the full list, visit Assignment Operators - Overview.

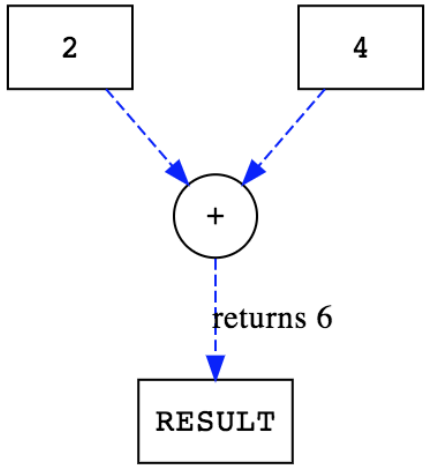
Expressions

An expression is a combination of any `value` (number, string, array, object) and set of `operators` that result in another value.

So we can say that the following is the example of *expression*:

```
2 + 4
```

And this is its correspondent parse tree:



Take the number two and add four to it.

Another example is this:

```
const result = ((7 + 5) / 3) - 8;
console.log(result);

// => -4
```

- Take the number 7, add it to 5

- Divide this new value by 3
- Take that value and then subtract 8
- Assign that value to `result`

Parentheses are known as a grouping operator.

It seems JavaScript knows in what order to put the numbers together. How does it do this?

Well it literally follows the basic mathematic rules - let's shortly refresh our memory.

Operator Precedence

In mathematics and computer programming, the order of operations (or operator precedence) is a collection of rules that define which procedures to perform first in order to evaluate a given mathematical expression.

Expressions in math have a particular order in which they get evaluated, based on the operators they use.

```
2 + 2 = 4
2 + 2 * 2 = 6
(2 + 2) * 2 = 8
```

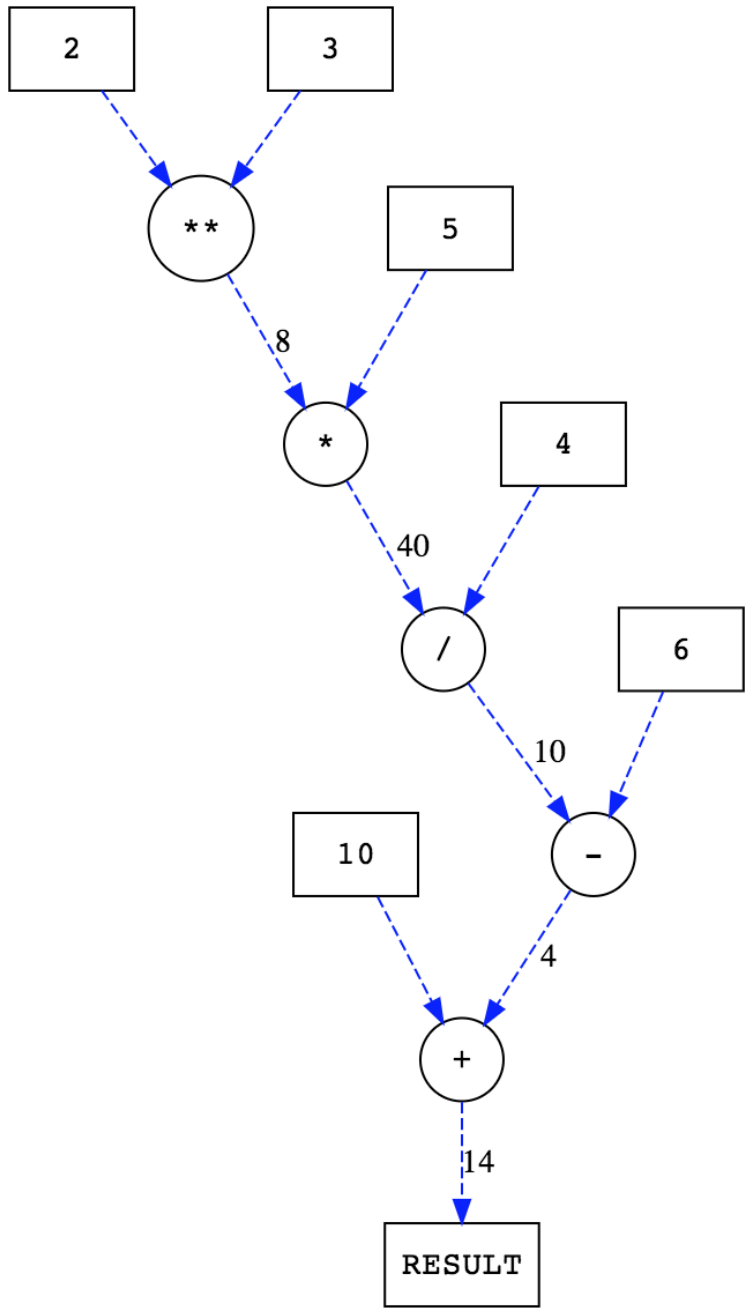
As we said, in JavaScript, the same as in math, we have to follow **PEMDAS** rules.

