Basic Javascript

History of JS

- JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.
- ECMAScript is the official name of the language.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- Since 2016 new versions are named by year (ECMAScript 2016 / 2017 / 2018).

Why JS

- JavaScript is the world's most popular programming language.
- JavaScript is the programming language of the Web.
- JavaScript is easy to learn.

How to use JS

JS code must be inserted between <script> and </script> tags. You can place any number of scripts in an HTML document. Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both. Placing scripts at the bottom of the <body> element improves the display speed, because script interpretation slows down the display. Scripts can also be placed in external files. External scripts are practical when the same code is used in many different web pages. JavaScript files have the file extension .js. To use an external script, put the name of the script file in the src (source) attribute of a <script> tag. Ex.<script src="myScript.js"></script> You can place an external script reference in <head> or <body> as you like. The script will behave as if it was located exactly where the <script> tag is located. Placing scripts in external files has some advantages:

- It separates HTML and JS code.
- It makes HTML and JavaScript easier to read and maintain.
- Cached JavaScript files can speed up page load.

Identifiers

General rules for constructing identifiers(case-sensitive) are:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must begin with a letter but can also begin with $ and _.
- Names are case sensitive (y and Y are different variables).
- Reserved words (like JavaScript keywords) cannot be used as names.

Declaration: var carName; //no value (undefined)

Initialization: carName = "Volvo";

It's a good programming practice to declare all variables at the beginning of a script. A variable declared without a value will have the value undefined. If you re-declare a JavaScript variable, it will not lose its value.

Comments

JS comments can be used to explain JavaScript code, and to make it more readable.

JS comments can also be used to prevent execution, when testing alternative code.

There are two types:

1. **Single Line Comments**: It start with //. Any text between // and the end of the line will be ignored by JS (will not be executed). Example:

   ```
   // no value (undefined)
   var carName;
   ```

2. **Multi-line Comments**: It start with /* and end with */. Any text between /* and */ will be ignored by JS. This example uses a multi-line comment (a comment block) to explain the code. Example:

   ```
   /* A variable declared without a value will have the value undefined. */
   var carName;
   ```

Data Types

JavaScript variables can hold different data types: numbers, strings, objects and more. In programming, data types is an important concept. To be able to operate on variables, it is important to know something about the type. JavaScript has dynamic types. This means that the same variable can be used to hold different data types.

- **Strings**: A string (or a text string) is a series of characters like "John Doe". Strings are written with quotes. You can use single or double quotes. Example:

```
let carName1 = "Volvo XC60";   // Using double quotes
let carName2 = 'Volvo XC60';   // Using single quotes
let answer1 = "It's alright";   // Single quote inside double quotes
let answer2 = "He is called 'Johnny'";   // Single quotes inside double quotes
let answer3 = 'He is called "Johnny"';   // Double quotes inside single quotes
```

- **Numbers**: JS has only one type of numbers. Numbers can be written with, or without decimals. Example:

```
let x1 = 34.00;   // Written with decimals
let x2 = 34;   // Written without decimals
```

Data Types(contd.)

Extra large or extra small numbers can be written with scientific (exponential) notation. Example:

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

- **Booleans**: They can only have two values: true or false. Example:

```
let x = 5, y = 5, z = 6;
(x == y)   // Returns true
(x == z)   // Returns false
```

- **Arrays**: JS arrays are written with square brackets. Array items are separated by commas. The following code declares (creates) an array called cars, containing three items (car names). Example:

```
const cars = ["Saab", "Volvo", "BMW"];
```

- **Objects**: JavaScript objects are written with curly braces {}. Object properties are written as name:value pairs, separated by commas. Example:

```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Operators

Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (same result as Math.pow(x,y))
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

Logical Operators

&&	logical and
||	logical or
!	logical not

Operators(contd.)

Assignment Operators

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

Operators(contd.)

String Operators

+	concatenate strings with strings/numbers/boolean, etc.

Comparison Operators

==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?:	ternary operator
??	nullish coalescing operator

Operators(contd.)

Type Operators

typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

Bitwise Operators

Operator	Description	Example	Same as	Result
	Decimal			
&	AND		5 & 1	
	0101 & 0001	0001		1
|	OR		5 | 1	
	0101 | 0001	0101		5
~	NOT		~ 5	
	~0101	1010		10
^	XOR		5 ^ 1	
	0101 ^ 0001	0100		4

Operators(contd.)

TypeOf: It returns the type of a variable or an expression in a string format. Types:

- typeof operator can return one of these primitive types:
  - string for strings.
  - number for numbers.
  - boolean for true/false.
  - undefined for undefined
- typeof operator can return one of two complex types:
  - function for functions.
  - object for objects, arrays, and null.

```
typeof ""                          // Returns "string"
typeof (3)                         // Returns "number"
typeof +                           // Returns "error". Operators have no data type
typeof true                        // Returns "boolean"
typeof [1, 2, 3, 4]                // Returns "object"
typeof function myFunc(){}         // Returns "function"
```

Operators(contd.)

```
typeof {name:'John', age:34}        // Returns "object"
var car;                                                    // Value is undefined, type is undefined
car = undefined;                                    // Value is undefined, type is undefined
person = null;                                              // Now value is null, hence the type is object
```

Note: undefined and null are equal in value but different in type.

InstanceOf: It returns true if the specified object is an instance of the specified object. It tests the presence of constructor.prototype in object's prototype chain. Ex.

```
var cars = ["Saab", "Volvo", "BMW"];
cars instanceof Array;                          // Returns true
cars instanceof Object;                         // Returns true
cars instanceof String;                         // Returns false
cars instanceof Number;                         // Returns false
```

Operators(contd.)

Nullish Coalescing(??)

It provides a short syntax for selecting a first "defined" variable from the list. It's extensively used to assign default values to variables. The result of a ?? b is: a if it's not null or undefined, b, otherwise.

So, x = a ?? b is a short equivalent to: x = (a !== null && a !== undefined) ? a : b;

The OR || operator can be used in the same way as ??. The important difference is that || returns the first truthy value vs ?? returns the first defined value. It's importance comes into play in situations when we deal with values which are treated as falsy values but are not undefined or null. Eg.

```
let height = 0;
console.log(height || 100); // 100
console.log(height ?? 100); // 0
```

The precedence of the ?? operator is rather low. So ?? is evaluated after most other operations, but before = and ?. Hence, if we need to choose a value with ?? in a complex expression, then consider adding parentheses. Ex. `let area = (height ?? 100) * (width ?? 50);`

If we omit parentheses then it would run it as: `let area = height ?? (100 * width) ?? 50;`

Due to safety reasons, it's forbidden to use ?? together with && and || operators. The code below triggers a syntax error: `let x = 1 && 2 ?? 3; // Syntax error`

Use explicit parentheses to work around it:

```
let x = (1 && 2) ?? 3; // Works
console.log(x); // 2
```

Type Conversion

JS variables can be converted to a new variable and another data type.

- By the use of a JavaScript function
  - global method String() & primitive data type method toString() can convert any number, boolean & date to a string.
  - global method Number() can convert strings, booleans & dates to numbers. Empty strings convert to 0. Anything else converts to NaN (Not a Number).
  - unary + operator can be used to convert a variable to a number. Ex.

