How to run JavaScript code

Most of the JavaScript we will be writing in the Foundations course will be run via the browser. On the NodeJS path we will run JavaScript outside of the browser environment.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>
            Page Title
        </title>
    </head>
    <body>
        <script>
                // Prints message to the browser console
            console.log("Hello, World!")
        </script>
    </body>
</html>
```

`console.log()` is the command to print something to the developer console in your browser. Use this to print the results from any of the following articles/exercises to the console.

Another way to include JavaScript in a webpage is through an external script. This is very similar to linking external CSS docs to your website.

```
<script src="javascript.js"></script>
```

External JavaScript files are used for more complex scripts.

Variables

Think of variables as "storage containers" for data in your code.

Until recently there was only one way to create a variable in JavaScript - the `var` statement. But in the newest JavaScript versions we have two more ways - `let` and `const`.

https://javascript.info/variables

Variables

Most of the time, a JavaScript application needs to work with information. Here are two examples:

1. An online shop - the information might include goods being sold and a shopping cart.
2. A chat application - the information might include users, messages, and much more.

Variables are used to store this information.

A variable

A variable is a "named storage" for data.

To create a variable in JavaScript, use the `let` keyword.

The statement below creates (declares) a variable with the name "message":

```
let message;
```

Now, we can put some data into it.

```
let message;

message = 'Hello'
```

The string is now saved into the memory area associated with the variable. We can access it using the variable name:

```
let message;
message = 'Hello!';

// Show the variable content in a pop-up window
alert(message);
```

We can combine the first two lines:

```
let message = 'Hello!';
alert(message);
```

We can also declare multiple variables in one line:

```
let user = 'John', age = 25, message = 'Hello';
```

But it is recommended to keep 1 variable per line:

```
let user = 'John';
let age = 25;
let message = 'Hello';
```

Some people also define multiple variables in multiline styles:

```
let user = 'John',
    age = 25,
    message = 'Hello';
```

or

```
let user = 'John'
  , age = 25
  , message = 'Hello';
```

You can choose your own style since these all do the same thing.

The `var` keyword is *almost* the same as `let`. It also declares a variable but there are subtle differences. We will explore them later.

A real-life analogy

The variable `message` can be imagined as a box labeled `"message"` with the value `"Hello!"` in it.

We can change the value of the variable as many times as we want:

```
let message;

message = 'Hello';

message = 'World';
```

```
alert(message);
```

This replaces the value of message with a new value.

We can also copy data from one variable into another:

```
let hello = 'Hello world!';

let message;

message = hello;

// These will print the same thing
alert(hello);
alert(message);
```

Don't declare variables more than once.

Pure functional programming languages like Haskell forbid changing variable values. Once the value is stored "in the box", it's there forever. If we need to store something else, we need to create a new box to hold it.

Variable naming

There are two limitations on variable names in JavaScript:

1. The name must contain only letters, digits, or the symbols `$` and `_`
2. The first character must not be a digit

Examples of valid names:

```
let userName;
let test123;
```

When the name contains multiple words, camelCase is used.

`'$'` and `'_'` have no special meaning.

Thus, these names are vaid:

```
let $ = 1;
let _ = 2;

// 3
alert($ + _);
```

Examples of incorrect variable names:

```
let 1a; // Cannot start with a digit
let my-name; // hyphens are not allowed in the name of a variable
```

Case matters (`apple` is not the same as `APPLE`).

It is possible to use any language:

```
let имя = '...';
let 我 = '...';
```

But there is an international convention to use English in variable names.

There is a list of reserved words which cannot be used as variable names since they are used by the language itself.

For example, `let`, `class`, `return`, and `function` are reserved so the following are errors:

```
let let = 5;
let return = 5;
```

Normally, we need to define a variable before using it. But in the old times, it was technically possible to create a variable by a mere assignment of the value without using `let`. This still works now if we don't put `use strict` in our scripts to maintain compatibility with old scripts.

```
// note no "use strict" in this example

num = 5; // the variable "num" is created if it didn't exist

alert(5); // 5
```

This is bad practice and would cause an error in strict mode:

```
"use strict"

num = 5; // error: num is not defined
```

Constants

To declare a constant (unchanging) variable, use `const` instead of `let`:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, can't reassign the constant
```

Use const when you're sure that you won't be changing the variable's value.

