JAVASCRIPT ES6 LANGUAGE

JavaScript Statements and Syntax

```
let x, y, z;     // Statement 1
x = 5;           // Statement 2
y = 6;           // Statement 3
z = x + y;       // Statement 4
```

- Semicolons (;) separate JavaScript statements.

```
a = 5; b = 6; c = a + b;
```

- Code after double slashes `//` or between `/*` and `*/` is treated as a **comment**.

All JavaScript identifiers are **case sensitive**

- JavaScript programmers tend to use camel case that starts with a lowercase letter. Ex: firstName, lastName, masterCard, interCity.

JavaScript Variables

- 4 Ways to Declare a JavaScript Variable:
  - Using `var`
  - Using `let`
  - Using `const`
  - Using nothing
- Always declare JavaScript variables with var, let , or const
- The var keyword is used in all JavaScript code from 1995 to 2015.
- The let and const keywords were added to JavaScript in 2015.
- If you want your code to run in older browser, you must use var
  - `let x = 5 + 2 + 3;`
  - `let person = "John Doe", carName = "Volvo", price = 200;`

JavaScript Let

- 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:

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

JavaScript Let

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

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

JavaScript Let

- Redeclaring Variables:

```
let x = 10;
// Here x is 10

{
let x = 2;
// Here x is 2
}

// Here x is 10
var x = 2;
// Now x is 2

var x = 3;
// Now x is 3
```

JavaScript Let

- Redeclaring Variables:

```
var x = 2;   // Allowed
let x = 3;   // Not allowed

{
let x = 2;   // Allowed
let x = 3;   // Not allowed
}

{
let x = 2;   // Allowed
var x = 3;   // Not allowed
}
```

JavaScript Let

- Redeclaring Variables:

```
let x = 2;   // Allowed

{
let x = 3;   // Allowed
}

{
let x = 4;    // Allowed
}
```

JavaScript Const

- When to use JavaScript const?
  - As a general rule, always declare a variable with `const` unless you know that the value will change
  - Use `const` when you declare:
    - A new Array
    - A new Object
    - A new Function
    - A new RegExp

JavaScript Const

- Constant Objects and Arrays
  - The keyword `const` is a little misleading
  - It does not define a constant value. It defines a constant reference to a value.
  - Because of this you can NOT:
    - Reassign a constant value
    - Reassign a constant array
    - Reassign a constant object
  - But you CAN:
    - Change the elements of constant array
    - Change the properties of constant object

JavaScript Const

- Constant Arrays
  - You can change the elements of a constant array:

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];

// You can change an element:
cars[0] = "Toyota";

// You can add an element:
cars.push("Audi");
console.log(cars);
```

  - But you can NOT reassign the array:

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

cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

JavaScript Const

- Constant Objects
  - You can change the properties of a constant object:

```
// You can create a const object:
const car = {type:"Fiat", model:"500", color:"white"};

// You can change a property:
car.color = "red";

// You can add a property:
car.owner = "Johnson";
console.log(car);
```

  - But you can NOT reassign the object:

```
const car = {type:"Fiat", model:"500", color:"white"};

car = {type:"Volvo", model:"EX60", color:"red"};    // ERROR
```

JavaScript Const

- Block Scope
  - Declaring a variable with `const` is similar to `let` when it comes to **Block Scope**.

```
const x = 10;
// Here x is 10

{
const x = 2;
// Here x is 2
}

// Here x is 10
```

JavaScript Operators

- JavaScript Arithmetic Operators

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

JavaScript Operators

- JavaScript 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
**=	x **= y	x = x ** y

JavaScript Operators

- JavaScript String Operators
  - ```
    let text1 = "John";
    let text2 = "Doe";
    let text3 = text1 + " " + text2; //John Doe
    ```
  - ```
    let text1 = "What a very ";
    text1 += "nice day"; //What a very nice day
    ```
  - ```
    let x = 5 + 5; //10
    let y = "5" + 5; //55  (string)
    let z = "Hello" + 5; //Hello5
    ```
  - If you add a number and a string, the result will be a string!
    ```

JavaScript Operators

- JavaScript Comparison Operators

Operator	Description	Comparing	Returns
==	equal to	x == 8	false
		x == 5	true
		x == "5"	true
===	equal value and equal type	x === 5	true
		x === "5"	false
!=	not equal	x != 8	true
!==	not equal value or not equal type	x !== 5	false
		x !== "5"	true
		x !== 8	true
>	greater than	x > 8	false
<	less than	x < 8	true
>=	greater than or equal to	x >= 8	false
<=	less than or equal to	x <= 8	true

JavaScript Operators

- JavaScript Logical Operators

Operator	Description
&&	logical and
||	logical or
!	logical not

JavaScript Data Types

- JavaScript variables can hold different data types: numbers, strings, objects and more:
  - ```
    let length = 16;                                  // Number
    let lastName = "Johnson";                         // String
    let x = {firstName:"John", lastName:"Doe"};       // Object
    ```
- JavaScript has dynamic types. This means that the same variable can be used to hold different data types:
  - ```
    let x;              // Now x is undefined
    x = 5;              // Now x is a Number
    x = "John";         // Now x is a String
    ```

