Function

Block of code that performs a specific task, can be invoked whenever needed

Function Definition

function functionName( ) {

//do some work

}

function functionName(param1, param2 ...) {
//do some work

}

Function Call

functionName( );

e.g.

```
// function to find square of a number
function findSquare(num) {
    // return square
    return num * num;
}
// call the function and store the result
let square = findSquare(3);
```

Arrow Function

Compact way of writing a function

const functionName = (param1, param2 ...) => { //do some work }

e.g.

```
const square = x => x * x;

// use the arrow function to square a number
console.log(square(5));
// Output: 25

let addition = (a, b) => a + b;
let sum = addition(30, 25);
console.log("Sum of given numbers is :", sum);
//Output: 55
```

Array Methods

1. **`map()`**

   Creates a new array with the results of calling a provided function on every element in the calling array.

   ```
   const numbers = [1, 2, 3, 4];
   const doubled = numbers.map(number => number * 2);
   console.log(doubled); // [2, 4, 6, 8]
   ```

2. **`filter()`**

   Creates a new array with all elements that pass the test implemented by the provided function.

   ```
   const numbers = [1, 2, 3, 4];
   const evens = numbers.filter(number => number % 2 === 0);
   console.log(evens); // [2, 4]
   ```

3. **`reduce()`**

   Executes a reducer function on each element of the array, resulting in a single output value.

   ```
   const numbers = [1, 2, 3, 4];
   const sum = numbers.reduce((accumulator, current) => accumulator +
   current, 0);
   console.log(sum); // 10
   ```

4. **`forEach()`**

   Executes a provided function once for each array element.

   ```
   arr.forEach(function(item, index, array) {
     // ... do something with an item
   });
   ```

   E.g.

   ```
   const numbers = [1, 2, 3, 4];
   numbers.forEach(number => console.log(number));
   // 1
   // 2
   // 3
   // 4
   ```

5. **`some()`**

   Tests whether at least one element in the array passes the test implemented by the provided function.

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

```
const hasEven = numbers.some(number => number % 2 === 0);
console.log(hasEven); // true
```

6. **every()**

   Tests whether all elements in the array pass the test implemented by the provided function.

```
const numbers = [1, 2, 3, 4];
const allEven = numbers.every(number => number % 2 === 0);
console.log(allEven); // false
```

7. Sort()

   Sorts the array *in place*, changing its element order.

```
let city = ["California", "Barcelona", "Paris", "Kathmandu"];

// sort the city array in ascending order
let sortedArray = city.sort();
console.log(sortedArray);

// Output: [ 'Barcelona', 'California', 'Kathmandu', 'Paris' ]
```

8. **find()**

   Returns the value of the first element in the array that satisfies the provided testing function.

```
const numbers = [1, 2, 3, 4];
const firstEven = numbers.find(number => number % 2 === 0);
console.log(firstEven); // 2
```

9. **findIndex()**

   Returns the index of the first element in the array that satisfies the provided testing function.

```
const numbers = [1, 2, 3, 4];
const firstEvenIndex = numbers.findIndex(number => number % 2 === 0);
console.log(firstEvenIndex); // 1
```

10. **slice()**

    Returns a shallow copy of a portion of an array into a new array object selected from start to end (end not included).

```
const numbers = [1, 2, 3, 4, 5];
const sliced = numbers.slice(1, 3);
console.log(sliced); // [2, 3]
```

11. **splice()**

Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

```
const numbers = [1, 2, 3, 4, 5];
const removed = numbers.splice(1, 2);
console.log(removed); // [2, 3]
console.log(numbers); // [1, 4, 5]
```

Object Methods

1. **`Object.keys()`**

   Returns an array of a given object's own enumerable property names.

   ```
   const person = { name: 'John', age: 30, city: 'New York' };
   const keys = Object.keys(person);
   console.log(keys); // ['name', 'age', 'city']
   ```

2. **`Object.values()`**

   Returns an array of a given object's own enumerable property values.

   ```
   const person = { name: 'John', age: 30, city: 'New York' };
   const values = Object.values(person);
   console.log(values); // ['John', 30, 'New York']
   ```

3. **`Object.entries()`**

   Returns an array of a given object's own enumerable property [key, value] pairs.