Precedence	Operator	Name
1	`()`	**P**arantheses
2	`**`	**E**xponents
3	`*`	**M**ultiplication
4	`/`	**D**ivision
5	`+`	**A**ddition
6	`-`	**S**ubtraction

In the numerical order, anything that comes first will be executed first (**1** for **Parentheses**, **2** for **Exponents**, etc.), meaning that anything in parantheses will be executed first, exponents second, multiplication third, etc.

```
const i = 10 + 5 * 2 ** 3 / 4 - 6
//  === 10 + 5 * 8 / 4 - 6 <== start with the exponents (2 ** 3)
//  === 10 + 40 / 4 - 6 <== then multiplication (5 * 8)
//  === 10 + 10 - 6 <== then division (40 / 4)
//  === 20 - 6 <== then addition (10 + 10 )
//  ==> 14 <== and finally finish with subtraction (20 - 6)
```

This Parse Tree diagram may help you understand it more visually :)



You can find a list of these operators, and the order in which they are executed here at MDN.

Exercise: Guess the Expression Result!

Take a solid guess at what the result of the expression is going to be!

Tip: To see the actual result, uncomment the `console.log` by pressing ⌘ + `/`

```
const expressionOne = ((2 * 2) + 5) * 6;
// console.log(expressionOne);

const expressionTwo = ((2* 2) + (5 * 3)) - 5;
// console.log(expressionTwo);

const expressionThree = (5 * 5) / (5 * 5);
// console.log(expressionThree);

const expressionFour = 5 * 5 - 5 * 4;
// console.log(expressionFour);
```

A string as data type

What is a string?

A `string` is simply **a sequence of characters**. A `character` can be a letter, number, punctuation, or even things such as new lines and tabs.

Creating a String

To create a string in JavaScript you have to use one of these **quotes**:

- `""` (*double quotes*),
- `''` (single quotes) or
- `` ``` `` (backticks).

Between double and single quotes there's no real difference, so it's matter of preference.

Backticks have "extra" functionality because using them we can **embed variables and expressions in the strings**:

```
let name = "Ana";
console.log(`Hello there, ${name}!`); // <== Hello there, Ana!
console.log(`${name} walks every day at least ${1+2} km 👏`); // <== Ana walks every day
at least 3km 👏
```

Another great functionality of backticks is possibility to easy create **new lines** in the same string (meaning the string can span into multiple lanes):

```
const fruits = `
1. banana 🍌
2. apple 🍎
3. orange 🍊
4. cherry 🍒
`;

console.log(fruits);
// 1. banana 🍌,
// 2. apple 🍎,
// 3. orange 🍊,
// 4. cherry 🍒
```

As we can see, each fruit is on its own line ✅.

Special characters

Some strings are special because they contain special characters. This means that we have to use `escape sequences` to make everything work.

For example, when you want to have something that is quoted in the middle of your string (sentence), you will have to use some “magic” 🎩.

```
const favBook = "My favorite book is "Anna Karenina".";
console.log(favBook); // <== error: Unexpected token
```

If you can use single quotes, no problem:

```
const favBook = "My favorite book is 'Anna Karenina'.";
console.log(favBook); // <== My favorite book is 'Anna Karenina'.
```

If you, however, for some reason have to use double quotes, your way around this would be using **backslash escape** character.

```
const favBook = "My favorite book is \"Anna Karenina\".";
console.log(favBook); // <== My favorite book is "Anna Karenina".
```

The same applies for the following:

```
const mood = 'I\'m OK.';
console.log(mood); // <== I'm OK.
```

So, to conclude, you should use `\` (backslash) when there's a need to excape a special character in a string.

It's still possible to create **multilane strings** with double or single quotes but with a help of "new line character" `\n`.

```
const fruits = " 1. banana 🍌\n 2. apple 🍎\n 3. orange 🍊\n 4. cherry 🍒";

console.log(fruits);
// 1. banana 🍌,
// 2. apple 🍎,
// 3. orange 🍊,
// 4. cherry 🍒
```

To summarize - these are different ways of doing the same:

```
console.log("Web Dev \nUX/UI");
console.log(`Web Dev
UX/UI`);

// both consoles are the same:
// Web Dev
// UX/UI
```

⭐ You can see a full list of these special characters at the Mozilla Developer Network.

String length

`.length` is a numeric property of a string.

```
const name = "Ana";
console.log(name.length); // <== 3
```

`length` is not a method of a string so don't try to get it by putting parentheses after 👎 `name.lenghth()` ❌

Methods for string manipulation

Manipulating and modifying strings in code are common operations. Simple things such as capitalizing a name, or checking to see if a word starts with some letter are very common.

JavaScript includes a **String library of methods** to simplify some of the most common tasks on strings. Let's look at how to perform some of these operations.

