JavaScript fundamentals

Index

Variables and Data Types

- A variable is an empty space holder for a future value
- There are a few types of values that JavaScript uses
  - Numbers and Booleans (true or false)
  - Strings or a sequence of characters
  - Functions
  - Arrays (lists of variables) and Objects

Primitive data types

- There are a few types of values that JavaScript uses:

Variable	Explanation	Example
String	This is a sequence of text known as a string. To signify that the value is a string, enclose it in single or double quote marks.	let myVariable = 'Bob'; or let myVariable = "Bob";
Number	This is a number. Numbers don't have quotes around them.	let myVariable = 10;
Boolean	This is a True/False value. The words true and false are special keywords that don't need quote marks.	let myVariable = true;
Array	This is a structure that allows you to store multiple values in a single reference.	let myVariable = [1,'Bob','Steve',10]; Refer to each member of the array like this: myVariable[0], myVariable[1], etc.

Primitive data types

- Special primitive types:
  - null for unknown values – a standalone type that has a single value null.
  - undefined for unassigned values – a standalone type that has a single value undefined.

Object	For more complex data structures. This can be anything. Everything in JavaScript is an object and can be stored in a variable.	let myVariable = document.querySelector('h1');

Variables

- Primitive types: undefined, number, string, boolean, function, object

```
let somevar1 = “Hola”;
typeof somevar1; // String

let somevar2 = 25;
typeof somevar2 // number

let somevar3 = new Date();
typeof somevar3; // Object
```

```
let somevar4;

typeof somevar4; // undefined
```

```
let somevar5 = () => {};

typeof somevar5; // function
```

Variables

- Weak, dynamic typing
- Variables have the type of the last thing assigned to it

```
let somevar;
typeof somevar; // undefined

somevar = 25;
typeof somevar // number

somevar = “hello”;
typeof somevar; // string
```

```
let somevar;
typeof somevar; // undefined

let somevar = () => {};
typeof somevar; // function
```

Operators

Operator	Explanation	Symbol(s)	Example
Addition	Add two numbers together or combine two strings.	+	6 + 9; 'Hello ' + 'world!';

Subtraction, Multiplication, Division	These do what you'd expect them to do in basic math.	-, *, /	9 - 3;
			8 * 2; // multiply in JS is an asterisk
			9 / 3;

Assignment	As you've seen already: this assigns a value to a variable.	=	let myVariable = 'Bob';
Strict equality	This performs a test to see if two values are equal and of the same data type. It returns a true/false (Boolean) result.	===	let myVariable = 3; myVariable === 4;

Not, Does-not-equal	This returns the logically opposite value of what it precedes. It turns a true into a false, etc.. When it is used alongside the Equality operator, the negation operator tests whether two values are *not* equal.	!, !==	For "Not", the basic expression is true, but the comparison returns false because we negate it:
			
			let myVariable = 3;
			!(myVariable === 3);
			
			"Does-not-equal" gives basically the same result with different syntax. Here we are testing "is myVariable NOT equal to 3". This returns false because myVariable IS equal to 3:
			
			let myVariable = 3;
			myVariable !== 3;

Operators - Loose vs Strict Equality

- Double equals (==) is a comparison operator, which transforms the operands having the same type before comparison.
- So, when you compare string with a number, JavaScript converts any string to a number. An empty string is always converts to zero. A string with no numeric value is converts to NaN (Not a Number), which returns false.
- === (Triple equals) is a strict equality comparison operator in JavaScript, which returns false for the values which are not of a similar type. This operator performs type casting for equality. If we compare 2 with “2” using ===, then it will return a false value.

Operators - Loose vs Strict Equality

```
const a = 100;
const b = '100';

console.log(a == b) // true
```

```
const a = 100;
const b = '100';

console.log(a === b) // false
```

- The type of variable a is number and the type of variable b is string
- The variable a is converted to a string before making the comparison.
- It is important to note that the actual values remains unchanged. It only implicitly gets converted while comparing.
- Triple equals checks for the types of the operands first – and those types differ in this example, a is number and type of variable b is string . So, it returns false.

Operators - Loose vs Strict Equality

```
const a = true;
const b = 'true';

console.log(a == b); // false
```

- The variable a (true) gets converted to a number before the comparison.
- So after comparison – where we're now comparing 1 and 'true' – we get false because the variables contain different values.