   ```
   const person = { name: 'John', age: 30, city: 'New York' };
   const entries = Object.entries(person);
   console.log(entries); // [['name', 'John'], ['age', 30], ['city',
   'New York']]
   ```

4. **`Object.assign()`**

   Copies all enumerable own properties from one or more source objects to a target object. It returns the modified target object.

   ```
   const target = { a: 1, b: 2 };
   const source = { b: 4, c: 5 };
   const returnedTarget = Object.assign(target, source);
   console.log(returnedTarget); // { a: 1, b: 4, c: 5 }
   console.log(target); // { a: 1, b: 4, c: 5 }
   ```

5. **Object spreading (`...`)**

   Allows copying properties from one object to another.

   ```
   const obj1 = { a: 1, b: 2 };
   const obj2 = { b: 3, c: 4 };
   const merged = { ...obj1, ...obj2 };
   console.log(merged); // { a: 1, b: 3, c: 4 }
   ```

Combining Arrays and Objects

Here’s an example that combines some of these methods to manipulate data:

```
const users = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 35 }
];

// Map to create a new array of user names
const userNames = users.map(user => user.name);
console.log(userNames); // ['Alice', 'Bob', 'Charlie']

// Filter to find users older than 30
const olderUsers = users.filter(user => user.age > 30);
console.log(olderUsers); // [{ name: 'Charlie', age: 35 }]

// Reduce to get the total age of all users
const totalAge = users.reduce((sum, user) => sum + user.age, 0);
console.log(totalAge); // 90

// Find a user named 'Bob'
const userBob = users.find(user => user.name === 'Bob');
console.log(userBob); // { name: 'Bob', age: 30 }

// Object methods to work with a single user
const user = { name: 'Alice', age: 25, city: 'Wonderland' };
const keys = Object.keys(user);
console.log(keys); // ['name', 'age', 'city']

const values = Object.values(user);
console.log(values); // ['Alice', 25, 'Wonderland']

const entries = Object.entries(user);
console.log(entries); // [['name', 'Alice'], ['age', 25], ['city',
'Wonderland']]
```

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.

Methods and properties are:

- 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 map.
- map.has(key) – returns true if the key exists, false otherwise.
- map.delete(key) – removes the element (the key/value pair) by the key.
- map.clear() – removes everything from the map.
- map.size – returns the current element count.

Chaining

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

```
map.set('1', 'str1'); //a string key
  .set(1, 'num1');  // a numeric key
  .set(true, 'bool1'); //a Boolean key
```

Iteration over Map:

For looping over a map, there are 3 methods:

- map.keys() – returns an iterable for keys,
- map.values() – returns an iterable for values,
- map.entries() – returns an iterable for entries [key, value], it's used by default in for..of

The insertion order is used

The iteration goes in the same order as the values were inserted. Map preserves this order, unlike a regular Object.

Object.entries: Map from Object

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

Object.fromEntries: Object from Map:

We've just seen how to create Map from a plain object with Object.entries(obj).

There's Object.fromEntries method that does the reverse: given an array of [key, value] pairs, it. Creates an object from them:

e.g.

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['apple', 4]
]);

// now prices = { banana: 1, orange: 2, apple: 4 }
alert(prices.orange); // 2
```

We can use Object.fromEntries to get a plain object from Map.

E.g. we store the data in a Map, but we need to pass it to a 3rd-party code that expects a plain object.

Here we go:

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('apple', 4);


let obj = Object.fromEntries(map.entries()); // make a plain object
(*)


// done!
// obj = { banana: 1, orange: 2, apple: 4 }


alert(obj.orange); // 2
```

Set

A Set is a special type collection – “set of values” (without keys), where each value may occur only once. Its main methods 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.delete(value) – removes the value, returns true if value existed at the moment of the call, otherwise false.
- set.has(value) – returns true if the value exists in the set, otherwise false.
- set.clear() – removes everything from the set.
- set.size – is the elements count.

Using Set in React-based applications can be beneficial for handling unique collections of data, such as managing a list of unique items, preventing duplicate entries, and efficiently performing operations like union, intersection, and difference.

For example, we have visitors coming, and we’d like to remember everyone. But repeated visits should not lead to duplicates. A visitor must be “counted” only once.

Set is just the right thing for that:

Iteration over Set:

We can loop over a set either with `for..of` or using `forEach`: