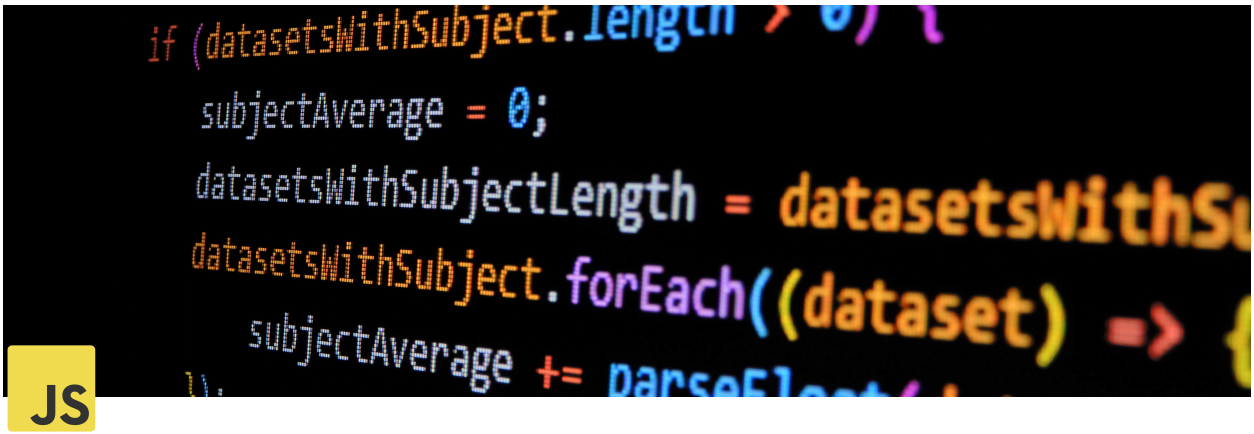

Javascript Questions

Most of the important topics have been covered that is almost asked in all the interviews, so prepare all these concepts well. Best of luck for you Interviews.

NOTE: - We update the Question Sets monthly.

▼ What are closures? Advantage and Disadvantage

Closure is a combination of functions bundled together in a lexical scope. The inner function has access to an outer function scope, variables and parameters even after the outer function has returned.

7 Question - https://dmitripavlutin.com/javascript-closures-interview-questions/#questions-1-closures-raise-your-hand

```
function outerFunc() {
  let outerVar = 'I am outside!';
  return function innerFunc() {
    console.log(outerVar); // => logs "I am outside!"
  }
}


function exec() {
  const myInnerFunc = outerFunc();
  myInnerFunc();
}
exec();
```

Advantages of closures:

1. Callbacks implementation in javascript is heavily dependent on how closures work
2. Mostly used in Encapsulation of the code
3. Also used in creating API calling wrapper methods

Disadvantages of closures:

1. Variables used by closure will not be garbage collected
2. Memory snapshot of the application will be increased if closures are not used properly

▼ Difference between regular function vs arrow function?

1. Syntax - No implicit return for arrow function
2. Argument Binding - In regular function, Arguments keywords can be used to access the arguments of which passed to function.

```
function regularFunction(a,b) {
    console.log(arguments)
}
regularFunction(1,2)
// Arguments[1,2]
```

```
const arrowFunction = (a,b) => {
    console.log(arguments)
}
arrowFunction(1,2)
//ReferenceError: arguments is not defined


// workaround for arrow function
var arrowFunction = (...args) => {
    console.log(...args)
}
arrowFunction(1,2)
// 1 2
```

3. this keyword - Normal function depends on how it is being called and arrow function is on where it is being called.
4. New keyword - Regular functions are constructible, they can be called using the new keyword.

```
function add (x, y) {
    console.log(x + y)
}
let sum = new add(2,3);
// 5
```

```
let add = (x, y) => console.log(x + y);
const sum = new add(2,4);
// TypeError: add is not a constructor
```

5. Function Hoisting - In regular function, function gets hoisting at top.

```
normalFunc()
function normalFunc() {
    return "Normal Function"
}
// "Normal Function"
```

In arrow function, function get hoisted where you define. So, if you call the function before initialisation you will get referenceError.

```
arrowFunc()
const arrowFunc = () => {
    return "Arrow Function"
}
// ReferenceError: Cannot access 'arrowFunc' before initialization
```

▼ What is Lexical Environment / lexical Scope?

Lexical scoping means that the accessibility of variables is determined by the position of the variables inside the nested scopes.