Operators - Loose vs Strict Inequality

```
const a = 2;

console.log(a != 2); // false
console.log(a != '2'); // false
```

```
const a = 2;

console.log(a !== 2); // false
console.log(a !== '2'); // true
```

- The JavaScript not equal or inequality operator (!=) checks whether two values are not equal and returns a boolean value. This operator tries to compare values irrespective of whether they are of different types.
- However, the “!==” or Strict inequality operator does not attempt to do so and returns false if the values are unequal or of different types.

Rules for Type Coercion

- If either operand is a `string`, the other operand will be converted to a `string`.
- If either operand is a `number`, the other operand will be converted to a `number`.
- If either operand is a `boolean`, it will be converted to a `number` (`true` becomes `1` and `false` becomes `0`).
- If one operand is an `object` and the other is a primitive value, the object will be converted to a primitive value before the comparison is made.
- If one of the operands is `null` or `undefined`, the other must also be `null` or `undefined` to return `true`. Otherwise it will return `false`.

Let and Scope

- Let creates a variable with scope
- Scope is a term that defines a boundary where variables live
- Scope is how you can ensure content inside a function is not affected by the outside
- Scope in Javascript is largely defined by curly brackets ('{}')

Let example

```
let a = 50;
let b = 100;
if (true) {
    let a = 60;
    let c = 10;
    console.log(a/c); // 6
    console.log(b/c); // 10
}
console.log(c); // 10
console.log(a); // 50
```

- The variable a is found both in the scope of this script, and in the scope of the if statement block
- The variable a within the block can be considered a different variable than the variable a outside the block

Problem with var

```
var name = "my name";
var myAge = 22;

if(myAge > 18) {
    var name = "another person name";
}

console.log(name);
//output => "another person name"
```

- var variable can be re-declared and updated
- re-declaration allows declaring more than one variable with the same name, because of this reason, if we declare a new variable by mistake, it will override the original variable value.

var vs let

```
var a = 5;
console.log(a); // 5
{
   var a = 3;
   console.log(a); // 3
}
console.log(a); // 3
```

- Redeclaring a variable with var in a different scope or block changes the value of the outer variable too

```
let a = 5;
console.log(a); // 5
{
   let a = 3;
   console.log(a); // 3
}
console.log(a); // 5
```

- Redeclaring a variable with let in a different scope or block treats that variable as a different variable

var vs let

```
var a = 2;
for(var a = 0; a < 3; a++) {
   console.log('hello');
}
console.log(a); // 3
```

```
let a = 2;
for(let a = 0; a < 3; a++) {
   console.log('hello');
}
console.log(a); // 2
```

- When a variable declared with var is used in a loop, the value of that variable changes
- When a variable declared with let is used in a loop, the value of a variable does not change

const

- There a times where you do not want a variable to change after assignment
- For example, if you have a variable that is set to the number PI
- You wouldn't want that variable PI to change during your program

const example

```
const B = "Constant variable";
B = "Assigning new value"; // shows error.

const LANGUAGES = ['Js', 'Ruby', 'Python',
'Go'];
LANGUAGES = "Javascript"; // shows error.

LANGUAGES.push('Java'); // Works fine.
console.log(LANGUAGES); // ['Js', 'Ruby',
'Python', 'Go', 'Java']
```

- The variable LANGUAGES can not be changed
- However, what LANGUAGES points to, if it is mutable can change

Strings

```
let greeting = "Hello";
let farewell = 'Bye';
let word = 'text';

let mix = greeting  + ' ' + farewell ;
let phrase = `you can embed another ${text}`;

console.log(`1 + 2 = ${1 +2}.`);
```

- Single and double quotes are essentially the same.
- Backticks, however, allow us to embed any expression into the string, by wrapping it in ${…}

Template literals

Old Way

```
function formatGreetings(name, age) {

var str = "Hi " + name +
   " your age is " + age;
```

Use string concatenation to build up string from variables.