Uppercase constants

There is a widespread practice to use constants as aliases for difficult-to-remember values that are known prior to execution.

Such constants are named using capital letters and underscores.

For instance, let's make constants for colors in so-called "web" (hexadecimal) format:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ... when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // "FF7F00"
```

Benefits:

- `COLOR_ORANGE` is much easier to remember than `"#FF7F00"`.
- It is much easier to mistype `"#FF7F00"` than `COLOR_ORANGE`.
- When reading the code, `COLOR_ORANGE` is much more meaningful than `"#FF7F00"`.

When should we use capitals for a constant and when should we name it normally? Let's make that clear.

For constants whose values are known prior to execution time (like a hex value for red), you should use the capital letter and underscore convention. For constants that are *calculated* in run-time, use the standard camelCase convention.

For instance:

```
const pageLoadTime = /* time taken by a webpage to load */
```

The value of `pageLoadTime` is not known prior to the page load, so it's named normally. But it's still a constant because it doesn't change after assignment.

In other words, capital-named constants are only used for aliases for "hard-coded" values.

Name things right

A variable name should have a clean, obvious meaning, describing the data that it stores.

Some good-to-follow rules are:

- Use human-readable names like `userName` or `shoppingCart`.
- Stay away from abbreviations or short names like `a`, `b`, `c` unless you really know what you're doing.
- Make names maximally descriptive and concise. Examples of bad names are `data` and `value`. Only use them if the code makes it exceptionally clear which data or value the variable is referencing.
- Agree on terms with your team and in your own mind. If a site visitor is called a "user" then we should name related variables `currentUser` or `newUser` instead of `currentVisitor` or `newManInTown`.

When should we reuse vs. create a new variable? If the name of a created variable wouldn't make sense with a certain new value, don't reuse it. Just create a new variable.

---

1. https://www.w3schools.com/js/js_arithmetic.asp

Numbers

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Remainder)
++	Increment
--	Decrement

`document.getElementById("demo").innerHTML = x;`
accesses the inner HTML of the HTML element with Id "`demo`" and sets it to the value of `x`.

`x ** y` is the same as `Math.pow(x, y)`

When many operations have the same precedence (like addition and subtraction or multiplication and division), they are computed from left to right usually (exponentiation like is right to left (`2 ** 3 ** 2` is `512`)).

---

https://www.w3schools.com/js/js_numbers.asp

JavaScript Numbers

JavaScript has only one type of number. Numbers can be written with or without decimals.

Extra large or extra small numbers can be written with scientific (exponent) notation:

```
let x = 123e5;  // 12300000
let y = 123e-5; // 0.00123
```

JavaScript Numbers are Always 64-bit Floating Point

Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point, etc.

Javascript numbers are always stored as double precision floating point numbers, following the international IEEE 754 standard.

This format stores numbers in 64 bits, where the fraction is stored in 52 bits, the exponent in 11, and the sign in 1.

Integer Precision

Integers (numbers without a period or exponent notation) are accurate up to 15 digits (because log_10(2^52) ~ 15.65...):

```
let x = 999999999999999;  // x will be 999999999999999
let y = 9999999999999999; // y will be 10000000000000000
```

`y` cannot be that precise so with imprecise precision it's approximately `10000000000000000` which is `1` more than `y`

Floating Precision

Floating point arithmetic is not always 100% accurate:

```
let x = 0.2 + 0.1; // 0.30000000000000004
```

To solve this problem, it helps to multiply and divide:

```
let x = (0.2 * 10 + 0.1 * 10) / 10;
```

Adding Numbers and Strings

JavaScript uses the + operator for both addition and concatenation.

Numbers are added. Strings are concatenated.

If you add two numbers: the result will be a number.

If you add two strings, the result will be a string concatenation.

If you add a number and a string, the result will be a string concatenation.

```
let x = 10;
let y = "20";
```

```
let z = x + y;  // 1020
```

If you add a string and a number, the result will be a string concatenation.

```
let x = "10";
let y = 20;
let z = x + y;  // 1020
```

A common mistake is to expect this result to be 30:

```
let x = 10;
let y = 20;
let z = "The result is: " + x + y;
```

First, the string concatenates with x to produce a string which then concatenates with y. So you get:
"The result is: 1020"

A common mistake is to expect this result to be 102030:

```
let x = 10;
let y = 20;
let z = "30";
let result = x + y + z;
```

The x + y first results in 30 and then 30 + z is a number concatenating with string resulting in: 3030

Numeric Strings

As seen, JavaScript strings can have numeric content:

```
let x = 100;    // x is a number
let y = "100";  // y is a string
```

JavaScript will try to convert string to numbers in all numeric operations (except addition because + concatenates when involving a string):

```
let x = "100";
let y = "10";
let z = x / y;  // Will work: 10
z = x * y;      // Will work: 1000
z = x - y;      // will Work: 90
z = x + y;      // Will not work like the others: 10010
```

NaN - Not a Number

`NaN` is a JavaScript reserved word indicating that a number is not a legal number.

Trying to do arithmetic with a non-numeric string will result in `NaN` (Not a Number):

```
let x = 100 / "Apple"; // NaN
```

However, if the string is numeric the result will be a number:

```
let x = 100 / "10"; // 10
```

`isNaN()` tests if a value is not a number:

```
let x = 100 / "Apple";
isNaN(x);               // true
```

If you use `NaN` in a mathematical operation, the result will also be `NaN`:

```
let x = NaN;
let y = 5;
let z = x + y;  // NaN
```

Or it might result in a concatenation if added with a string:

```
let x = NaN;
let y = "5";
let z = x + y;  // NaN5
```

`NaN` is a number (paradoxically); `typeof NaN` returns `number`:

```
typeof NaN; // number
```

Infinity

`Infinity` (or `-Infinity`) is the value JavaScript will return if you calculate a number outside the largest possible number.

```
let myNumber = 2;
// Will break out after a few iterations
while (myNumber != Infinity) {
      my_number = my_number * my_number;
}
```

Divison by 0 (zero) also generates `Infinity`:

```
let x = 2 / 0;  // Infinity
let y = -2 / 0; // -Infinity
```

`Infinity` is a number: `typeof Infinity` returns `number`.

```
typeof Infinity; // number
```

Hexadecimal

JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

`let x = 0xFF;`

Never write a number with a leading zero (like 07).
Some JavaScript versions interpret numbers as octal if they written with a leading zero.

By default, JavaScript displays numbers as **base 10** decimals.

But you can use the `toString()` method to output numbers from **base 2** to **base 36**.

```
let myNumber = 32;
myNumber.toString(36); // w
myNumber.toString(32); // 10
myNumber.toString(16); // 20
myNumber.toString(12); // 28
myNumber.toString(10); // 32
myNumber.toString(8);  // 40
myNumber.toString(2);  // 10000
```

JavaScript Numbers as Objects

Normally JavaScript numbers are primitive values created from literals:

`let x = 123; // typeof x -> number`

But numbers can also be defined as objects with the keyword `new`:

`let y = new Number(123); // typeof y -> object`

In general, **do not** create Number objects.

The `new` keyword complicates the code and slows down execution speed.

Number Objects can produce unexpected results:

```
let x = 500;
let y = new Number(500);
x == y; // true
x === y; // false
```

Comparing two **distinct** JavaScript objects **always** returns **false**.

```
let x = new Number(500);
let y = new Number(500);
x == y;                 // false
x === y;                // false
```

Comparing two JavaScript objects that refer to the same object returns true.

```
let x = new Number(500);
let y = x;
x == y;                 // true
x === y;                // true
```

Basic math in JavaScript - numbers and operators

1. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math

Useful Number methods

The JavaScript developer console is similar to the line-by-line Python console interpreter.

To round your number to a fixed number of decimal places, use the `toFixed()` method:

```
const lotsOfDecimal = 1.766584958675746364;
lotsOfDecimal; // 1.766584958675746364
const twoDecimalPlaces = lotsOfDecimal.toFixed(2);
twoDecimalPlaces; // 1.77
```

Converting to number data types

Sometimes you might end up with a number that is stored as a string type, which make it difficult to perform calculations with it. This most commonly happens when data is entered into a `form` input, and the input type is `text`. You can pass the string value into the `Number()` constructor to return a number version of the same value.

```
let myNumber = "74";
myNumber += 3;' // 743
myNumber = Number(myNumber) + 3; // 77
```

Comparison Operators

`===` tests for strict equality and `!==` tests for strict non-equality. This means they check that the value *and* the type match. `==` and `!=` return true if the values are the same, regardless of type.

