sourcemind

FUNCTIONS IN DEPTH

Functions

JavaScript Function Syntax

A JavaScript function is defined with the `function` keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:
(parameter1, parameter2, ...)

The code to be executed, by the function, is placed inside curly brackets: **{}**

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

Function Types

- Function Declaration
- Function Expression
- Arrow Function

```
// Function declaration
function add(a, b) {
    console.log(a + b);
}

// Calling a function
add(2, 3);
```

```
// Function Expression
const add = function(a, b) {
    console.log(a+b);
}

// Calling function
add(2, 3);
```

Arrow Function

```
// Single line of code
let add = (a, b) => a + b;

console.log(add(3, 2));
```

```
// Multiple line of code
const great = (a, b) => {
    if (a > b)
        return "a is greater";
    else
        return "b is greater";
}

console.log(great(3,5));
```

sourcemind

Self-Invoking Functions in JavaScript

```
(function () {
  // body
}());
```

```
(function (w, d, $) {
  // body
}(window, document, jQuery));
```

sourcemind

Function this(example)

```
function getAge() {
  console.log(this.age);
}

const user = {
  name: "User",
  age: 16,
  getAge: getAge
}

const age = user.getAge;
const ageNew = new getAge();

getAge();
user.getAge();
age();
ageNew();
```

sourcemind

call(), apply(), bind(), - “this” refresher

The **bind()** method creates a new function that, when called, has its this keyword set to the provided value.

The **call()** method calls a function with a given this value and arguments provided individually.

call() and **apply()** serve the **exact same purpose.** The ***only difference between how they work is that*** call() expects all parameters to be passed in individually, whereas apply() expects an array of all of our parameters.

Example:

```
const person = {
    name: "Armen",
    age: 25,
}

function getPersonInfo(gender){
      console.log(`${this.name} is ${this.age} years old.`);
      console.log(`Gender is ${gender}`)
}

getPersonInfo.call(person, "MALE");
getPersonInfo.apply(person, ["MALE"]);
```

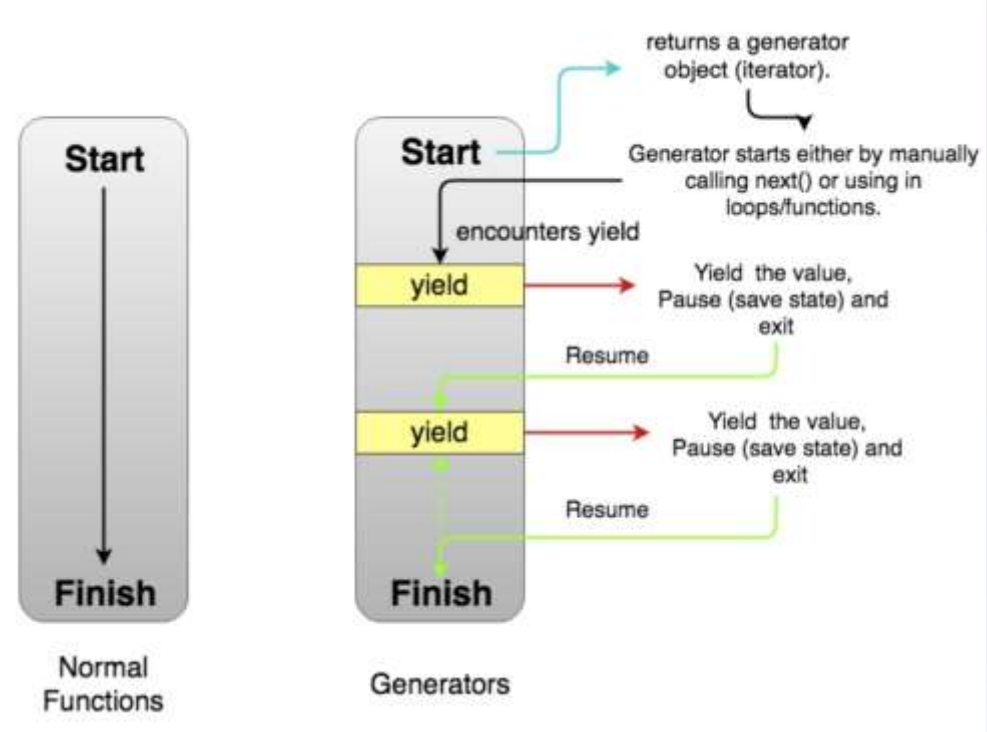
sourcemind

Generator functions

ES6 introduced a new way of working with functions and iterators in the form of **Generators (or generator functions)**. A generator is a function that **can stop midway** and then continue *from where it stopped.* **In short, a generator *appears* to be a function but it *behaves* like an iterator.**