Simpler, the lexical scoping means that inside the inner scope you can access variables of outer scopes.

It's called *lexical* (or *static*) because the engine determines (at lexing time) the nesting of scopes just by looking at the JavaScript source code, without executing it.

```
function one(){

  function two(){
   console.log(a);
  }

  var a=2;
  console.log(a);
  two();
}

var a=1;
console.log(a);
one();
```

Output for the above code is `1 2 2`

In the above example in the case of function two, its outer lexical environment is function one's execution context and for function one, the outer lexical environment is the global execution context.

When javascript asked for the value of var a in function two's execution context it couldn't find it so it moved down and searched it in its outer lexical environment i.e. function one in this case.

▼ What is Currying?

Currying is the process of taking a function with multiple arguments and turning it into a sequence of functions each with only one single arguments.

```
function multiply(a) {
  return function executeMultiply(b) {
    return a * b;
  }
}
const double = multiply(2);
double(3); // => 6
double(5); // => 10
const triple = multiply(3);
triple(4); // => 12


currying upto n number
======================
function sum(a){
  return (b)=>{
    return b ? sum(a+b) : a
  }
```

```
}

sum(2)(3)(4)(5)(6)()



const createURL = baseURL => {
  const protocol = "https";

  // we now return a function, that accepts a 'path' as an argument
  return path => {
    return `${protocol}://${baseURL}/${path}`;
  };
};

// we create a new functions with the baseURL value in it's closure scope
const createSiteURL = createURL("mysite.com");

// create URLs for our main site
const homeURL = createSiteURL("");
const loginURL = createSiteURL("login");
const productsURL = createSiteURL("products");
const contactURL = createSiteURL("contact-us");

console.log(loginURL)
```

▼ Difference between Call, Apply and Bind?

Call and Apply calls a function, while bind returns a new function. Arguments are passed individually on call while apply expects an array of arguments.

call and apply➖

```


let participant1 = {
  name:"lily",
  battery: 40,
  chargeBattery: function(a,b){
    this.battery = this.battery + a + b;
  }
}

let participant2 = {
  name:"john",
  battery: 30
}

participant1.chargeBattery.call(participant2, 20, 30)
participant1.chargeBattery.apply(participant2, [ 20, 30 ])

-----------------------------------------------------------------------------------------------
var pokemon = {
    firstname: 'Pika',
    lastname: 'Chu ',
    getPokeName: function() {
        var fullname = this.firstname + ' ' + this.lastname;
        return fullname;
    }
};

var pokemonName = function(snack, hobby) {
    console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
};

pokemonName.call(pokemon,'sushi', 'algorithms'); // Pika Chu  loves sushi and algorithms
pokemonName.apply(pokemon,['sushi', 'algorithms']); // Pika Chu  loves sushi and algorithms
```

Bind -

```
var pokemon = {
    firstname: 'Pika',
    lastname: 'Chu ',
    getPokeName: function() {
        var fullname = this.firstname + ' ' + this.lastname;
        return fullname;
    }
};

var pokemonName = function(snack, hobby) {
    console.log(this.getPokeName() + ' loves ' + snack + ' and ' + hobby);
};

var logPokemon = pokemonName.bind(pokemon); // creates new object and binds pokemon. 'this' of pokemon === pokemon now

logPokemon('sushi', 'algorithms'); // Pika Chu  loves sushi and algorithms
```

▼ What is an Event Loop?

An event loop is something that pulls the task of the callback queue / microtask queue and places it onto the execution call stack whenever the call stack becomes empty

Refer to this video for a clear understanding on Event Loop and will help you to answer every questions on even loop -

https://www.youtube.com/watch?v=28AXSTCpsyU

▼ Prototype in Javascript?

Prototype is mechanism by which javascript object inherit features from another object.

```
// we create a generic function with firstname and lastname
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

//we inherited the properties of the function and made a prototype of it
Person.prototype.getFullName = function(){
  return this.firstName + this.lastName
}

//here we are creating an new constructor function with different arguments
let lydia = new Person("Lydia", "Hallie");

lydia.getFullName() // This now works!
```

▼ what is memoization?

It is a technique used to speed up a program by storing the result of expensive function calls and returning the cached results when same input occurs again.