New Way

```
function formatGreetings(name, age) {

let str =
   `Hi ${name} your age is ${age}`;
```

Also allows multi-line strings:

```
`This string has
two lines`
```

Very useful in frontend code. Strings can be delimited by " ", ' ', or ` `

Conditionals

```
// Full if else
let result = '';
if (our_value) {
   result = 'we got a value';
} else {
   result = 'no value';
}

// Ternary
result = our_value ? 'we got a value' : 'no value';
```

While Loops

```
while (condition) {
    // body of loop
}
```

```
let answerQuestion = function() {
    let answer = prompt("What is 4 + 4");
    if (answer == "8") {
        return true;
    } else {
        return false;
    }
}
let answer = false;
while (answer != true) {
    answer = answerQuestion();
}
alert("Correct Answer!");
```

Do-While Loops

```
do {
    // body of loop
} while (condition)
```

```
// program to display numbers
let i = 1;
const n = 5;

// do...while loop from 1 to 5
do {
    console.log(i);
    i++;
} while(i <= n);
```

Functions

Our First Function

```
function helloWorld(first_name, last_name) {
   let message = "Hello World " + first_name + " " + last_name;
   return message;
}

let first = prompt("Enter your first name");
let last = prompt("Enter your last name");
let output = helloWorld(first, last);
alert(output);
```

What does this function do?

Functions

What does this function do?

```
function helloWorld(first_name, last_name) {
   let message = "Hello World " + first_name +
                 " " + last_name;
   return message;
}

let first = prompt("Enter your first name");
let last = prompt("Enter your last name");
let output = helloWorld(first, last);
alert(output);
```

- Declared a function that we put in two values or inputs: (first name and last name)
- The function declared a variable (message) that contains the string "Hello World [your first name] [your last name]"
- After it's done, it will produce one value or output: (the message) and return it
- We ask the user for their first and last name, and print out the result from the function

Functions

5 ways to define

1. Function declaration

```
function sum(a, b) {
   return a+b;
}
sum(2,3);
```

2. Function expression

```
let sum = function(a, b) {
   return a+b;
}
sum(2,3);
```

3. Arrow function

```
let sum =(a, b) => {
   return a+b;
}
sum(2,3);
```

4. IIFE function

```
(function(a, b) {
   return a+b;
})(2,3);
```

5. Function constructor

```
let sum = new Function(
'a',
'b',
'return a+b'
);
sum(2,3);
```

Arrow Functions

- There is a new way of defining functions
- There are a few reasons for this (and that's actually a pun, but you can look that up to figure it out)
- This new way of writing function also helps with clearly defining scope

Arrow Functions Example

```
function oldOne(name) {
    console.log("Hello " + name);
}
oldOne("Kay");

// New Syntax
let newOne = (name) => {
    console.log("Hello " + name);
}
newOne("Kay");
```

- The parameters are named in the parentheses outside the name of the function
- Note how you assign a variable to a function (and can use let for scope)

Arrow Functions Example

- Arrow functions are handy for simple actions, especially for one-liners. They come in two flavors:
  - Without curly braces: **(...args) => expression** – the right side is an expression: the function evaluates it and returns the result. Parentheses can be omitted, if there's only a single argument, e.g. n => n*2.
  - With curly braces: **(...args) => { body }** – brackets allow us to write multiple statements inside the function, but we need an **explicit return** to return something.

```
let arrowLineExample = a => a;
console.log(arrowLineExample(1)); //1

let arrowBadBlockExample = a => { a};
console.log(arrowBadBlockExample(2)); // undefined
```

```
let arrowBlockExample = a => {return a};
console.log(arrowBlockExample(2)); // 2
```

for Loops

```
for (initialExpression; condition; updateExpression) {
   // for loop body
}
```