A normal function such as this one cannot be stopped *before* it finishes its task i.e its last line is executed

```
function normalFunc() {
  console.log('I')
  console.log('cannot')
  console.log('be')
  console.log('stopped.')
}
```



sourcemind

Generator function

```
function * generatorFunction() { // Line 1
  console.log('This will be executed first.');
  yield 'Hello, ';   // Line 2

  console.log('I will be printed after the pause');
  yield 'World!';
}

const generatorObject = generatorFunction(); // Line 3

console.log(generatorObject.next().value); // Line 4
console.log(generatorObject.next().value); // Line 5
console.log(generatorObject.next().value); // Line 6

// This will be executed first.
// Hello,
// I will be printed after the pause
// World!
// undefined
```

In JavaScript, a generator is a function which returns an object on which you can call next().
Every invocation of next() will return an object of shape —

```
{
  value: Any,
  done: true|false
}
```

sourcemind

Example

```
function* generatorFunction(){
    yield 1;
    yield 2;
    yield 3;
}

const generator = generatorFunction()

console.log(generator.next().value) // 1
console.log(generator.next().value) // 2
console.log(generator.next().value) // 3
```

Example

```
function* generatorFunction(){
    yield 1;
    yield 2;
    yield 3;
}

const generator = generatorFunction();

let done = false;

while(!done){
    console.log(generator.next().value);
    done = generator.next().done;
}
```

Example

```
function* generatorFunction(){
    yield 1;
    yield 2;
    yield 3;
}

const generator = generatorFunction();

for(value of generatorFunction()){
    console.log(value);
}
```

```
const a = [...generatorFunction()];

console.log(a);
```

Iterable Object

```
const user =  {
   name: "Aram"
}

for(let key of user) {
    console.log(key)
}
```

```
for(let key of user) {
                 ^

TypeError: user is not iterable
    at Object.<anonymous> (/tmp/LqH43gRmYk.js:62:16
```

sourcemind

Iterable Object

```
const user =  {
   name: "Aram"
}

for(let key of user) {
    console.log(key)
}
```

```
for(let key of user) {
                 ^

TypeError: user is not iterable
    at Object.<anonymous> (/tmp/LqH43gRmYk.js:62:16
```

```
user[Symbol.iterator] = function *() {
    yield 1;
    yield 2;
    yield 3
}
```

sourcemind

Iterable Object

```
const range = {
    start: 1,
    end: 10
}

for(item of range) {
    console.log(item)
}
```

Iterable Object

```
const range = {
    start: 5,
    end: 10
}




range[Symbol.iterator] = function* () {
  for(let i= this.start; i <= this.end; i++) {
      yield i
  }
}

for(item of range) {
    console.log(item)
}
```

sourcemind

Iterable Object

```
const range = {
    start: 1,
    end: 10,
    [Symbol.iterator](){
        return {
            current: this.start,
            end: this.end,
            next() {
                if (this.current <= this.end) {
                    return {done: false, value: this.current++}
                }
                return {done: true, value: this.current}
            }
        }
    }
}

for(item of range) {
    console.log(item)
}
```

sourcemind

Task

1. Create Range class which instance is iterable
   1.1 Class should receive from, to arguments

```
for (item of new Range(10, 20)) {
    console.log(item)
}
```

Tasks

1. Write a JavaScript function that returns a passed string with letters in alphabetical order.
2. Write a JavaScript function to find the first not repeated character.
3. Write a JavaScript function that accept a list of country names as input and returns the longest country name as output.
4. Create doubleValues function which available for every array (like map, forEach) which return new array with duplicated values.