```
var y = "5";          // y is a string
var x = + y;          // x is a number
```

- Automatically by JavaScript itself
  - When JavaScript tries to operate on a "wrong" data type, it will try to convert the value to a "right" type. Ex.

```
5 + null              // returns 5 because null is converted to 0
"5" + null            // returns "5null" because null is converted to "null"
"5" + 2                       // returns "52" because 2 is converted to "2"
"5" - 2                       // returns 3 because "5" is converted to 5
"5" * "2"                     // returns 10 because "5" and "2" are converted to 5 and 2
```

Implicit Type Conversion

Original Value	Converted to Number	Converted to String	Converted to Boolean
false	0	"false"	false
true	1	"true"	true
0	0	"0"	false
1	1	"1"	true
"0"	0	"0"	**true**
"000"	0	"000"	**true**
"1"	1	"1"	true
NaN	NaN	"NaN"	false
Infinity	Infinity	"Infinity"	true
-Infinity	-Infinity	"-Infinity"	true
""	**0**	""	**false**
"20"	20	"20"	true
"twenty"	NaN	"twenty"	true
[ ]	**0**	""	true
[20]	**20**	"20"	true
[10,20]	NaN	"10,20"	true
["twenty"]	NaN	"twenty"	true
["ten","twenty"]	NaN	"ten,twenty"	true
function(){}	NaN	"function(){}"	true
{ }	NaN	"[object Object]"	true
null	**0**	"null"	false
undefined	NaN	"undefined"	false

Conditions

JS conditional statements are used to perform different actions based on different conditions. Kinds:

- Use if to specify a block of code to be executed, if a specified condition is true
- Use else to specify a block of code to be executed, if the same condition is false
- Use else if to specify a new condition to test, if the first condition is false
- Use switch to specify many alternative blocks of code to be executed

The if Statement

Use the if statement to specify a block of JavaScript code to be executed if a condition is true. Syntax:

```
if (condition) {
  //  block of code to be executed if the condition is true
}
```

Note that if is in lowercase letters. Uppercase letters (If or IF) will generate a JavaScript error. Example: Make a "Good day" greeting if the hour is less than 18:00:

```
if (hour < 18) {
  greeting = "Good day";
}
```

Conditions(contd.)

The else Statement

Use the else statement to specify a block of code to be executed if the condition is false.

```
if (condition) {
  //  block of code to be executed if the condition is true
} else {
  //  block of code to be executed if the condition is false
}
```

Example: If the hour is less than 18, create a "Good day" greeting, otherwise "Good evening":

```
if (hour < 18) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

Conditions(contd.)

The else if Statement

Use the else if statement to specify a new condition if the first condition is false. Syntax:

```
if (condition1) {
  //  block of code to be executed if condition1 is true
} else if (condition2) {
  //  block of code to be executed if the condition1 is false and condition2 is true
} else {
  //  block of code to be executed if the condition1 is false and condition2 is false
}
```

Example: If time is less than 10:00, create a "Good morning" greeting, if not, but time is less than 20:00, create a "Good day" greeting, otherwise a "Good evening":

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

Switch

The switch statement is used to perform different actions based on different conditions.

Syntax:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Switch(contd.)

This is how it works. The switch expression is evaluated once. The value of the expression is compared with the values of each case. If there is a match, the associated block of code is executed. If there is no match, the default code block is executed. Example:

```
switch (new Date().getDay()) {
  case 0:
    day = "Sunday"; break;
  case 1:
    day = "Monday"; break;
  case 2:
     day = "Tuesday"; break;
  case 3:
    day = "Wednesday"; break;
  case 4:
    day = "Thursday"; break;
  case 5:
    day = "Friday"; break;
  case 6:
    day = "Saturday";
}
```

Switch(contd.)

The break Keyword

When JavaScript reaches a break keyword, it breaks out of the switch block. This will stop the execution inside the switch block. It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway. Note: If you omit the break statement, the next case will be executed even if the evaluation does not match the case.

The default Keyword

The default keyword specifies the code to run if there is no case match. The default case does not have to be the last case in a switch block. If default is not the last case in the switch block, remember to end the default case with a break. Example

```
switch (new Date().getDay()) {
  default:
    text = "Looking forward to the Weekend";
    break;
  case 6:
    text = "Today is Saturday";
    break;
  case 0:
    text = "Today is Sunday";
}
```

Switch(contd.)

Common Code Blocks

Sometimes you will want different switch cases to use the same code. In this example case 4 and 5 share the same code block, and 0 and 6 share another code block. Example:

```
switch (new Date().getDay()) {
  case 4:
  case 5:
    text = "Soon it is Weekend";
    break;
  case 0:
  case 6:
    text = "It is Weekend";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

Switch(contd.)

Switching Details

If multiple cases matches a case value, the first case is selected. If no matching cases are found, the program continues to the default label. If no default label is found, the program continues to the statement(s) after the switch.

Strict Comparison

Switch cases use strict comparison (===). The values must be of the same type to match. A strict comparison can only be true if the operands are of the same type. In this example there will be no match for x. Example:

```
let x = "0";
switch (x) {
  case 0:
    text = "Off";
    break;
  case 1:
    text = "On";
    break;
  default:
    text = "No value found";
}
```

Loops

JS loops can execute a block of code a number of times. Kinds:

- for - loops through a block of code a number of times.
- for/in - loops through the properties of an object. (Will be covered in objects topic)
- for/of - loops through the values of an iterable object. It lets you loop over Arrays, Strings, Maps, NodeLists, and more. (Will be covered in arrays topic)
- while - loops through a block of code while a specified condition is true.
- do/while - will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Loops(contd.)

The For Loop

The for loop has the following syntax:

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

Example

```
for (let i = 0; i < 5; i++) {
  text += "The number is " + i + "<br>";
}
```

From the example above, you can read:

Statement 1 sets a variable before the loop starts (let i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5).

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

Loops(contd.)

The While Loop

The while loop loops through a block of code as long as a specified condition is true. Syntax

```
while (condition) {
  // code block to be executed
}
```

In the following example, the code in the loop will run, over and over again, as long as a variable (i) is less than 10:

Example:

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

Note: If you forget to increase the variable used in the condition, the loop will never end. This will crash your browser.

Loops(contd.)

The Do While Loop

The do while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true. Syntax:

```
do {
  // code block to be executed
}
while (condition);
```

The example below uses a do while loop. The loop will always be executed at least once, even if the condition is false, because the code block is executed before the condition is tested. Example:

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

Break & Continue

The Break Statement

The break statement "jumps out" of a loop. It was used to "jump out" of a switch() statement. The break statement can also be used to jump out of a loop. Example:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { break; }
  text += "The number is " + i + "<br>";
}
```

In the example above, the break statement ends the loop ("breaks" the loop) when the loop counter (i) is 3.

The Continue Statement

The continue statement "jumps over" one iteration in the loop. The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop. This example skips the value of 3. Example:

```
for (let i = 0; i < 10; i++) {
  if (i === 3) { continue; }
  text += "The number is " + i + "<br>";
}
```

Functions

JS function is a block of code designed to perform a particular task. It's defined with the function keyword, followed by a name, followed by parentheses (). The parentheses may include parameter names separated by commas. Inside the function, the arguments (the parameters) behave as local variables. Ex.