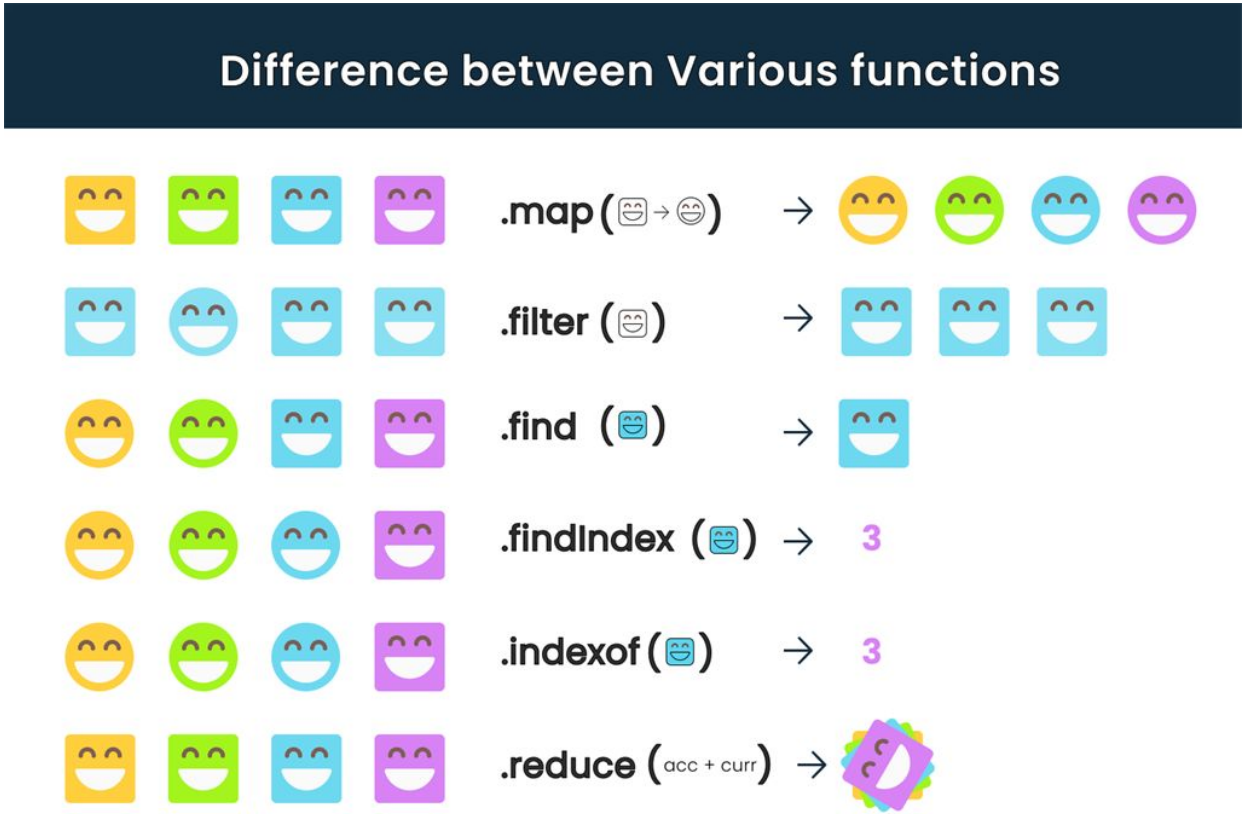
```
function countToX(x) {
    let message = "";
    for(let i = 0; i <= x; i = i+1) {
        message = message + i + " ";
    }
    return message;
}
```

Arrays

- Array indexes start from 0

```
let data = [2,3, "KAY"];
console.log(data.length);   // 3

let primero = data[0]; // 2
let ultimo = data[data.length - 1]; // "KAY"
```

Arrays


Difference between Various functions
.map (→)
.filter ( )
.find ( )
.findIndex ( )
3
.indexof ( )
3
.reduce (acc + curr)

Array - Map

Sintax

```
reduce(callbackFn)
reduce(callbackFn, initialValue)
```

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce((accumulator, currentValue) =>
accumulator + currentValue, initialValue);

console.log(sumWithInitial);
// Expected output: 10
```

Array - Filter

Sintax

```
reduce(callbackFn)
reduce(callbackFn, initialValue)
```

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce((accumulator, currentValue) =>
accumulator + currentValue, initialValue);

console.log(sumWithInitial);
// Expected output: 10
```

Array - Reduce

Sintax

```
reduce(callbackFn)
reduce(callbackFn, initialValue)
```

```
const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4
const initialValue = 0;
const sumWithInitial = array1.reduce(
      (accumulator, currentValue) => accumulator + currentValue,
      initialValue);

console.log(sumWithInitial);
// Expected output: 10
```

Array - Map

```
const array1 = [1, 4, 9, 16];

// Pass a function to map
const map1 = array1.map((x) => x * 2);

console.log(map1);
// Expected output: Array [2, 8, 18, 32]
```

Array - Filter

```
const words = ['spray', 'limit', 'elite', 'exuberant',
'destruction', 'present'];

const result = words.filter((word) => word.length > 6);

console.log(result);
// Expected output: Array ["exuberant", "destruction", "present"]
```