JavaScript Data Types

- JavaScript 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:

```
let carName1 = "Volvo XC60";   // Using double quotes
let carName2 = 'Volvo XC60';   // Using 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";

// 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
```

JavaScript Data Types

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

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

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

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

JavaScript Data Types

- JavaScript Booleans

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

JavaScript Data Types

- JavaScript Arrays
  - JavaScript 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):

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

  - Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

```
const cars = ["Saab", "Volvo", "BMW"];
console.log(cars[0]); // Saab
```

JavaScript Data Types

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

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

  - The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

```
const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
console.log(person.firstName + ' ' + person.lastName);
// John Doe
console.log(person);
//{firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}
```

JavaScript Data Types

- The typeof Operator
  - You can use the JavaScript `typeof` operator to find the type of a JavaScript variable.
  - The `typeof` operator returns the type of a variable or an expression:

```
typeof ""               // Returns "string"
typeof "John"           // Returns "string"
typeof "John Doe"       // Returns "string“
typeof 0                // Returns "number"
typeof 314              // Returns "number"
typeof 3.14             // Returns "number"
typeof (3)              // Returns "number"
typeof (3 + 4)          // Returns "number"
```

JavaScript Data Types

- Undefined
  - In JavaScript, a variable without a value, has the value `undefined`. The type is also `undefined`.

```
let car;     // Value is undefined, type is undefined
```

  - Any variable can be emptied, by setting the value to `undefined`. The type will also be `undefined`.

```
car = undefined;     // Value is undefined, type is undefined
```

- Empty Values
  - An empty value has nothing to do with `undefined.`
  - An empty string has both a legal value and a type.

```
let car = "";     // The value is "", the typeof is "string"
```

JavaScript Functions

```
x=5
y=10
let j = myFunction(x, y);
Var x
Var y
// Function is called, return value will end up in x

function myFunction(a, b) {
  return a * b;
// Function returns the product of a and b
}
console.log(x); //12
```

JavaScript Objects

- Objects are variables too. But objects can contain many values.
- This code assigns **many values** (Fiat, 500, white) to a **variable** named car:

```
const car = {type:"Fiat", model:"500", color:"white"};
```

- The values are written as **name:value** pairs (name and value separated by a colon).
- It is a common practice to declare objects with the `const` keyword.

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

JavaScript Objects

- Object Properties
  - The **name:values** pairs in JavaScript objects are called **properties**:

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue

- Accessing Object Properties
  - You can access object properties in two ways:

```
objectName.propertyName (ex: person.lastName;)
objectName["propertyName"] (ex: person["lastName"];)
```

JavaScript Objects

- Object Methods
  - Objects can also have **methods**.
  - Methods are **actions** that can be performed on objects.
  - Methods are stored in properties as **function definitions**.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

  - A method is a function stored as a property.

JavaScript Objects

- Object Methods

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};

  console.log(person.fullName()); //accessing fullName method

  console.log(person);
```

- In the example above, `this` refers to the **person object**.
- Accessing Object Methods:

objectName.methodName()

JavaScript Strings

- String Length
  - To find the length of a string, use the built-in `length` property:

```
let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let length = text.length;
```

- Escape Character

Code	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

```
let text = "We are the so-called \"Vikings\" from the north.";
let text= 'It\'s alright.’;
let text = "The character \\ is called backslash.";
```

JavaScript Strings

- Extracting String Parts
  - There are 3 methods for extracting a part of a string:

```
slice (start, end)
substring (start, end)
substr (start, length)
```

- JavaScript String slice()
  - `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).

```
let str = "Apple, Banana, Kiwi";
let part1 = str.slice(7, 13);
console.log(part1); // Banana
let part2 = str.slice(7);
console.log(part2); // Banana, Kiwi
```

JavaScript Strings

- JavaScript String slice()
  - If a parameter is negative, the position is counted from the end of the string.

```
let str = "Apple, Banana, Kiwi";
let part3 = str.slice(-4);
console.log(part4); //Kiwi
let part4 = str.slice(-12, -6);
console.log(part4); // Banana
```

- JavaScript String substring()
  - `substring()` is similar to `slice()`
  - The difference is that start and end values less than 0 are treated as 0 in `substring()`