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Accessing a function without () will return the function definition instead of the function result. The code inside the function will execute when "something" invokes (calls) the function. Variables declared within a JavaScript function, become LOCAL to the function. Local variables can only be accessed from within the function. When JavaScript reaches a return statement, the function will stop executing. If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement. Functions often compute a return value. The return value is "returned" back to the "caller".

Functions(contd.)

A function can also be created using a new operator called the “new Function“ syntax. It’s environment is set to reference not the current Lexical Environment, but the global one. So, such a function doesn’t have access to outer variables, only to the global ones. Ex.

```
function getFunc() {
  let value = "test";
  let func = new Function('console.log(value)');
  return func;
}
getFunc()(); // error: value is not defined
```

Self-Invoking Functions are those function expressions which will execute automatically if the expression is followed by (). Ex.

```
(function () { // anonymous self-invoking function
  var x = "Hello!!";  // I will invoke myself
})();
```

Functions(contd.)

JavaScript functions can best be described as objects. JavaScript functions have both properties and methods. The arguments.length property returns the number of arguments received when the function was invoked. Ex.

```
function myFunction(a, b) {
  return arguments.length; // 2
}
```

Named Function Expression(NFE), is a term for function expressions that have a name. NFE allows the name function to reference itself internally but the name function is not visible outside of the function. Ex.

```
let sayHi = function func(who) {
  if (who) {
    console.log(`Hello, ${who}`);
  } else {
    func("Guest"); // use func to re-call itself
  }
};
sayHi(); // Hello, Guest
func(); // Error, func is not defined (not visible outside of the function)
```

This comes handy in cases when there is a recursive call to the function within the body. This is because if the variable with function expression is reassigned and is also used inside then that becomes a problem.

Functions(contd.)

Function Parameters

These are the names listed in the function definition. Function arguments are the real values passed to (and received by) the function. JS function definitions do not specify data types for parameters, do not perform type checking on the passed arguments & do not check the number of arguments received.

Default Parameters

If a function is called with missing arguments (less than declared), the missing values are set to: undefined. Sometimes this is acceptable, but sometimes it is better to assign a default value to the parameter. Ex.

```
function (a=1, b=1) {
  // function code
}
```

Functions(contd.)

Arguments Object

JavaScript functions have a built-in object called the arguments object that contains an array of the arguments used when the function was called (invoked). If a function is called with too many arguments (more than declared), these arguments can be reached using the arguments iterable object.

Arguments are passed by value: The function only gets to know the values, not the argument's locations. If a function changes an argument's value, it does not change the parameter's original value. Changes to arguments are not visible (reflected) outside the function.

Pure Functions

These are the functions which when given the same input will always return the same output produce no side-effects. Example:

```
var tax = 20;
function calculateGST( productPrice ) {
    return productPrice * (tax / 100) + productPrice;
}
```

Functions(contd.)

Arrow functions allow us to write shorter function syntax. Ex.

```
const hello = function() {
  return "Hello World!";
}
```

```
const hello = () => {
  return "Hello World!";
}
```

Using const is safer than using var, because a function expression is always constant value. If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword. Ex. `hello = () => "Hello World!";`

If you have parameters, you pass them inside the parentheses. Ex.

```
hello = (val) => "Hello " + val;
```

In fact, if you have only one parameter, you can skip the parentheses as well. Ex.

```
hello = val => "Hello " + val;
```

Functions(contd.)

Arrow functions have no binding of this. In regular functions, this keyword represents the object that called the function, which could be the window, the document, a button or whatever. With arrow functions, this keyword always represents the object that defined the arrow function. Hence, Sometimes the behavior of regular functions is what you want, if not, use arrow functions.

Arrow functions are not hoisted. They must be defined before they are used. They also do not have the “arguments” variable like a normal function.

Lambda Function

Lambda means function expression used as data. They are simply expressions that create functions. This is really important for a programming language to support first-class functions which basically means:

Passing functions as arguments to other functions. Ex. `arr.sort((a, b) => a-b);`

Assigning them to variables. Ex. `const hello = () => "Hello World!";`

Scope

Scope determines the accessibility (visibility) of variables/functions. Types of scope: Global scope & Local scope.

Global Scope

A variable declared outside a function, becomes GLOBAL. A global variable has global scope: All scripts and functions on a web page can access it. If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable. Ex.

```
// code here can use carName
function myFunction() {
  carName = "Volvo";
}
```

In "Strict Mode", undeclared variables are not automatically global.

With JavaScript, the global scope is the complete JavaScript environment. In HTML, the global scope is the window object. All global variables belong to the window object. Ex.

```
var carName = "Volvo"; // code here can use window.carName
```

Do NOT create global variables unless you intend to. Your global variables (or functions) can overwrite window variables (or functions). Any function, including the window object, can overwrite your global variables and functions.

Scope(contd.)

Local Scope

It has 2 types: Function scope & Block scope

Function Scope

Variables declared within a JavaScript function, become local to the function. Local variables have Function scope: They can only be accessed from within the function. Local variables are created when a function starts, and deleted when the function completes. Function arguments (parameters) work as local variables inside functions. Ex.

```
// code here can NOT use carName & param
function myFunction(param) {
  let carName = "Volvo";
  // code here CAN use carName & param
}
// code here can NOT use carName & param
```

Scope(contd.)

Block Scope

Before ES6 (2015), JavaScript had only Global Scope and Function Scope. ES6 introduced two important new JavaScript keywords: let and const. These two keywords provide Block Scope in JavaScript. Variables declared inside a { } block cannot be accessed from outside the block. Example:

```
{
  let x = 2;
  const y = 2;
}
// x & y can NOT be used here
```

Variables declared with the var keyword can NOT have block scope. Variables declared inside a { } block can be accessed from outside the block. Example:

```
{
  var x = 2;
}
// x CAN be used here
```

Let

Let is used to define a block scope variable. Variables declared inside a block {} can not be accessed from outside the block.

Global variables defined with the var keyword belong to the window object. Ex.

```
var carName = "Volvo"; // code here can use window.carName
```

Global variables defined with the let keyword do not belong to the window object. Ex.

```
let carName = "Volvo"; // code here can not use window.carName
```

Redeclaring a var variable with let & vice versa, in the same scope, or in the same block, is not allowed. Ex.

```
{
  var x = 4;   // Allowed
  let x = 5    // Not allowed
}
```

```
{
  let x = 5    // allowed
  var x = 4;  // Not allowed
}
```

Variables defined with let are not hoisted to the top & can't be redeclared.

Const

Const behave like let variables, except they cannot be reassigned. It's a Block Scope variable. Variables declared inside a block {} can not be accessed from outside the block. Ex.

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

const variables must be assigned a value when declared otherwise an error. Ex.

```
const PI;    PI = 3.14159265359; // Error
```

const does NOT define a constant value. It defines a constant reference to a value. Because of this, we cannot change constant primitive values, but we can change the properties of constant objects. Hence, you can change the values of an array and properties of an object but you cannot reassign them. Ex.

```
// constant array
const cars = ["Saab", "Volvo", "BMW"];
cars[0] = "Toyota"; // change an element
cars.push("Audi"); // add an element
cars = ["Toyota", "Volvo", "Audi"]; // ERROR
```

```
// constant object
const car = {type:"Fiat", model:"500", color:"white"};
car.color = "red"; // change a property:
car.owner = "Johnson"; // add a property:
car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

Variables defined with const are not hoisted to the top & can't be redeclared.

Number

JS has only one type of number with or without decimals. They are stored as double precision floating point numbers. This format stores numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63.

Integers (numbers without a period or exponent notation) are accurate up to 15 digits. Ex.

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

The maximum number of decimals is 17, but floating point arithmetic is not always 100% accurate. Ex.

```
var x = 0.2 + 0.1; // x will be 0.30000000000000004
```

JS strings can have numeric content. JS will try to convert strings to numbers in all numeric operations. Ex.

```
var x = "100";              var y = "10"; var z = x / y;       // z will be 10
```

NaN is a JavaScript reserved word indicating that a number is not a legal number. NaN is a number: typeof NaN returns number. Ex.

```
var x = 100 / "Apple";  // x will be NaN (Not a Number)
```

Number(contd.)

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

Number Methods:

- **isNaN()** (global JavaScript function) can find out if a value is a number. Ex.
  isNaN(x); // returns true because x is Not a Number
- toString() method returns a number as a string.
- toExponential() returns a string, with a number rounded and written using exponential notation. Ex. var x = 9.656; x.toExponential(2);// returns 9.66e+0
  The parameter is optional. If you don't specify it, JavaScript will not round the number.
- toFixed() returns a string, with the number written with a specified number of decimals. Ex. var x = 9.656; x.toFixed(0); // returns 10
- toPrecision() returns a string, with a number written with a specified length. Ex. var x = 9.656; x.toPrecision(); // returns 9.656
- valueOf() returns a number as a number. Since, a number can be a primitive value (typeof = number) or an object (typeof = object). The valueOf() method is used internally in JavaScript to convert Number objects to primitive values. Ex. var x = 123; x.valueOf(); // returns 123 from variable x

Number(contd.)

- Number() (global JavaScript function) can be used to convert JavaScript variables to numbers. Ex.

```
Number(true); // returns 1
Number("10"); // returns 10
Number("10,33"); // returns NaN
```

- parseInt() parses a string and returns a whole number. Spaces are allowed. Only the first number is returned. Ex.

```
parseInt("10.33");      // returns 10
parseInt("10 20 30");   // returns 10
parseInt("years 10");   // returns NaN
```

- parseFloat() parses a string and returns a number. Spaces are allowed. Only the first number is returned. Ex.

```
parseFloat("10.33");     // returns 10.33
parseFloat("10 20 30");  // returns 10
parseFloat("years 10");  // returns NaN
```

Math

JS Math object allows you to perform mathematical tasks on numbers. Ex.

```
Math.PI; // returns 3.141592653589793
```

Unlike other global objects, the Math object has no constructor. Methods and properties are static.

Methods:

- Math.round(x) returns the value of x rounded to the nearest integer. Ex.

```
Math.round(4.9);    // returns 5
Math.round(4.7);    // returns 5
Math.round(4.4);    // returns 4
Math.round(4.2);    // returns 4
Math.round(-4.2);    // returns -4
```

- Math.pow(x, y) returns the value of x to the power of y. Ex.

```
Math.pow(8, 2);      // returns 64
```

- Math.sqrt(x) returns the square root of x. Ex.

```
Math.sqrt(64);      // returns 8
```

Math(contd.)

- Math.abs(x) returns the absolute (positive) value of x. Ex.

```
Math.abs(-4.7);     // returns 4.7
```

- Math.ceil(x) returns the value of x rounded up to the nearest integer. Ex.

```
Math.ceil(4.9);     // returns 5
Math.ceil(4.7);     // returns 5
Math.ceil(4.4);     // returns 5
Math.ceil(4.2);     // returns 5
Math.ceil(-4.2);     // returns -4
```

- Math.floor(x) returns the value of x rounded down to its nearest integer. Ex.

```
Math.floor(4.9);    // returns 4
Math.floor(4.7);    // returns 4
Math.floor(4.4);    // returns 4
Math.floor(4.2);    // returns 4
Math.floor(-4.2);    // returns -5
```

Math(contd.)

- Math.sin(x) returns the sine (a value between -1 and 1) of the angle x (given in radians). If you want to use degrees instead of radians, you have to convert degrees to radians. Angle in radians = Angle in degrees x PI / 180. Ex.

  ```
  Math.sin(90 * Math.PI / 180);     // returns 1 (the sine of 90 degrees)
  ```

- Math.cos(x) returns the cosine (a value between -1 and 1) of the angle x (given in radians). If you want to use degrees instead of radians, you have to convert degrees to radians. Angle in radians = Angle in degrees x PI / 180. Ex.

  ```
  Math.cos(0 * Math.PI / 180);     // returns 1 (the cos of 0 degrees)
  ```

- Math.min() and Math.max() can be used to find the lowest or highest value in a list of arguments. Ex.

  ```
  Math.min(1, 2, 3, 4);     // returns 1
  Math.max(1, 2, 3, 4);     // returns 4
  ```

- Math.random() returns a random number between 0 (inclusive), and 1 (exclusive). Ex.

  ```
  Math.random();     // returns 0.1
  ```

String

JS strings are used for storing and manipulating text. A JavaScript string is zero or more characters written inside quotes. You can use single or double quotes. Example:

```
let carName1 = "Volvo XC60";  // Double quotes
let carName2 = 'Volvo XC60';  // Single quotes
```

You can use quotes inside a string, as long as they don't match the quotes surrounding the string:

```
let answer1 = "It's alright";
let answer2 = "He is called 'Johnny'";
let answer3 = 'He is called "Johnny"';
```

Escape Characters

Because strings must be written within quotes, JavaScript will misunderstand this string:

```
let text = "We are the so-called "Vikings" from the north.";
```

The string will be chopped to "We are the so-called ". The solution to avoid this problem, is to use the backslash escape character. The backslash (\) escape character turns special characters into string characters. Example:

```
let text = "We are the so-called \"Vikings\" from the north.";
var x = 'It\'s alright.';
```

Note: Strings are immutable: Strings cannot be changed, only replaced.

String(contd.)

length property returns the length of a string. Example:

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
text.length;    // Will return 26
```

Methods:

- indexOf() method returns the index of (the position of) the first occurrence of a specified text in a string & returns -1 if the text is not found. Ex.

  ```
  let str = "Please locate where 'locate' occurs!";
  str.indexOf("locate")    // Returns 7
  ```

  It can accept a second parameter as the starting position for the search. Ex.

  ```
  str.indexOf("locate", 19)    // Returns 21
  ```

- lastIndexOf() method returns the index of the last occurrence of a specified text in a string & returns -1 if the text is not found. Ex. `str.lastIndexOf("locate")    // Returns 21`

  It can accept a second parameter as the starting position for the search. Since lastIndexOf() method searches backwards (from the end to the beginning), if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string. Ex. `str.lastIndexOf("locate", 15)    // Returns 7`

String(contd.)

- search() method searches a string for a specified value and returns the position of the match. Ex.

  ```
  let str = "Please locate where 'locate' occurs!";
  str.search("locate")     // Returns 7
  ```

  slice() extracts a part of a string and returns the extracted part in a new string. The method takes 2 parameters: the start position, and the end position (end not included). Ex.

  ```
  let str = "Apple, Banana, Kiwi";
  str.slice(7, 13)     // Returns Banana
  ```

  If a parameter is negative, the position is counted from the end of the string. Ex.

  ```
  str.slice(-12, -6)    // Returns Banana
  ```

  If you omit the second parameter, the method will slice out the rest of the string. Ex.

  ```
  str.slice(7);    // Returns Banana,Kiwi
  str.slice(-12)    // Returns Banana,Kiwi
  ```

- substring() is similar to slice() but cannot accept negative indexes. Ex.

  ```
  substring(7, 13)    // Returns Banana
  ```

  If you omit the second parameter, substring() will slice out the rest of the string. Ex.

  ```
  substring(7)    // Returns Banana,Kiwi
  ```

String(contd.)

- substr() method's second parameter specifies the length of the extracted part. Ex.

  ```
  let str = "Apple, Banana, Kiwi";
  str.substr(7, 6)    // Returns Banana
  ```

  If you omit the second parameter, substr() will slice out the rest of the string. Ex.

  ```
  str.substr(7)    // Returns Banana,Kiwi
  ```

  If the first parameter is negative, the position counts from the end of the string. Ex.

  ```
  str.substr(-4)    // Returns Kiwi
  ```

- replace() method replaces a specified value with another value in a string. It does not change the string it is called on. It returns a new string. By default, it is case sensitive & replaces only the first match. Ex.

  ```
  let text = "Please visit Microsoft and Microsoft!";
  let newText = text.replace("Microsoft", "W3Schools");  // Returns Please visit W3Schools and Microsoft!
  ```

  To replace case insensitive, use a regular expression with an /i flag (insensitive). Ex.

  ```
  let newText = text.replace(/MICROSOFT/i, "W3Schools");  // Returns Please visit W3Schools and Microsoft!
  ```

  To replace all matches, use a regular expression with a /g flag (global match). Ex.

  ```
  let newText = text..replace(/Microsoft/g, "W3Schools");  // Returns Please visit W3Schools and W3Schools!
  ```

String(contd.)

- replaceAll() method replaces all occurrences for a specified value with another value in a string. It does not change the string it is called on. It returns a new string. By default, it is case sensitive & replaces only the first match. To replace case insensitive, use a regular expression with an /i flag (insensitive). To replace all matches, use a regular expression with a /g flag (global match). Ex.

  ```
  let text = "Please visit Microsoft and Microsoft!";
  let newText = text..replaceAll("Microsoft", "W3Schools");  // Returns Please visit W3Schools and W3Schools!
  ```

- toUpperCase() converts a string to uppercase. Ex.

  ```
  let text1 = "Hello World!";       // String
  let text2 = text1.toUpperCase();  // text2 is text1 converted to upper
  ```

- toLowerCase() converts a string to lowercase. Ex.

  ```
  let text1 = "Hello World!";       // String
  let text2 = text1.toLowerCase();  // text2 is text1 converted to lower
  ```

- concat() joins two or more strings. Ex.

  ```
  let text1 = "Hello", text2 = "World";
  let text3 = text1.concat(" ", text2); // Returns Hello World
  ```

  It can be used instead of the plus operator. Ex.

  ```
  text = "Hello" + " " + "World!";
  text = "Hello".concat(" ", "World!"); // Returns Hello World
  ```

String(contd.)

- trim() method removes whitespace from both sides of a string. Ex.

  ```
  let text = "                Hello World!                ";
  text.trim()    // Returns "Hello World!"
  ```

- charAt() method returns the character at a specified index (position) in a string. Ex.

  ```
  let text = "HELLO WORLD";
  text.charAt(0)          // Returns H
  ```

- charCodeAt() method returns the unicode of the character at a specified index in a string. Ex.

  ```
  let text = "HELLO WORLD";
  text.charCodeAt(0)      // Returns 72
  ```

- property access [ ] on strings returns the character. Ex.

  ```
  let text = "HELLO WORLD";
  text[0]                 // returns H
  ```

  But it's a little unpredictable:

  - It makes strings look like arrays (but they are not)
  - If no character is found, [ ] returns undefined, while charAt() returns an empty string.
  - It is read only. str[0] = "A" gives no error (but does not work!)

String(contd.)

- split() converts a string to an array. Ex.

  ```
  let text = "Hello";
  text.split(",")                 // Split on commas
  text.split(" ")      // Split on spaces
  text.split("");  // Split in characters, returns ['H', 'e', 'l', 'l', 'o']
  ```

  If the separator is omitted, the returned array will contain the whole string in index [0]. If the separator is "", the returned array will be an array of single characters. Ex.

  ```
  text.split("");  // Split in characters, returns ['Hello']
  ```

Note: All string methods return a new string. They don't modify the original string.

Template literals (Template strings)

They are enclosed by the backtick (` `) character instead of double or single quotes. They can contain placeholders. These are indicated by the dollar sign and curly braces (${expression}). Ex.

```
let a = 5, b = 10;
console.log('Fifteen is ' + (a + b) + ' and not ' + (2 * a + b) + '.'); // "Fifteen is 15 and not 20." (concatenation approach)
console.log(`Fifteen is ${a + b} and not ${2 * a + b}.`); // "Fifteen is 15 and not 20." (Template literal approach)
```

Array

JS arrays are used to store multiple values in a single variable. Syntax:

var array_name = [item1, item2, ...];

Ex. const cars = ["Saab", "Volvo", "BMW"];

You access an array element by referring to the index number. Ex. var name = cars[0];

Index can also be used to change the value of array elements. Ex. cars[0] = "Opel";

length property of an array returns the length of an array (the number of array elements). Ex. fruits.length;

Don't use the new Array(). Use [] instead. The new keyword only complicates the code. Ex.

var points = new Array(40, 100); // Creates an array with two elements (40 and 100)

var points = new Array(40); // Creates an array with 40 undefined elements !!!!!

To recognize an Array, we can user:

Array.isArray() Ex. Array.isArray(fruits); // returns true

Instanceof Ex. fruits instanceof Array; // returns true

The JavaScript for of statement loops through the values of an iterable object. Syntax & Example:

```
let iterable; // Array,String,Maps,NodeList,etc.
for (variable of iterable) {
  // code block to be executed
}
```

```
const cars = ["BMW", "Volvo", "Mini"];
for (let x of cars) {
  text += x;
}
```

Array(contd.)

Methods:

- toString() converts an array to a string of (comma separated) array values. Ex.

  ```
  const fruits = ["Banana", "Orange", "Apple", "Mango"];
  console.log(fruits.toString()); // Banana,Orange,Apple,Mango
  ```

- join() also joins all array elements into a string. Ex.

  ```
  console.log(fruits.join(" , ")); // Banana , Orange , Apple , Mango
  ```

- pop() removes the last element from an array. It returns the "popped out" value. Ex.

  ```
  console.log(fruits.pop()); // Mango
  ```

- push() adds a new element to an array (at the end). It returns the new array length. Ex.

  ```
  console.log(fruits.push("Grapes")); // Banana,Orange,Apple,Mango,Grapes
  ```

- shift() removes the first array element and "shifts" all other elements to a lower index. It returns the value that was "shifted out". Ex.

  ```
  console.log(fruits.shift()); // Banana
  ```

Array(contd.)