Array - Reduce

Sintax

```
reduce(callbackFn)
reduce(callbackFn, initialValue)
```

```
const array1 = [1, 2, 3, 4];

const initialValue = 0;
const sumWithInitial = array1.reduce(
     (accumulator, current) => accumulator + current,initialValue);
// 0 + 1 + 2 + 3 + 4

console.log(sumWithInitial); // Expected output: 10
```

Array - foreach

```
const array1 = ['a', 'b', 'c'];

array1.forEach((element) => console.log(element));

// Expected output: "a"
// Expected output: "b"
// Expected output: "c"
```

Primitive data types

Method	Runs through each item	Executes given function	Returns the result	Number of elements in result (compared to original array)
.map	✔	✔	in array	=
.filter	✔	✔	if true, in array	=<
.forEach	✔	✔	no *return is undefined*	none
.reduce	✔	✔	in array or anything else	one (a single number or string) *Reduce transforms an array into something else*
for loop	✔	until condition is false *You know the number of iterations beforehand*	*They run code blocks. They aren't functions so don't need to return*	>, = or <
while loop	✔	while condition is true *You don't know the number of iterations beforehand*		>, = or <

@manumagalhaes

for...of loops

Old Way

```
var a = [5,6,7];
var sum = 0;
for (var i = 0; i < a.length; i++) {
    sum += a[i];
}
```

Iterator over an array

New Way

```
let a = [5,6,7];
let sum = 0;
for (let ent of a) {
    sum += ent;
}
```

- Iterate over arrays, strings, Map, Set, without using indexes.
- Each element of that array is assigned to the variable ent once
- Note you do not have access to the index while using this construct

Array

for...in vs for... of

```
let list = [4, 5, 6];

for (let i in list) {
   console.log(i); // "0", "1", "2",
}

for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

for...of vs for...in

- For of (iterates over values):

```
for(let student of students) {
    console.log(student);
} //Prints out all student names
```

- For in (iterates over keys):

```
for(let key in students){
    console.log(key + ": " + students[key]);
} //Prints out all the index and student names
```

for ... in loops

```
const student = {
    name: 'Monica',
    class: 7,
    age: 12
}
```

```
// using for...in
for (let key in student) {
    // display the properties
    console.log(`${key} => ${student[key]}`);
}
```

- In each iteration of the loop, a key is assigned to the key variable.
- The loop continues for all object properties.

for...of Vs for...in

for...of	for...in
The `for...of` loop is used to iterate through the values of an iterable.	The `for...in` loop is used to iterate through the keys of an object.
The `for...of` loop cannot be used to iterate over an object.	You can use `for...in` to iterate over an iterable such arrays and strings but you should avoid using `for...in` for iterables.

Array - forEach

Sintax

```
array.forEach(callback(element [, index] [, arr])[, thisValue])
```

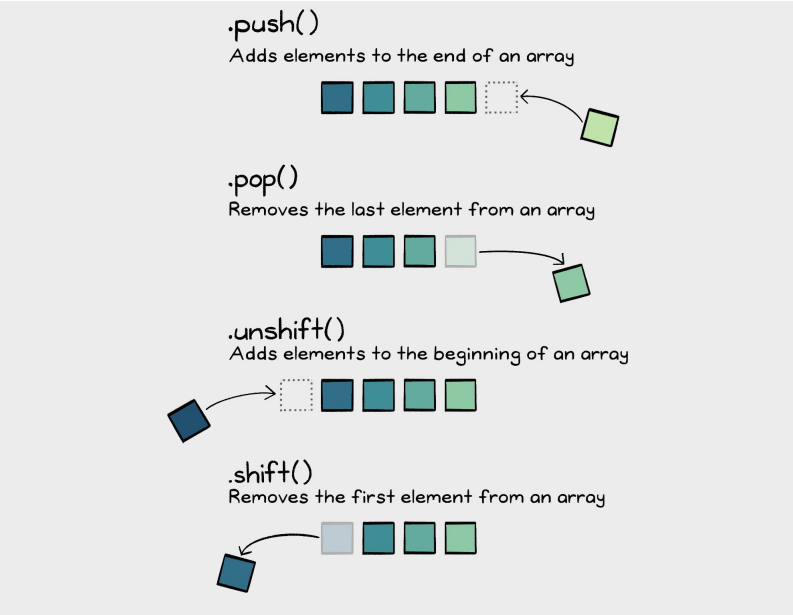
```
const array1 = ['a', 'b', 'c'];
array1.forEach((element) => console.log(element));