```
let str = "Apple, Banana, Kiwi";
let part1 = str.substring(-10, 5);
console.log(part1); // Apple
let part2 = str.substring(-10);
console.log(part2); // Apple, Banana, Kiwi
```

JavaScript Strings

- JavaScript String substr()
  - `substr ()` is similar to `slice()`
  - The difference is that the second parameter specifies the **length** of the extracted part
- Replacing String Content
  - The `replace()` method replaces a specified value with another value in a string:

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

  - Note:
    - The `replace()` method does not change the string it is called on.
    - The `replace()` method returns a new string.
    - The `replace()` method replaces **only the first** match

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

JavaScript Strings

- Replacing String Content
  - By default, the `replace()` method is case sensitive
  - To replace case insensitive, use a **regular expression** with an `/i` flag (insensitive):

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

  - To replace all matches, use a **regular expression** with a `/g` flag (global match):

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

JavaScript Strings

- JavaScript String toUpperCase()

```
let text1 = "Hello World!";
let text2 = text1.toUpperCase();
```

- JavaScript String toLowerCase()

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

// text2 is text1 converted to lower
```

JavaScript Strings

- JavaScript String concat()
  - `concat()` joins two or more strings:

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

  - The `concat()` method can be used instead of the plus operator. These two lines do the same:

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

- Methods: trim(), trimStart(), trimEnd()

```
let text1 = "     Hello World!     ";
let text2 = text1.trimEnd();
```

trimStart(), trimEnd() support from JS 2019

JavaScript Strings

- JavaScript String localeCompare()
  - `Syntax: localeCompare(`*`compareString`*`)`
  - `Result:`

    -1 if the string is sorted before the *compareString*
    0 if the two strings are equal
    1 if the string is sorted after the *compareString*

```
let text1 = "ab";
let text2 = "ab";
let result = text1.localeCompare(text2); //0
```

```
let text1 = "ab";
let text2 = "cd";
let result = text1.localeCompare(text2);//-1
```

JavaScript Strings

- Extracting String Characters
  - There are 3 methods for extracting string characters: `charAt (position)`, `charCodeAt (position)`, Property access [ ]
  - The `charAt()` method returns the character at a specified index (position) in a string:

```
let text = "HELLO WORLD";
let char = text.charAt(0);
```

  - The `charCodeAt()` method returns the unicode of the character at a specified index in a string. The method returns a UTF-16 code (an integer between 0 and 65535).

```
let text = "HELLO WORLD";
let char = text.charCodeAt(0);
```

  - ECMAScript 5 (2009) allows property access [ ] on strings:

```
let text = "HELLO WORLD";
let char = text[0];
```

JavaScript Strings

- Converting a String to an Array
  - A string can be converted to an array with the `split()` method:

```
text.split(",")    // Split on commas
text.split(" ")    // Split on spaces
text.split("|")    // Split on pipe
```

  - 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:

```
let text="Hello Word!";

let myAr=text.split("");

console.log(myAr[0]); //H

console.log(myAr); // ['H', 'e', 'l', 'l', 'o', ' ', 'W',
'o', 'r', 'd', '!']
```

JavaScript Strings

- JavaScript String Search
  - The `indexOf()` method returns the index of (the position of) the `first` occurrence of a specified text in a string

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

  - The `lastIndexOf()` method returns the index of the **last** occurrence of a specified text in a string

```
let str = "Please locate where 'locate' occurs!";
let i = str.lastIndexOf("locate");
console.log(i);//21
```

  - Both `indexOf()`, and `lastIndexOf()` return -1 if the text is not found

JavaScript Strings

- JavaScript String Search
  - Both methods accept a second parameter as the starting position for the search

```
let str = "Please locate where 'locate' occurs!";
let i = str.indexOf("locate“, 15);
console.log(i);//21
```

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

```
let str = "Please locate where 'locate' occurs!";
let i = str.search("locate“);
console.log(i);//7
```

JavaScript Strings

- JavaScript String Search
  - The two methods, `indexOf()` and `search()` are **NOT** equal
    - The `search()` method cannot take a second start position argument.
    - The `indexOf()` method cannot take powerful search values (regular expressions).
  - The `match()` method searches a string for a match against a regular expression, and returns the matches, as an Array object.
    - `let text = "The rain in SPAIN stays mainly in the plain";`
    - `const myAr=text.match(/ain/g);`
    - `console.log(myAr); // ['ain', 'ain', 'ain']`
  - The `includes()` method returns true if a string contains a specified value.

    string.includes(*searchvalue*, *start*)

JavaScript Template Literals

- **Template Literals** use back-ticks (``) rather than the quotes ("") to define a string

```
let text = `Hello World!`;
```

- With **template literals**, you can use both single and double quotes inside a string

```
let text = `He's often called "Johnny"`;
```

- **Template literals** allows multiline strings

```
let text =
  `The quick
  brown fox
  jumps over
  the lazy dog`;
```

JavaScript Template Literals

- **Template literals** allow variables in strings

```
let firstName = "John";
let lastName = "Doe";
let text = `Welcome ${firstName}, ${lastName}!`;
```

Automatic replacing of variables with real values is called **string interpolation**

```
let text = `He's often called "Johnny"`;
```

- **Template literals** allow expressions in strings

```
let price = 10;

let VAT = 0.25;

let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;

console.log(total); //12.50
```

Automatic replacing of expressions with real values is called **string interpolation**.