- unshift() adds a new element to an array (at the beginning), and "unshifts" older elements. It returns the new array length. Ex.

  ```
  const fruits = ["Banana", "Orange", "Apple", "Mango"];
  console.log(fruits.unshift("Grapes")); // Grapes,Banana,Orange,Apple,Mango
  ```

- delete operator can be used to delete elements because arrays are objects. Ex.

  ```
  delete fruits[0]; // Changes the first element in fruits to undefined
  ```

- splice() can be used to add new items to an array. Ex.

  ```
  var fruits = ["Banana", "Orange"];
  fruits.splice(1, 0, "Lemon"); // Array becomes Banana, Lemon, Orange
  ```

  splice() can be used to remove elements without leaving "holes" in the array. Ex.

  ```
  var fruits = ["Banana", "Orange", "Apple", "Mango"];
  var deleted = fruits.splice(0, 1); // Removes '1' element at '0' index of fruits
  console.log(deleted); // Banana
  ```

  The first parameter defines the position where new elements should be added (spliced in). The second parameter defines how many elements should be removed. splice() method returns an array with the deleted items, if any.

Array(contd.)

- concat() creates a new array by merging (concatenating) existing arrays. Ex.

  ```
  var myChildren = girls.concat(boys); // joins girls & boys
  ```

  concat() can take any number of array arguments. Ex.

  ```
  var myChildren = girls.concat(boys, dogs); // joins girls with boys & dogs
  ```

  concat() can also take values as arguments.

  ```
  var myChildren = arr1.concat(["Emil", "Tobias", "Linus"]);
  ```

  concat() does not change the existing arrays. It always returns a new array.

- slice() slices out a piece of an array into a new array. Ex.

  ```
  var fruits = ["Banana", "Orange", "Lemon", "Apple"];
  var citrus = fruits.slice(1); // citrus =  Orange,Lemon,Apple
  ```

  slice() can take two arguments like slice(1, 3). It selects elements from the start argument, and up to (but not including) the end argument. Ex.

  ```
  var fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
  var citrus = fruits.slice(1, 3); // citrus =  Orange,Lemon,Apple
  ```

  It creates a new array & does not remove any elements from the source array.

Array(contd.)

- sort() sorts an array alphabetically. Ex.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();        // Apple,Banana,Mango,Orange
```

By default, it sorts values as strings. However, if the numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". Because of this, the sort() method will produce incorrect results when sorting numbers. It can fixed by providing a compare function. Ex.

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

Use the same trick to sort an array descending. Ex.

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
```

Arrays often contain objects. Example:

```
const cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
  {type:"BMW", year:2010}
];
```

Array(contd.)

Even if objects have properties of different data types, the sort() method can be used to sort the array. The solution is to write a compare function to compare the property values: Example:

```
cars.sort(function(a, b){return a.year - b.year});
```

Comparing string properties is a little more complex. Example:

```
cars.sort(function(a, b){
  let x = a.type.toLowerCase();
  let y = b.type.toLowerCase();
  if (x < y) {
    return -1;
  }
  if (x > y) {
    return 1;
  }
  return 0;
});
```

- reverse() reverses the elements in an array. Ex.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.reverse(); // Returns Mango,Apple,Orange,Banana
```

Array(contd.)

- forEach() calls a function (a callback function) once for each array element. Syntax: arr.forEach((element, index, array) => { ... }). Ex.

  ```
  const a = ["a", "b", "c"];
  a.forEach((entry) => (console.log(entry)));
  ```

- map() creates a new array by performing a function on each array element. It does not execute the function for array elements without values. It does not change the original array. Syntax: arr.map((element, index, array) => { ... }). Ex.

  ```
  const array = [1, 4, 9, 16];
  const result = array.map(x => x * 2); // expected output: Array [2, 8, 18, 32]
  ```

- filter() creates a new array with array elements that passes a test. Syntax: arr.filter((element, index, array) => { ... }). Ex.

  ```
  const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
  const result = words.filter(word => word.length > 6); // expected output: ["exuberant", "destruction", "present"]
  ```

- reduce(callback(accumulator, currentValue[, index[, array]]) [, initialValue]) runs a function on each array element to produce (reduce it to) a single value. It does not reduce the original array. It works from left-to-right in the array. Ex.

  ```
  let numbers = [1, 2, 3];
  const result = numbers.reduce((accumulator, current) => accumulator + current); // expected output: 6
  ```

  For right to left use reduceRight(). Ex.

  ```
  const result = numbers.reduceRight((accumulator, current) => accumulator + current); // expected output: 6
  ```

Array(contd.)

- every() checks if all array values pass a test. Returns a boolean value as output. Works like &&. Syntax: arr.every((element, index, array) => { ... }). Ex.
  const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
  const result = words.every(word => word.length > 6); // expected output: false
- some() checks if some array values pass a test. Syntax: arr.some((element, index, array) => { ... }). Ex.
  const result = words.some(word => word.length > 6); // expected output: true
- indexOf() searches an array for an element value and returns its position. It returns -1 if the item is not found. If the item is present more than once, it returns the position of the first occurrence. Syntax: array.indexOf(item, start); // 'item' Required. The item to search for & 'start' (optional) where to start the search. Ex.
  let position = words.indexOf("elite"); // expected output: 2
  let position = words.indexOf("elite", 3); // expected output: -1
  Negative values will start at the given position counting from the end, and search to the end. Ex.
  let position = words.indexOf("elite", -4); // expected output: 2
- Array.lastIndexOf() returns the position of the last occurrence of the specified element. It returns -1 if the item is not found. If the item is present more than once, it returns the position of the first occurrence. Syntax:
  array.lastIndexOf(item, start); // 'item' Required

Array(contd.)

The item to search for & 'start'(optional) where to start the search. Ex.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
let position = words.lastIndexOf("elite"); // expected output: 2
let position = words.lastIndexOf("elite", 3); // expected output: -1
```

Negative values will start at the given position counting from the end, and search to the beginning. Ex.

```
let position = words.lastIndexOf("elite", -4); // expected output: 2
```

- includes() returns a boolean value for whether an array contains a specified element. Ex.

```
words.includes("'elite'"); // is true
```

- find() returns the value of the first array element that passes a test function. Syntax: `arr.find((element, index, array) => { ... } )`. Ex.

```
const array = [1, 4, 9, 16];
const result = array.find(x => x > 2); // expected output: 4
```

- findIndex() returns the index of the first array element that passes a test function. Syntax: `arr.findIndex((element, index, array) => { ... } )`. Ex.

```
const array = [1, 4, 9, 16];
const result = array.findIndex(x => x > 2); // expected output: 1
```

Object

JS objects are containers for named values called properties & functions.

