Numbers & Strings

Learning objectives

After this lesson you will:

- know what are the two main kinds of data types in JavaScript based on their values
- be able to use the number data type
- be able to use the string data type
- become more familiar with some string methods

Two Main Kinds of Data Types

There are two kinds of data types 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 concept of immutability but, for now, keep in mind that **all primitive data types are immutable**.

This video would be a good start to understand primitives and non-primitives.

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 a data type also supports **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 incorrect mathematical operation, such as:

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

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

NaN is not a normal number, although it belongs to this data type. If you get *NaN* and you expected to get a number after some mathematical operation, you are probably performing the operation on a string or some other data type that isn't a number.

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 executed here is addition.

Let's recap some basic math operations:

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

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).

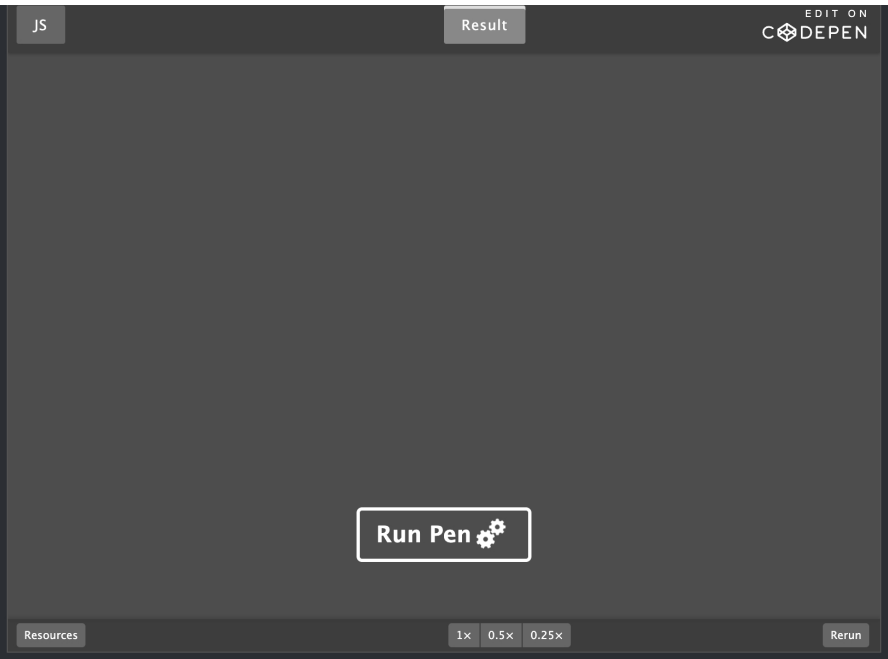
In JavaScript, we can easily use exponentiation by using the ** (exponentiation) operator:

```
console.log(2 ** 5);
// 2 * 2 * 2 * 2 * 2
// => 32
```

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:



Assignment 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 an example of 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 the value computed. (**Remember this when we talk about immutability a bit later in the lesson.**)

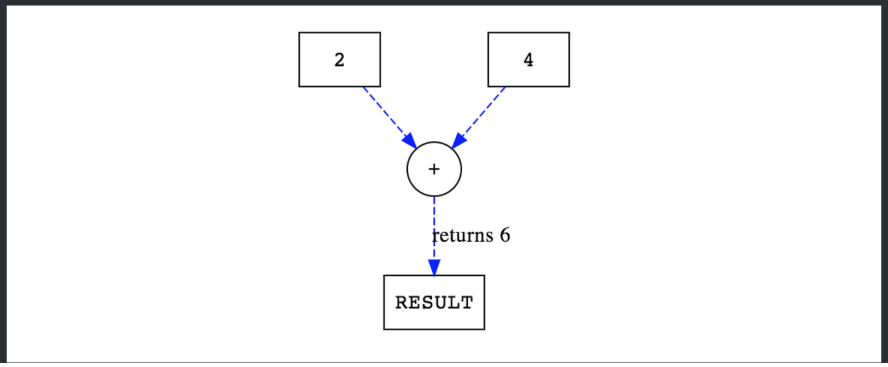
Expressions

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 an example of an *expression*:

```
2 + 4;
```

And this is its correspondent parse tree:



Take 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 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.