JavaScript Numbers

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

```
let x = 3.14;     // A number with decimals
let y = 3;        // A number without decimals
```

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

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

- Floating Precision
  - Floating point arithmetic is not always 100% accurate:

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

  - To solve the problem above, it helps to multiply and divide:

```
let x = (0.2 * 10 + 0.1 * 10) / 10; //0.3
```

JavaScript Numbers

- **Adding Numbers and Strings**
  - JavaScript uses the + operator for both addition and concatenation. Numbers are added. Strings are concatenated.
  - If you add two numbers, the result will be a number

```
let x = 10;
let y = 20;
let z = x + y;//30
```

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

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

JavaScript Numbers

- **Adding Numbers and Strings**
  - If you add a number and a string, the result will be a string concatenation

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

```
let x = 10;
let y = 20;
let z = "The result is: " + x + y; //1020
```

```
let x = 10;
let y = 20;
let z = "30";
let result = x + y + z; //3030
```

JavaScript Numbers

- **Number Strings**
  - JavaScript will try to convert strings to numbers in all numeric operations (except +)

```
let x = "100";
let y = "10";
let z = x * y; //1000
```

```
let x = "100";
let y = "10";
let z = x - y; //90
```

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

JavaScript Numbers

- **NaN - Not a Number**
  - `NaN` is a JavaScript reserved word indicating that a number is not a legal number.

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

  - You can use the global JavaScript function `isNaN()` to find out if a value is a not a number

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

console.log(isNaN(x)); //true
```

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

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

JavaScript Number Methods

- `toFixed()` returns a string, with the number written with a specified number of decimals

```
let x = 9.656;
x.toFixed(0); //10
x.toFixed(2); //9.66
x.toFixed(4); //9.6560
x.toFixed(6); //9.656000
```

- `Number()` can be used to convert JavaScript variables to numbers

```
Number(true); //1
Number(false); //0
Number("10"); //10
Number("  10"); //10
Number("10  "); //10
Number(" 10  "); //10
Number("10.33"); //10.33
Number("10,33"); //NaN
Number("10 33"); //NaN
Number("John"); //NaN
```

JavaScript Number Methods

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

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

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

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

JavaScript Arrays

- An array is a special variable, which can hold more than one value

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

- Spaces and line breaks are not important. A declaration can span multiple lines

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

- You can also create an array, and then provide the elements

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

JavaScript Arrays

- With JavaScript, the full array can be accessed by referring to the array name

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

console.log(cars);
```

- Arrays are a special type of objects. The `typeof` operator in JavaScript returns "object" for arrays
- Arrays use **numbers** to access its "elements". In this example, `person[0]` returns John

```
const person = ["John", "Doe", 46];
```

- Objects use **names** to access its "members". In this example, `person.firstName` returns John

```
const person = {firstName:"John", lastName:"Doe", age:46};
```

JavaScript Arrays

- You can have variables of different types in the same Array.
- You can have objects in an Array. You can have functions in an Array. You can have arrays in an Array

```
myArray[0] = Date.now;
myArray[1] = myFunction;
myArray[2] = myCars;
```

- The `length` property of an array returns the length of an array (the number of array elements)

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
console.log(fruits.length);//4
```

- To loop through an array, using `for` loop or `Array.forEach()` function

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
for (let i=0;i<fruits.length;i++)
console.log(`${i+1}. ${fruits[i]}`);
```

JavaScript Array Methods

- The `join()` method also joins all array elements into a string.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
//Banana * Orange * Apple * Mango
```

- The `pop()` method removes the last element from an array and returns the value that was "popped out".

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

- The `push()` adds a new element to an array (at the end) and returns the new array length.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.push("Kiwi");//5
```

JavaScript Array Methods

`splice(start, deleteCount?, string[])`

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

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi");
console.log(fruits);
//'Banana', 'Orange', 'Lemon', 'Kiwi', 'Apple', 'Mango’
```

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 2, "Lemon", "Kiwi");
console.log(fruits);//'Banana', 'Orange', 'Lemon', 'Kiwi'
```

JavaScript Array Methods

- With clever parameter setting, you can use `splice()` to remove elements without leaving "holes" in the array.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(0, 1);
console.log(fruits);//'Orange', 'Lemon', 'Kiwi'
```

JavaScript Array Methods

`slice(start?, end?)`

- The `slice()` method slices out a piece of an array into a new array.
- The `slice()` method creates a new array

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(1);
console.log(citrus); // 'Orange', 'Lemon', 'Apple', 'Mango’
```

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(2, 4); //not include 4
console.log(citrus);// 'Lemon', 'Apple'
```

JavaScript Array Sort

- The `sort()` method sorts an array alphabetically.

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

- The `sort()` method will produce incorrect result when sorting numbers. You can fix this by providing a **compare function**

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

JavaScript Array Iteration

- The `forEach()` method calls a function (a callback function) once for each array element.