Adding To Strings

We can easily `concatenate` or add characters to strings with the `+` or `+=` operator.

```
let emptyContainer = "";
emptyContainer += "Hello there, student!";
// += is equivalent to saying:
// emptyContainer = emptyContainer + "Hello there, student!";
console.log(emptyContainer);

// ERROR CASE
// At this moment the value of the emptyContainer is "Hello there, student!"
emptyContainer + " How are you?";
// We would expect the value to be "Hello there, student! How are you?"
// However, the value is still "Hello there, student!", because we didn't reassign the
variable value with `+=`
console.log(emptyContainer);

// If we want it to console "Hello there, student! How are you?",
// we need to have the following: emptyContainer += " How are you?";
```

Accessing characters

One of the ways to access the characters inside the string is using **`charAt(n)`** method.

`charAt(n)` shows the character on the `n`th position in the string but keep in mind, the first character is indexed with zero (0).

```
const greeting = "Hello there!";
console.log(`"${greeting}" is a string and it's length is ${greeting.length}.`);
// "Hello there!" is a string and it's length is 12.
console.log(greeting.charAt(0)); // <== H
```

```
console.log(greeting.charAt(1)) // <== e
console.log(greeting.charAt(5)); // <== " "
console.log(greeting.charAt(11)); // <== !
console.log(greeting.charAt(12)); // <== "" as an empty string
```

We can access characters inside of strings with their **`index`** number. As we said, the index **starts at 0**.

```
const greeting = "Hello there!";
console.log(greeting[0]); // <== H
console.log(greeting[3]); // <== l
console.log(greeting[9]); // <== r
console.log(greeting[-2]); // undefined
```

Finding a substring

JavaScript has a cool **`.indexOf(substr)`** method that returns the index where a particular character/substring occurs. If the substring was not found, it returns `-1`.

```
const message = "Don't be sad, be happy!";
console.log(message.indexOf("Don't")); // <== 0
console.log(message.indexOf("t")); // <== 4
console.log(message.indexOf("Be")); // <== -1 (capitalized Be ≠ lowercased be)
console.log(message.indexOf("py")); // 20
```

The substring `be` appears more than once. To see the next occurance, we need to tell somehow our .indexOf() method to skip the first one.

```
const message = "Don't be sad, be happy!";
console.log(message.indexOf("be")); // <== 6
console.log(message.indexOf("be", 7)); // <== 14
```

What we did was passing a second parameter, which represents a value where the first occurence appeared (it was 6) + 1. So we are telling the method to skip the positions from `0 to 7` and keep looking for the occurence of the first parameter (in our case: *"be"*).

If we need to look for a substring but from the end to its beginning, you can use **`str.lastIndexOf(substr)`**. It shows occurences in the reverse order.

```
const message = "Don't be sad, be happy!";
console.log(message.lastIndexOf("be"));
// The index of the first "be" from the end is 14
```

📝Practice

Write code that finds the index of the letter “j” in `My favorite dessert is jello`.

.repeat()

Repeat does exactly what it sounds like. Call repeat on a specific string, and pass it an argument of the times it is to be repeated.

```
console.log("$".repeat(3));
console.log("la".repeat(10));
```

Getting a substring

In JavaScript, we can use

- **.substring()**,
- **.substr()** and
- **.slice()**
  to get a substring from a string. Each of these methods is used for **getting the part of the string between start and end** but they have slight differences.

```
const message = "Don't be sad, be happy!";
let withSubstring = message.substring(0,3);
console.log(withSubstring); // <== Don

let withSubstr = message.substr(0,3);
console.log(withSubstr); // <== Don

let withSlice = message.slice(0,3);
console.log(withSlice); // <== Don
```

As we can see, they all give the same results. What if we pass a **negative** values?

```
let withSubstring = message.substring(-3,-1);
console.log(withSubstring); // <== "" (empty string)
```

```
let withSubstr = message.substr(-3,-1);
console.log(withSubstr); // <== "" (empty string)

let withSlice = message.slice(-3,-1);
console.log(withSlice); // <== py
```

Only **.slice() supports negative values** and they mean the **position is counted from the string end**.

It's matter of your personal preference which one to use.

Sorting the strings - .toLocaleCompare()

According to MDN, the **.localeCompare()** method returns a number indicating whether a string comes before or after or is the same as some other string in sort order.

How this method works?

```
'str1'.localeCompare('str2');
```

👍 Returns **1** if `str1` is greater than `str2` according to the language rules.

👍 Returns **-1** if `str1` is less than `str2`.

👍 Returns **0** if they are equal.

```
console.log('barcelona'.localeCompare('miami') ); // -1
console.log('miami'.localeCompare('barcelona') ); // 1
console.log('Miami'.localeCompare('miami') ); // 1
```

ES6 introduced a couple more methods but we will cover them in the later learning unit.

Summary

In this lesson we learned how to declare, use and manipulate numbers and strings, two primitive data types.

Extra Resources

- Most of the JavaScript String methods can be found at MDN
- Using special characters in strings