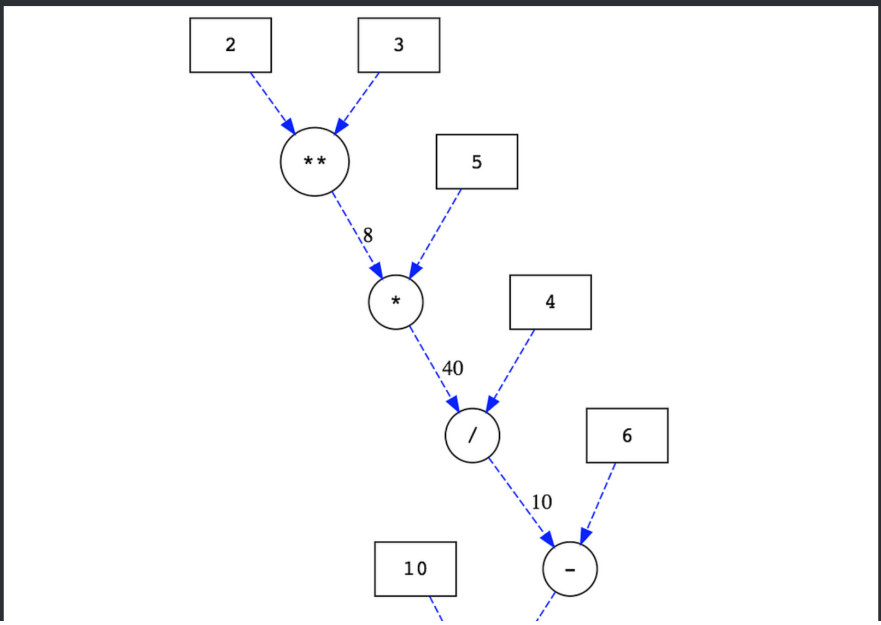
Expressions in math have a particular order in which they get evaluated, based on the operators they use.

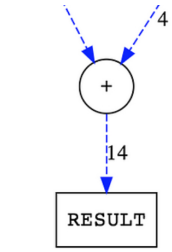
2 + 2 = 4 2 + 2 *2 = 6 (2 + 2)* 2 = 8

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

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

```
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)
```





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

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 (grave accents).

There's no real difference between double and single quotes, so it is a matter of preference. **Backticks**, however, have **"extra" functionality**. Using backticks, we are actually creating **template literals**.

> ✅ **Template literals** are string literals allowing embedded expressions.

Strings in JavaScript have been historically limited. ES6 Template Strings introduces an entirely different way of solving these problems.

```
let greeting = `Yo, Root Learn!`;
```

One of their first real benefits is **string substitution**. Substitution **allows us to take any valid JavaScript expression and place it inside a template literal, and the result will be output as part of the same string**.

In simple English, using backticks we can **embed variables and expressions inside the strings**. Template strings can contain placeholders for string substitution using the **${ } syntax**, as demonstrated below:

```
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.
```

ES5 Old-school style!!

```
var customer = { firstName: 'Foo', lastName: 'Kim' };
var message = 'Hello ' + customer.firstName + ' ' + customer.lastName + '!!';
console.log(message);
```

ES6 Interpolation style!!

```
let customer = { firstName: 'Foo', lastName: 'Kim' };
let message = `Hello ${customer.firstName} ${customer.lastName}!!`;
console.log(message);
```

As you can see, this kind of interpolation using template literals makes our code much more readable and cleaner!

Multiline Interpolation

Another great functionality of backticks is being able to easily add **new lines** in the same string (meaning the string can span 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. With other kinds of strings that would cause a syntax error.

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 double quotes 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 apostrophes inside single quote strings:

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

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

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

> ⭐ 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
```

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.

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 nth 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 also access characters inside of strings with square brackets and 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 occurrence, 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)); // <== 1
```

What we did was passing a second parameter, which represents a value where the first occurrence appeared (it was 6) + 1. So we are telling the method to skip the positions from 0 to 7 and keep looking for the occurrence 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 occurrences 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
```

Summary

Summary

In this lesson, we learned how to declare, use, and manipulate numbers and strings, two primitive data types. Also, we got familiar with some of the inherited number and string methods as well as with some of the new ones (introduced with ES6 updates).