Properties: The name:values pairs in JavaScript objects are called properties and can be accessed in three ways:

- objectName.propertyName
- objectName["propertyName"]
- objectName[expression]

New properties can be added to an existing object by simply giving it a value. Ex:

```
person.nationality = "English";
```

delete keyword deletes both the value of the property and the property itself. After deletion, the property cannot be used before it is added back again. The delete operator is designed to be used on object properties. It has no effect on variables or functions. Ex.

```
delete person.age;  // or delete person["age"];
```

All properties have a name & a value. The value is one of the property's attributes. Other attributes are: enumerable, configurable, and writable. In JavaScript, all attributes can be read, but only the value attribute can be changed (and only if the property is writable).

JS for...in statement loops through the properties of an object. Ex.

```
let user = { name: "John", age: 30 };
for(let prop in user) {
  console.log(prop, user[prop]); // logs name & John in first iteration
}
```

Object(contd.)

To test whether a property exists, we have an operator **in** for that. "key" in object. Ex.

```
let user = { name: "John", age: 30 };
console.log( "age" in user ); // true, user.age exists
console.log( "blabla" in user ); // false, user.blabla doesn't exists
```

Please note that on the left side of in there must be a property name. That's usually a quoted string. If we omit quotes, that means a variable, it should contain the actual name to be tested. Ex.

```
let key = "age";
alert( key in user ); // true, property "age" exists
```

Most of the time the comparison with undefined works fine. But there's a special case when it fails & in works correctly. It's when an object property exists, but stores undefined. Ex.

```
let obj = {
  test: undefined
};
alert(obj.test); // it's undefined, so - no such property?
alert("test" in obj); // true, the property does exist!
```

If we loop over an object, we do not get all properties in the same order they were added. Integer properties are sorted, others appear in creation order. To avoid the sorting of integers we can make them as non-integers by prefixing them with a '+'.

Object(contd.)

Functions are actions that can be performed on objects & are stored in properties as function definitions. They can be accessed as objectName.methodName(). Ex.

```
var person = {
  firstName: "John",
  lastName : "Doe",
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

In a function definition, this refers to the "owner" of the function. In example above, this is the person object that "owns" the fullName function. In other words, this.firstName means the firstName property of this object. Note: Objects cannot be compared.

```
var x = new String("John");
var y = new String("John");
// (x == y) is false because x and y are different objects
```

When a JavaScript variable is declared with the keyword "new", the variable is created as an object:

```
var x = new String(); // Declares x as a String object
var y = new Number(); // Declares y as a Number object
var z = new Boolean(); // Declares z as a Boolean object
```

Object(contd.)

Avoid String, Number and Boolean objects. They complicate your code and slow down execution speed.

Info: ES6 defines ComputedPropertyName as part of the grammar for object literals, allows you to create attribute of an object from identifier like this:

```
var data = "value";
obj = { [data]: 10 };
```

Methods

- Object.keys(object) - Returns enumerable properties(keys) as an array. Ex.

  ```
  let user = { name: "John", age: 30 };
  console.log(Object.keys(user)); // returns ['name', 'age']
  ```

- Object.values(object) - Returns values as an array. Ex.

  ```
  let user = { name: "John", age: 30 };
  console.log(Object.values(user)); // returns ['John', 30']
  ```

- Object.entries(object) - Returns keys & values as an array in the form [key, value]. Ex.

  ```
  let user = { name: "John", age: 30 };
  console.log(Object.values(user)); // returns [['name', 'John'], ['age', 30']]
  ```

Map

Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.

It's methods and properties are followed by example:

- new Map() – creates the map.
- map.set(key, value) – stores the value by the key.
- map.get(key) – returns the value by the key, undefined if key doesn't exist in the map.
- map.size – returns the current element count.

```
let map = new Map();
map.set('1', 'str1'); // a string key
map.set(1, 'num1'); // a numeric key
map.set(true, 'bool1'); // a boolean key
console.log(map.get(1)); // 'num1'
console.log(map.get('1')); // 'str1'
console.log(map.size); // 3
```

- map.has(key) – returns true if the key exists, false otherwise. Ex:

```
console.log(map.has(1)); // true
console.log(map.has(2')); // false
```

Map(contd.)

- map.delete(key) – removes the value by the key. Ex:

  ```
  map.delete(1); // deletes entry with key 1
  ```

- map.clear() – removes everything from the map. Ex:

  ```
  map.clear();
  console.log(map.size); // 0
  ```

map[key] isn't the right way to use a Map. Although map[key] also works, e.g. we can set `map[key] = 2`, this is treating map as a plain JavaScript object, so it implies all corresponding limitations (no object keys and so on). So we should use map methods. Map can also use objects as keys. Ex.

```
let john = { name: "John" };
let visitsCountMap = new Map(); // for every user, let's store their visits count
visitsCountMap.set(john, 123); // john is the key for the map
console.log(visitsCountMap.get(john)); // 123
```

Every map.set call returns the map itself, so we can "chain" the 'set' calls. Ex:

```
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

Map(contd.)

For looping over a map, there are 3 methods followed by examples:

- map.keys() - Returns iterable object for keys
- map.values() - Returns iterable object for values
- map.entries() - Returns keys & values as iterable object in [key, value] format

```
const map = new Map();
map.set('1', 'str1'); // a string key
map.set(1, 'num1'); // a numeric key
map.set(true, 'bool1');
const iterator1 = map.keys();
console.log(iterator1.next().value); // expected output: "1"
console.log(iterator1.next().value); // expected output: 1
const iterator2 = map.values();
console.log(iterator2.next().value); // expected output: "str1"
console.log(iterator2.next().value); // expected output: "num1"
const iterator3 = map.entries();
console.log(iterator3.next().value); // expected output: ["0", "foo"]
console.log(iterator3.next().value); // expected output: [1, "bar"]
```

Map(contd.)

The iteration goes in the same order as the values were inserted. Map preserves this order, unlike a regular Object. Besides that, Map has a built-in forEach method, similar to Array. Ex.

```
recipeMap.forEach((value, key) => { // runs the function for each (key, value) pair
  alert(`${key}: ${value}`);
});
```

When a Map is created, we can pass an array (or another iterable) with key/value pairs for initialization. Ex.

```
let map = new Map([ // array of [key, value] pairs
  ['1', 'str1'],
  [1, 'num1']
]);
console.log(map.get('1')); // str1
```

We can create a Map from a plain object, by using the built-in method Object.entries(obj) that returns an array of key/value pairs. Ex.

Map(contd.)

```
let obj = {
  name: "John",
  age: 30
};
let map = new Map(Object.entries(obj));
console.log(map.get('name')); // John
```

We can create a plain object from a Map, by using the built-in object method Object.fromEntries(obj) that accepts an array of key/value pairs from map's entries(). Ex.

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);
let obj = Object.fromEntries(map.entries()); // entries() returns iterable obj of [key, value]
console.log(obj.orange); // 2
let obj = Object.fromEntries(map);//Shorthand: map returns [key, value] as map.entries()
```

Set

A Set is a special type collection – "set of values" (without keys), where each value can occur only once. Its main methods followed by examples are:

- new Set(iterable) – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
- set.add(value) – adds a value, returns the set itself.
- set.size – is the elements count.

```
const set = new Set();
set.add(1); // a numeric value
set.add('abc');  // a string value
set.add(true); // a boolean value
set.add({a: 1, b: 2})
console.log(set.size)); // 4
console.log(set)); // Set(4) [ "abc", 1, true, {...}, ]
```

- set.delete(value) – removes the value, returns true if value existed at the moment of the call, otherwise false. Ex:

```
set.delete(true);
console.log(set.size)); // 3
```

Set(contd.)

- set.has(value) – returns true if the value exists in the set, otherwise false. Ex.

```
set.has(1);          // true
set.has(3);          // false, since 3 has not been added to the set
```

- set.clear() – removes everything from the set. Ex.

```
set.clear();
console.log(set.size)); // 0
```

The main feature is that repeated calls of set.add(value) with the same value don't do anything. That's the reason why each value appears in a Set only once. Ex:

```
const set = new Set();
set.add(5);
set.add(5);
console.log(set.size)); // 1
```

We can loop over a set either with for..of or using forEach, just like an array. Ex.

```
let set = new Set(["oranges", "apples", "bananas"]);
for (let value of set) console.log(value);
set.forEach((value, valueAgain, set) => { // the same with forEach
  console.log(value);
});
```

Set(contd.)

The callback function passed in forEach has 3 arguments: a value, then the same value valueAgain, and then the target object. That's for compatibility with Map where the callback passed forEach has 3 arguments. It helps to replace Map with Set & vice versa.

For looping over a set, there are 3 methods with examples:

- set.keys(object) - Returns iterable for values

  ```
  let set = new Set(["oranges", "apples", "bananas"]);
  const iterator1 = set.keys();
  console.log(iterator1.next().value); // expected output: “oranges”
  console.log(iterator1.next().value); // expected output: "apples"
  ```

- set.values(object) - Same as set.keys(), for compatibility with Map

  ```
  const iterator2 = set.values();
  console.log(iterator2.next().value); // expected output: “oranges”
  console.log(iterator2.next().value); // expected output: "apples"
  ```

- set.entries(object) - Returns iterable object for entries [value, value]. Ex:

  ```
  const iterator3 = set.entries();
  console.log(iterator3.next().value); // expected output: ["oranges", "oranges"]
  console.log(iterator3.next().value); // expected output: ["apples", "apples"]
  ```

Iteration over Map and Set is always in the insertion order, so we can't say that these collections are unordered, but we can't reorder elements or directly get an element by its number.

Destructuring

The destructuring assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Array

Destructuring array. Ex.

```
let arr = ["Ilya", "Kantor"];
let [firstName, surname, middleName = "Kumar"] = arr; // We can replace missing one's by "default" values by providing '='
console.log(firstName); // Ilya
console.log(surname); // Kantor
console.log(middleName); // Kumar
```

Destructuring string into variables by converting them to array. Ex.

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

Unwanted elements of the array can also be thrown away via an extra comma. Ex.

```
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
console.log(title); // Consul
```

Destructuring(contd.)

We can use it with any iterable.

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

We can use any “assignables” at the left side. For instance, an object property.

```
let user = {};
[user.name, user.surname] = "Alpha Beta".split(' ');
console.log(user.name); // Alpha
```

Swapping values of two variables.

```
let firstName = "Alpha";
let surname= "Beta";
[firstName, surname] = [surname, firstName]; // Swap values
console.log(`${firstName} ${surname}`); // Alpha Beta
```

Destructuring(contd.)

Object

Existing object on the right can be split into variables. The left side contains a “pattern" for corresponding properties. In the simple case, that’s a list of variable names in {...}. We can also assign a property to a variable with another name & can set it by colon. For potentially missing properties we can set default values using "=". Ex.

```
let options = {
  title: "Menu",
  width: 100
};
let {width: w, height = 200, title} = options; // { sourceProperty: targetVariable }
console.log(title); // Menu (title -> title)
console.log(w); // 100 (width -> w)
console.log(height); // 200 (height -> height)
```

The order does not matter. This works too: Ex.

```
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

Destructuring(contd.)

We also can combine both the colon and equality. Ex.

```
let options = {
  title: "Menu",
  width: 100
};
let {width: w, height: h = 200, title} = options;
console.log(title); // Menu
console.log(w); // 100
console.log(h); // 200
```

If we have a complex object with many properties, we can extract only needed ones. Ex.

```
let options = {
  title: "Menu",
  width: 100
};
let { title } = options; // only extract title as a variable
console.log(title); // Menu
```

Destructuring(contd.)

Nested

If an object or an array contains other nested objects and arrays, we can use more complex patterns to extract deeper portions.

Ex.

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};
let { // destructuring assignment split in multiple lines for clarity
  size: { // put size here
    width,
    height
  },
  items: [item1, item2], // assign items here
  title = "Menu" // not present in the object (default value is used)
} = options;
```

Destructuring(contd.)

```
console.log(title); // Menu
console.log(w); // 100
console.log(h); // 200
console.log(item1); // Cake
console.log(item2); // Donut
```

Smart function parameters

There are times when a function has many parameters, most of which are optional. In real-life, the problem is how to remember the order of arguments. Destructuring comes to the rescue!. We can pass parameters as an object, and the function immediately destructurizes them into variables. Ex.

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};
function showMenu({
  title = "Untitled",
  width: w = 200,
  height: h = 100,
  items = [item1, item2]
```

Destructuring(contd.)

```
}) { // ...and it immediately expands it to variables
  // title, items – taken from options but width, height – defaults used
  console.log(`${title} ${w} ${h}`); // My menu 200 100
  console.log(item1); // Item1
  console.log(item2); // Item2
}
showMenu(options); // we pass object to function
```

If we want all values by default, then we should specify an empty object. Ex.

```
showMenu({}); // ok, all values are default
showMenu(); // this would give an error
```

We can fix this by making {} the default value for the whole object of parameters. Since, the whole arguments object is {} by default, so something always to destructurize. Ex.

```
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
  console.log(`${title} ${weight} ${height}`);
}
showMenu(); // Menu 100 200
```

Spread operator

Spread syntax allows an iterable such as an array/string/object to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected, or an object expression to be expanded in places where zero or more key-value pairs (for object literals) are expected. Syntax:

For function calls: myFunction(...iterableObj);

For array literals: [...iterableObj, '4', 'five', 6];

For strings: [...'five']; // [f,i,v,e] we could also use Array.from(). Array.from operates on both array-likes and iterables. The spread syntax works only with iterables.

For object literals (new in ECMAScript 2018): let objClone = { ...obj }; Ex.

```
let x = [5, 10, 15];
console.log(Math.max(...x));
```

Rest operator

Rest syntax looks exactly like spread syntax but is used for destructuring arrays and objects. In a way, rest syntax is opposite of spread syntax: spread 'expands' an array into its elements, while rest collects multiple elements and 'condenses' them into a single element. Ex.

For arrays:

```
function myFun(a, b, ...manyMoreArgs) {
  console.log(a); // "one"
  console.log(manyMoreArgs); // ["three", "four", "five", "six"]
}
myFun("one", "two", "three", "four", "five", "six");
```

The rest parameters gather all remaining arguments, so the following does not make sense and causes an error.

For objects:

```
let options = {
  title: "Menu",  width: 100,  height: 200
};
let { title,  ...rest} = options; // title = title property & rest = object with rest of properties, now title="Menu", rest={height: 200, width: 100}
console.log(rest.height); // 200
console.log(rest.width); // 100
```

Thank You