```
10 == "10"; // true
10 === "10"; // false
```

Also, `===` tests for reference equality so:

```
let num1 = new Number(500);
let num2 = 500;
console.log(num1 == num2); // true
console.log(num1 === num2); // false
```

```
let num1 = new Number(500);
let num2 = num1;
console.log(num1 == num2); // true
console.log(num1 === num2); // true
```

Conditions use booleans to branch and thus make decisions in our code. For example, booleans can be used to:

- Display the correct text label on a button depending on whether a feature is turned on or off
- Display a game over message if a game is over or a victory message if the game has been won
- Display the correct seasonal greeting depending on what holiday season it is
- Zoom a map in or out depending on what zoom level is selected

For example, toggling between two states:

```
const btn = document.querySelector("button");
const txt = document.querySelector("p");

btn.addEventListener("click", updateBtn);

function updateBtn() {
    if (btn.textContent === "Start machine") {
        btn.textContent = "Stop machine";
        txt.textContent = "The machine has started!";
    }
    else {
        btn.textContent = "Start machine;"
                txt.textContent = "The machine is stopped.";
    }
}
```

---

3. https://javascript.info/operators

Basic operators, maths

Terms: "unary", "binary", "operand"

The unary operator `-` reverses the sign of a number.

```
let x = 1;
x = -x;
```

```
alert(x); // -1
```

Numeric conversion, unary +

The unary `+` transforms operands that are not numbers into numbers!

```
let x = 1;
alert(+x);      // 1

let y = -2;
alert(+y);      // -2

alert(+true);   // 1
alert(+"");     // 0
```

So it does the same thing as `Number(...)`, but it is shorter.

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. What if we want to sum them?

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

alert(+apples + +oranges);      // 5
// Same as: alert(Number(apples) + Number(oranges));
```

Assignment

Assignment = returns a value

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert(a); // 3
alert(c); // 0
```

`b + 1` gets assigned to `a`, then `3 - a` gets assigned to `c`.

This is a bad way to write things though.

Chaining assignments

```
let a, b, c;

a = b = c = 2 + 2;

alert(a); // 4
alert(b); // 4
alert(c); // 4
```

Modify-in-place

Assignment operators like `/=` have the same precedence as a normal assignment, so they run after most other calculations:

```
let n = 2;
n *= 3 + 5;     // Same as n *= 8
alert(n); // 16
```

Bitwise operators

Bitwise operators treat arguments as 32-bit integer numbers and work on the level of their binary representation

- AND (`&`)
- OR (`|`)
- XOR (`^`)
- NOT (`~`)
- LEFT SHIFT (`<<`)
- RIGHT SHIFT (`>>`)
- ZERO_FILL RIGHT SHIFT (`>>>`)

Comma

The comma operator allows use to evaluate several expressions, dividing them with a comma `,`. Each of them is evaluated but only the result of the last one is returned:

```
let a = (1 + 2, 3 + 4);
alert(a);  // 7
```

The `1 + 2` is executed but not used. Note that the comma operator has very low precedence (even lower than assignment), so without the parentheses above, the code would effectively be:

`(a = 1 + 2), 3 + 4`

Commas are usually used to do several actions in one line like in a for loop intialization:

```
for (a = 1, b = 3, c = a * b; a < 10; a++) {
    ...
}
```

Exercise: Type conversions

```
"" + 1 + 0 -> "10" // A string concatenation followed by another

"" - 1 + 0 -> -1 // The empty string, "", is converted to 0 and is
subtracted by 1 and added with 0

true + false -> 1 // true is converted to 1 and false is converted to 0

6 / "3" -> 2

"2" * "3" -> 6

4 + 5 + "px" -> "9px"

"$" + 4 + 5 -> "$45"

"4" - 2 -> 2

"4px" - 2 -> NaN

"  -9  " + 5 -> "  -9  5" // String concatenation

"  -9  " - 5 -> -14 // Subtraction always converts to numbers, so it makes "
-9  " a number, -9
```

```
null + 1 -> 1 // null becomes 0 after the numeric conversion

undefined + 1 -> NaN  // undefined becomes NaN after numeric conversion

" \t \n" - 2 -> -2    // Whitespaces are trimmed when converted to a number.
So equivalent to "" - 2 which is 0 - 2 = -2
```