// Expected output: "a"
// Expected output: "b"
// Expected output: "c"
```

Array - forEach

```
function logArrayElements(element, index, array) {
    console.log("a[" + index + "] = " + element);
}

[2, 5, , 9].forEach(logArrayElements);

// Prints out:
// a[0] = 2
// a[1] = 5
// a[2] = 9

//Note that the 2nd index is avoided since there is no element in that
position in the array.
```

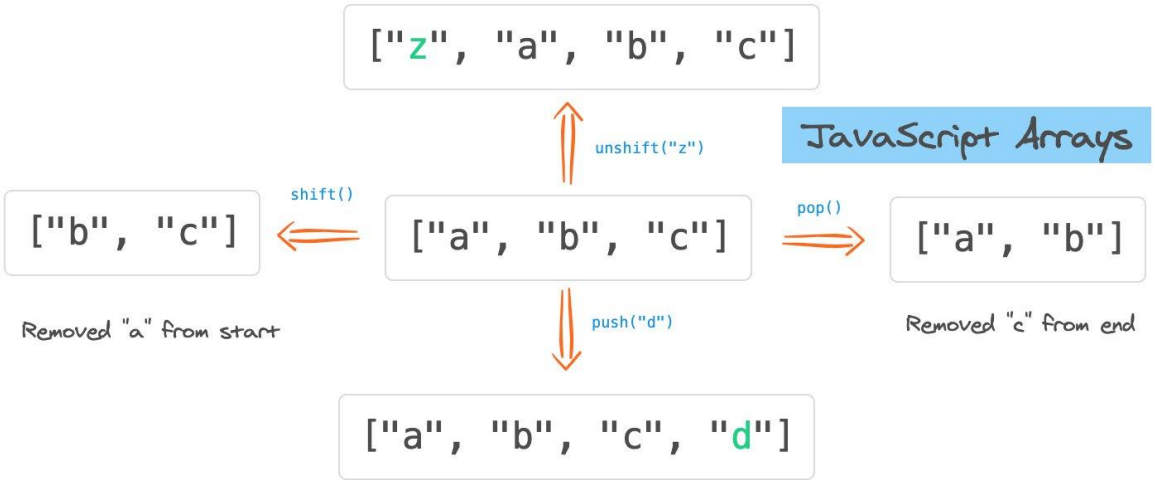
Array

push, .pop, .shift, and .unshift



Array

.push, .pop, .shift, and .unshift



[Javascript Basics: Use .push, .pop, .shift, and .unshift to Manipulate Arrays]

Arrays <-> Strings

- In JavaScript, strings are immutable
- JavaScript includes many built-in functions for creating and manipulating strings in various ways.
- Very often there are situations where strings have to be converted into arrays in order to manipulate their content.
- There are many ways to split a string into character arrays, i.e.:

```
let someArray = someString.split();

let someArray = Array.from(someString);
```

- To build up a new string from an array you can use

```
let someOtherString = someArray.join();
```

Destructuring arrays

Old Way

```
var a = arr[0];
var b = arr[1];
var c = arr[2];
```

New Way

```
let [a,b,c] = arr;
```

```
var arr = [1,2,3,4,5,6];
var a = arr[0];
var b = arr[1];
var d = arr[3];
var e = arr[4];
```

```
let arr = [1,2,3,4,5,6];
let [a,b,,d,e] = arr;

console.log(a); //1
console.log(b); //2
console.log(d); //4
console.log(e); //5
```

- You can skip what you don't want by leaving that position blank

Cloning arrays

- Shallow copy
  - Spread Operator
  - Array.from
  - Array.slice
  - Array.map
  - While Loop
  - For Loop
- Deep copy
  - JSON.parse and JSON.stringify

Cloning arrays

```
const fruits = ["Strawberry", "Mango"];


// Create a copy using spread syntax.
const fruitsCopy = [...fruits]; // ["Strawberry", "Mango"]


// Create a copy using the from() method.
const fruitsCopy2 = Array.from(fruits); // ["Strawberry", "Mango"]