```
const numbers = [45, 4, 9, 16, 25];
let sum=0;
numbers.forEach(myFunction);

function myFunction(value) {
  sum += value;
}
console.log(sum);//99
```

JavaScript Array Iteration

```
var Tinh_tong=(Danh_sach)=>{
    let Ket_qua=0;
    Danh_sach.forEach(n=>{
        Ket_qua += n;
    });
    return Ket_qua;
}

var Danh_sach_so=[12,8,3,7,4,15];
var Tong=Tinh_tong(Danh_sach_so);
console.log("Tổng các phần tử trong danh sách: " + Tong);
```

JavaScript Array Iteration

- The `map()` method creates a new array by performing a function on each array element.

```
const numbers1 = [45, 4, 9, 16, 25];
const numbers2 = numbers1.map(myFunction);

function myFunction(value) {
  return value * 2;
}
console.log(numbers1);//[45, 4, 9, 16, 25]
console.log(numbers2);//[90, 8, 18, 32, 50]
```

JavaScript Array Iteration

- The `filter()` method creates a new array with array elements that pass a test.
- Syntax:

```
filter(function(currentValue, index, arr), thisValue)
```

Parameter	Description
function()	Required. A function to run for each array element.
currentValue	Required. The value of the current element.
index	Optional. The index of the current element.
arr	Optional. The array of the current element.
thisValue	Optional. Default undefined A value passed to the function as its this value.

JavaScript Array Iteration

- The `filter()` method
- Ex:

```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);
function myFunction(value) {
  return value > 18;
}
console.log(over18);//45, 25
```

- The `indexOf(), lastIndexOf()`

```
arrayname.indexOf(item, [start])
arrayname.lastIndexOf(item, [start])
```

JavaScript Array Iteration

- `find()` method
  - The `find()` method returns the value of the first array element that passes a test function
  - Syntax:

    `find(function(currentValue, index, arr),thisValue)`

function()	Required. A function to run for each array element.
currentValue	Required. The value of the current element.
index	Optional. The index of the current element.
arr	Optional. The array of the current element.
thisValue	Optional. Default undefined. A value passed to the function as its this value.

JavaScript Array Iteration

- `find()` method
  - Ex:

```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);
function myFunction(value) {
  return value > 18;
}
console.log(first);//25
```

JavaScript Array Iteration

- `findIndex()` method
  - Syntax:

```
findIndex(function(currentValue, index, arr), thisValue)
```

Parameter	Description
function()	Required. A function to be run for each array element.
currentValue	Required. The value of the current element.
index	Optional. The index of the current element.
arr	Optional. The array of the current element.
thisValue	Optional. Default undefined. A value passed to the function as its this value.

JavaScript Array Iteration

- `findIndex()` method
  - Ex:

```
const array1 = [5, 12, 8, 130, 44];
const isLargeNumber = (element) => element > 13;
console.log(array1.findIndex(isLargeNumber));
// expected output: 3
```

JavaScript Array Iteration

- ECMAScript 2016 introduced `includes()` method to check if an element is present in an array (including NaN)
- Syntax: `includes(element, start)`

Parameter	Description
element	Required. The value to search for.
start	Optional. Start position. Default is 0.

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

JavaScript Array and Object Excercise

- Exercise 1
  - Complete the following code that can change digit to reading word.
  - For example, 12.3 => “one two dot three”

```
1
2   let number = '123.34'
3   let numToWord = {
4       '1': 'one',
5       '2': 'two'
6   }
7
8   for (var i = 0; i < number.length; i ++) {
9       console.log(number[i]);
10  }
11
```

- Then, write a function for this

JavaScript Array and Object Excercise

- Exercise 2
  - (1) Write down a function that sum every element in array. E.g. sumArray([12,3,4,1,2,3]) = 25
  - (2) Write function that count word size case-insensitively.
  - Input: "Hello world hello hello earth earth" (Not limited to these word, it can be any words)
  - Output: Object{hello : 3, world : 1, earth : 2 }

Destructing Arrays

- The old way of assigning array items to a variable:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const car = vehicles[0];
const truck = vehicles[1];
const suv = vehicles[2];
```

- The new way of assigning array items to a variable:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car, truck, suv] = vehicles;
```

- When destructuring arrays, the order that variables are declared is important.
- If we only want the car and suv we can simply leave out the truck but keep the comma:

```
const vehicles = ['mustang', 'f-150', 'expedition'];
const [car,, suv] = vehicles;
```

Destructing Arrays

- Destructuring comes in handy when a function returns an array:

```
function calculate(a, b) {
   const add = a + b;
   const subtract = a - b;
   const multiply = a * b;
   const divide = a / b;
   return [add, subtract, multiply, divide];
 }
 const [add, subtract, multiply, divide] = calculate(4, 7);
```

Destructing Objects

- The old way of using an object inside a function:

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}
myVehicle(vehicleOne);
function myVehicle(vehicle) {
    const message = 'My ' + vehicle.type + ' is a ' + vehicle.color + ' ' +
vehicle.brand + ' ' + vehicle.model + '.';
}
```

Destructing Objects

- The new way of using an object inside a function:

```
const vehicleOne = {
  brand: 'Ford',
  model: 'Mustang',
  type: 'car',
  year: 2021,
  color: 'red'
}
myVehicle(vehicleOne);
function myVehicle({type, color, brand, model}) {
  const message = 'My ' + type + ' is a ' + color + ' ' + brand + ' ' + model + '.';
}
```

- Notice that the object properties do not have to be declared in a specific order.

Spread Operater

- The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

```
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];
const numbersThree = […numbersOne, 9, 10];
```

- The spread operator is often used in combination with destructuring.
- Assign the first and second items from numbers to variables and put the rest in an array:

```
const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;
```

Spread Operater

- We can use the spread operator with objects too
- Combine these two objects:

```
const myVehicle = {
  brand: 'Ford',
  model: 'Mustang',
  color: 'red'
}
const updateMyVehicle = {
  type: 'car',
  year: 2021,
  color: 'yellow'
}
const myUpdatedVehicle = {...myVehicle, ...updateMyVehicle}
```

JavaScript Function Rest Parameter

- JavaScript does not have overloading function
- The **rest parameter** syntax allows a function to accept an indefinite number of arguments as an array, providing a way to represent overloading function
- Syntax:

```
function f(a, b, ...theArgs) {
  // …
}
```

- A function definition's last parameter can be prefixed with ... to be placed within a standard JavaScript array
- Only the last parameter in a function definition can be a rest parameter

JavaScript Function Rest Parameter

```
function myFun(a, b, ...manyMoreArgs) {
  console.log("a", a);
  console.log("b", b);
  console.log("manyMoreArgs", manyMoreArgs);
}

myFun("one", "two", "three", "four", "five", "six");

// Console Output:
// a, one
// b, two
// manyMoreArgs, ["three", "four", "five", "six"]
```

JavaScript Dates

- Date objects are created with the `new Date()` constructor.

```
new Date() //current date and time
new Date(year, month, day, hours, minutes, seconds,
milliseconds)
new Date(milliseconds)
new Date(date string)
```

- JavaScript counts months from **0** to **11**. Specifying a month higher than 11, will not result in an error but add the overflow to the next year
- JavaScript will (by default) output dates in full text string format

```
const d = new Date();
console.log(d);
```

JavaScript Date Get Methods

Method	Description
getFullYear()	Get the **year** as a four digit number (yyyy)
getMonth()	Get the **month** as a number (0-11)
getDate()	Get the **day** as a number (1-31)
getHours()	Get the **hour** (0-23)
getMinutes()	Get the **minute** (0-59)
getSeconds()	Get the **second** (0-59)
getMilliseconds()	Get the **millisecond** (0-999)
getTime()	Get the time (milliseconds since January 1, 1970)
getDay()	Get the weekday as a number (0-6)
Date.now()	Get the time. ECMAScript 5.

JavaScript Date Set Methods

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

JavaScript Math

- The Math object is static.
- JavaScript provides 8 mathematical constants that can be accessed as Math properties

```
Math.E          // returns Euler's number
Math.PI         // returns PI
Math.SQRT2      // returns the square root of 2
Math.SQRT1_2    // returns the square root of 1/2
Math.LN2        // returns the natural logarithm of 2
Math.LN10       // returns the natural logarithm of 10
Math.LOG2E      // returns base 2 logarithm of E
Math.LOG10E     // returns base 10 logarithm of E
```

JavaScript Math

Method	Description
abs(x)	Returns the absolute value of x
ceil(x)	Returns x, rounded upwards to the nearest integer
exp(x)	Returns the value of E^x
floor(x)	Returns x, rounded downwards to the nearest integer
max(x, y, z, ..., n)	Returns the number with the highest value
min(x, y, z, ..., n)	Returns the number with the lowest value
pow(x, y)	Returns the value of x to the power of y
random()	Returns a random number between 0 and 1
round(x)	Rounds x to the nearest integer
sign(x)	Returns if x is negative, null or positive (-1, 0, 1)
sqrt(x)	Returns the square root of x
trunc(x)	Returns the integer part of a number (x)

JavaScript If Else

```
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
}
```

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

JavaScript Switch case

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

```
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";
}
```

JavaScript Switch

- If you omit the break statement, the next case will be executed even if the evaluation does not match the case
- It is not necessary to break the last case in a switch block. The block breaks (ends) there anyway.
- The `default` keyword specifies the code to run if there is no case match

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

JavaScript Switch

- The `default` case does not have to be the last case in a switch block

```
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";
}
```

- If `default` is not the last case in the switch block, remember to end the default case with a break

JavaScript Switch

- Sometimes you will want different switch cases to use the same code

```
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";
}
```

JavaScript For Loop

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

```
for (var i = 0; i < 10; i++) {
  // some code
}

// Here i is 10
```

```
for (let i = 0; i < 10; i++) {
  // some code
}

// Here i is 5
```

JavaScript For In

```
for (key in object) {
  // code block to be executed
}
```

```
const person = {fname:"John", lname:"Doe", age:25};

let text = "";
for (let x in person) {
  text += person[x];
}
console.log(text);
```

JavaScript For In

```
for (variable in array) {
  code
}
```

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
for (let x in numbers) {
  txt += numbers[x];
}
```

JavaScript For Of

```
for (variable of  iterable) {
  code
}
```

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

let text = "";
for (let x of cars) {
  text += x;
}
```

JavaScript While Loop

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


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

JavaScript Errors

- The `try` statement defines a code block to run (to try).
- The `catch` statement defines a code block to handle any error.
- The `finally` statement defines a code block to run regardless of the result.
- The `throw` statement defines a custom error.

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of
the try / catch result
}
```

JavaScript Errors

```
try{
    myFunction(12);
}
catch (er){
    console.log(er);
}
// too high
function myFunction(x) {
        if(x == "") throw "empty";
        if(isNaN(x)) throw "not a number";
        x = Number(x);
        if(x < 5) throw "too low";
        if(x > 10) throw "too high";
    }
```

JavaScript Hoisting

- Hoisting is JavaScript's default behavior of moving declarations to the top.
- In JavaScript, a variable can be declared after it has been used.
- In other words; a variable can be used before it has been declared.
- Ex1:

```
x=5
console.log(x+10) //15
```

- Ex2:

```
x=5
console.log(x+10) //15
var x
```

- Ex3:

```
x=5
console.log(x+10)
let x
//Error
```

JavaScript Hoisting

- One of the advantages of hoisting is that it lets you use a function before you declare it in your code.

```
catName("Tiger");
function catName(name) {
   console.log(`My cat's name is ${name}`);
} //My cat’s name is Tiger
```

- Arrow function are not hoisting

```
catName("Tiger");
var catName =(name) =>{
  console.log(`My cat's name is ${name}`);//error
}

var catName =(name) =>{
  console.log(`My cat's name is ${name}`);
}
catName("Tiger"); //ok
```

JavaScript Strict Mode

- `"use strict";` Defines that JavaScript code should be executed in "strict mode".
- With strict mode, you can not, for example, use undeclared variables.

```
"use strict";
x = 3.14;

// This will cause an error because x is not declared

myFunction();

function myFunction() {
  "use strict";
  y = 3.14;   // This will cause an error
}
```

JavaScript Arrow Function

- Arrow functions were introduced in ES6.
- Arrow functions allow us to write shorter function syntax

```
let myFunction = (a, b) => {if (a>b) return a
    else return b;}
console.log(myFunction(4,6));
```

- Before Arrow:

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

- With Arrow Function:

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

JavaScript Arrow Function

- Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
//This works only if the function has only one statement.
```

- If you have parameters, you pass them inside the parentheses

```
let add = (val1, val2) =>val1+val2;
console.log(add(3,7));
```

- if you have only one parameter, you can skip the parentheses as well:

```
hello = val => "Hello " + val;
console.log(hello("Mary"));
```

JavaScript Arrow Function

```
function fn1(a){
    return a*3
}
//const fn1=a=>a*3
const fn2=function(a){
    return a/5
}
const fn3=(a,b)=>{
    return a+b
}
console.log(fn3(fn1(fn2(2)),fn1(fn2(10))))
```

JavaScript Classes

- Use the keyword `class` to create a class
- Always add a method named `constructor()`

```
class ClassName {
  constructor() { ... }
}
```

- The “Car” class has two initial properties: "name" and "year"

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
```

JavaScript Classes

- When you have a class, you can use the class to create objects

```
let myCar1 = new Car("Ford", 2014);
let myCar2 = new Car("Audi", 2019);
```

- The Constructor Method:
  - The constructor method is a special method:
  - It has to have the exact name "constructor"
  - It is executed automatically when a new object is created
  - It is used to initialize object properties
  - If you do not define a constructor method, JavaScript will add an empty constructor method.

JavaScript Classes

- Class Methods:

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

```
class Car {
    constructor(name, year) {
      this.name = name;
      this.year = year;
    }
    age() {
      let date = new Date();
      return date.getFullYear() -
this.year;
    }
  }

  let myCar = new Car("Ford", 2014);
  console.log("My car is " + myCar.age() +
" years old.");
```

JavaScript Classes

- **Class Inheritance**:
- To create a class inheritance, use the `extends` keyword.

```
class Car {
  constructor(brand) {
    this.carname = brand;
  }
  present() {
    return 'I have a ' + this.carname;
  }
}
class Model extends Car {
  constructor(brand, mod) {
    super(brand);
    this.model = mod;
  }
  show() {
    return this.present() + ', it is a
' + this.model;
  }
}
let m = new Model("Ford",2022)
console.log(m)//Model {carname: 'Ford', model: 2022}
```

JavaScript Classes

- **Override method**:

```
class Animal {
    constructor(name) {
      this.speed = 0;
      this.name = name;
    }
   run(speed) {
      this.speed = speed;
      console.log(`${this.name} runs with speed ${this.speed}.`);
    }
   stop() {
      this.speed = 0;
      console.log(`${this.name} stands still.`);
    }
}
```

JavaScript Classes

- **Override method**:

```
class Rabbit extends Animal {
    hide() {
        console.log(`${this.name} hides!`);
    }

    stop() {
      super.stop(); // call parent stop
      this.hide(); // and then hide
    }
  }
  let rabbit = new Rabbit("White Rabbit");
  rabbit.run(5); // White Rabbit runs with speed 5.
  rabbit.stop(); // White Rabbit stands still. White Rabbit hides!
```

JavaScript Modules

- Modules:
  - JavaScript modules allow you to break up your code into separate files.
  - This makes it easier to maintain the code-base.
  - JavaScript modules rely on the `import` and `export` statements

File person.js:

```
export const name = "Jesse";
export const age = 40;
```

File index.js:

```
import { name, age } from "./person.js";
```

JavaScript Modules

- Export:
  - There are two types of exports: Named and Default.
  - Named Export: You can create named exports two ways: in-line individually, or all at once at the bottom
    - In-line individually:

File person.js:

```
export const name = "Jesse";
export const age = 40;
```

    - All at once at the bottom:

File person.js:

```
const name = "Jesse";
const age = 40;
export {name, age};
```

JavaScript Modules

- Export:
  - Default Export: You can only have one default export in a file

File message.js :

```
const message = () => {
const name = "Jesse";
const age = 40;
return name + ' is ' + age + 'years old.';
};

export default message;
```

JavaScript Modules

- Import:
  - You can import modules into a file in two ways, based on if they are named exports or default exports.
  - Named exports are constructed using curly braces. Default exports are not.
  - Import from named exports

```
import { name, age } from "./person.js";
```

  - Import from default exports

```
import message from "./message.js";
```

JavaScript Modules

- Export – Import Example test in Visual Studio Code:
  - Create JSON file package.json:

```
{
 "type": "module"
}
```

  - File message.js:

```
const message = () => {
const name = "Jesse";
const age = 40;
return name + ' is ' + age + ’ years old.';
};

export default message;
```

JavaScript Modules

- Export – Import Example test in Visual Studio Code:
  - File index.js:

```
import message from "./message.js";

console.log(message());
 //Jesse is 40 years old.
```

JavaScript JSON

- JSON:
  - JSON is a format for storing and transporting data
  - JSON stands for **J**ava**S**cript **O**bject **N**otation
  - JSON is a lightweight data interchange format
  - JSON is "self-describing" and easy to understand
  - This JSON syntax defines an employees object: an array of 3 employee records (objects)

```
{
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
}
```

JavaScript JSON

- JSON Syntax Rules:
  - Data is in name/value pairs
  - Data is separated by commas
  - Curly braces hold objects
- JSON Objects:
  - JSON objects are written inside curly braces.
  - Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

JavaScript JSON

- JSON Arrays:
  - JSON arrays are written inside square brackets.
  - Just like in JavaScript, an array can contain objects:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

JavaScript JSON

- Converting a JSON Text to a JavaScript Object: use the JavaScript built-in function `JSON.parse()`

```
let text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
const obj = JSON.parse(text);
console.log(obj.employees[1].firstName + " " +
obj.employees[1].lastName);//Anna Smith
```

Exercise

Cho bảng dữ liệu về Các mặt hàng Tivi như sau:

Ma_so	Ten	Don_gia
LG	Tivi LG	500000
SAMSUNG	Tivi Samsung	150000
SONY	Tivi Sony	450000
KHAC	Tivi Khác	250000

a. Tạo biến mảng Danh_sach_Nhom_Tivi để lưu dữ liệu trên
b. Sắp xếp tăng dần theo đơn giá, in ra mã số và đơn giá
c. Sắp xếp tăng dần theo mã số, in ra mã số và tên theo cú pháp Template literal

Exercise

d. Tìm và in thông tin về mặt hàng có mã số LG
e. Tìm và in vị trí của mặt hàng có mã số LG trong mảng
f. Lọc danh sách mặt hàng trong tên có chứa ký tự ‘s’
g. Thêm mặt hàng {TOSHIBA, Tivi Toshiba, 300000} vào mảng
h. Xóa mặt hàng ‘Tivi khác’ khỏi mảng (dùng splice). Cho biết tổng số phần tử còn lại

More Exercise

JavaScript Exercises, Practice, Solution - w3resource

(https://www.w3resource.com/javascript-exercises/)