// Create a copy using the slice() method.
const fruitsCopy3 = fruits.slice(); // ["Strawberry", "Mango"]
```

Default Parameters

Parameters not specified

Old Way

```
function myFunc(a,b) {
   a = a || 1;
   b = b || "Hello";
}
```

Unspecified parameters are set to undefined. You need to explicitly set them if you want a different default.

New Way

```
function myFunc (a = 1, b = "Hello") {

}
```

Can explicitly define default values if parameter is not defined.

Default Parameters

- Convenient ability to assign parameters to a function a value if not specified by the caller

```
let func = (a, b = 10) => {
   return a + b;
}
console.log(func(20)); // 20 + 10 = 30

console.log(func( 20, 50)); // 20 + 50 = 70

let notWorkingFunction = (a =  10, b) => {
   return a + b;
}
console.log(notWorkingFunction( 20)); // NAN. Not gonna work.
```

Default Parameters

- Default function parameters allow named parameters to be initialized with default values if **no value** or **undefined** is passed.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters

Rest parameter ...

Old Wav

```
function myFunc() {
    for (var i = 0; i < arguments.length; i++){
        console.log(arguments[i]);
    }
}
myFunc("Nick", "Anderson", 10, 12, 6);
// Nick
// Anderson
// 10
// 12
// 6
```

Parameters not listed but passed can be accessed using the arguments array.

New Way

```
function myFunc (a,b,...theArgsArray) {
  theArgsArray.forEach(e=>console.log(e));
}
myFunc("Nick", "Anderson", 10, 12, 6);
// 10
// 12
// 6
```

Additional parameters can be placed into a named array.

Rest parameter...

- Ability to define a function with a variable number of parameters
- You do not have to pass an array in order to have a variable number of parameters

Rest parameter...

```
let sumElements = (...arr) => {
    console.log(arr); // [10, 20, 40, 60, 90]
    let sum = arr.reduce((accE,curE)=>accE+curE);
    console.log(sum);
}

sumElements(10, 20, 40, 60, 90);
sumElements(10, 20, 90);

// [10, 20, 40, 60, 90]
// 220
// [10, 20, 90]
// 120
```

- You can pass a variable number of parameters
- Those parameters are available as an array inside the function

Spread operator ...

Old Way

```
var iniArr = ['My', 'name', 'is'];
var finalArr = ['Hello.'];

// Copy items from iniArr to finalArr using push()
for (var i = 0; i < iniArr.length; i++) {
   finalArr.push(iniArr[i]);
}
finalArr.push('Mia');
```

New Way

```
const iniArr = ['My', 'name', 'is'];
const finalArr  = ['Hello.',...iniArr ,'Mia'];
```

- Expand an array to pass its values to a function or insert it into an array.
- Works on iterable types: strings & arrays

Spread operator ...

Old Way

```
let arr1 = [ 1, 2, 3];
let arr2 = arr1;
console.log(arr1); // [1, 2, 3]
console.log(arr2); // [1, 2, 3]

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3, 4]
```

Objects are assigned by reference and not by values

New Way

```
let arr1 = [ 1, 2, 3];
// copy using spread syntax
let arr2 = [...arr1];

// append an item to the array
arr1.push(4);

console.log(arr1); // [1, 2, 3, 4]
console.log(arr2); // [1, 2, 3]
```

Arrays can be copied using the Spread Operator

Falsy values

```
if (false) {
    // Not reachable
}

if (null) {
    // Not reachable
}

if (undefined) {
    // Not reachable
}

if (0) {
    // Not reachable
}
```

```
if (-0) {
    // Not reachable
}

if (0n) {
    // Not reachable
}

if (NaN) {
    // Not reachable
}

if ("") {
    // Not reachable
}
```

They are coerced to false in Boolean contexts

- || returns the first truthy value.
- ?? returns the first defined value.

Nullish coalescing operator (??)

- Nullish coalescing operator (??)

  val1 ?? val2; - Returns val2 if it is null or undefined otherwise val1, like val1 || val2

```
let height = 0;
console.log(height || 300); // 300
console.log(height ?? 300); // 0

let name = "";
console.log(name || "Anonymus3"); // Anonymus3
console.log(name ?? "Anonymus3"); // ""
```