Code to implement memoization on a function

```
const Memo = (func) => {
    let cache = {}
    return (...args)=>{
      const value = JSON.stringify(args);
      if(value in cache){
        console.log("From Cache")
```

```
        return cache[value]
      } else {
        //caching it first
        console.log("without Cache")
        cache[value] = func.apply(this, args)
        return cache[value]
      }
    }
  }

const memoized = Memo(pass your own function here)

memoized(pass function arguments)
```

▼ What is higher order functions?

It is a function that accepts another function as an argument and even can return the function as a value.

#Higher order functions A **higher-order** function is a function that does at least one of the following:

- Takes one or more functions as an input
- Outputs a function

```
var animals = [
  {name: 'Milo', type: 'cat', eyes: 2},
  {name: 'Peluza', type: 'cat', eyes: 2},
  {name: 'Yummy', type: 'cat', eyes: 2},
  {name: 'Doggy', type: 'dog', eyes: 2},
  {name: 'Sammy', type: 'dog', eyes: 2}
];

// Using map method: Map It iterates over an array and return a new array with the changes you do to it.

const getAnimals = animals.map(animal => `${animal.name} is a ${animal.type}`);

getAnimals = [ 'Milo is a cat',
                'Peluza is a cat',
                'Yummy is a cat',
                'Doggy is a dog',
                'Sammy is a dog'
              ]

So Map, filter are examples of Higher order functions
```

▼ What is event delegation?

It is a useful pattern of adding an event listener to a single element (parent) instead of assigning events on multiple elements.

Event bubbling provides the foundation for event delegation in browsers. Now you can bind an event handler to a single parent element, and that handler will get executed whenever the event occurs *on any of its child nodes* (and any of their children in turn). **This is event delegation.**

Here's an example of it in practice:

```
<ul onclick="alert(event.type + '!')">
    <li>One</li>
    <li>Two</li>
    <li>Three</li>
</ul>
```

With that example if you were to click on any of the child `<li>` nodes, you would see an alert of `"click!"`, even though there is no click handler bound to the `<li>` you clicked on. If we bound `onclick="..."` to each `<li>` you would get the same effect.

▼ What is promises?

Promise is an object that produce a single value. With promises, we can postpone the execution of a code block until an asynchronous operation is completed. This way, other operations can keep running without interruption.

Promises have three states:

1. Pending: the initial state of the promise

2. Fulfilled: the specified operation was completed.

3. Rejected: the operation did not complete.

```
let prom = new Promise((res, rej) => {
  console.log('synchronously executed');
  if (Math.random() > 0.5) {
    res('Success');
  } else {
    rej('Error');
  }
})


prom.then((val) => {
  console.log('asynchronously executed: ' + val);
}).catch((err) => {
  console.log('asynchronously executed: ' + err);
}).finally(() => {
  console.log('promise done executing');
});


console.log('last log');
```

▼ What is strict mode in JS?

It is useful for writing secure JS code. It prevents from bugs from happening and throws more exception.

▼ Difference between null and undefined?

- null is a special keyword that indicates an absence of value.
- undefined property indicates that a variable has not been assigned a value including null too.

▼ Difference between asynchronous and synchronous?

Synchronous are blocking and asynchronous are not, synchronous complete running the current code before executing the next code are executed while asynchronous continue on the next code without completing the current code.

▼ Var, let and const?

- var are function scoped,

Example -

```
function setWidth(){
    var width = 100;
    console.log(width);
```

```
}
width;
// Returns:
Uncaught ReferenceError: width is not defined
```

- Let and const are block-scoped.

Example -

```
let age = 100;
  if (age > 12){
    let dogYears = age * 7;
    console.log(`You are ${dogYears} dog years old!`);
  }

dogYears;

Uncaught ReferenceError: dogYears is not defined
```

▼ What is DOM?

It stands for Document Object Model. this can be used to access and change the document structure, style, and content.

When a web page is loaded, the browser creates the DOM, which stands for Document Object Model It defines the logical structure of documents and the way a document is accessed and manipulated.

The **DOM** is an 'application programming interface' which means that we use to interact with web pages by:

- Creating and building documents, navigate their structure, and add, modify, or delete elements and content
- Change, delete, or add using the Document Object Model, such as the background color, as well as individual nodes.

Structure of the DOM

In the Document Object Model, documents have a logical structure which is very much like a tree:

- Document: which is the HTML web page
- Objects: Every HTML elements in the document is an object:<head></head> *head tag*<body></body> *body tag* *unlisted tag* *paragraph tag*
- The model describes how each elements is laid out into a tree diagram, from top to bottom.
- (Nodes): Everything we can change in the document is a node:ElementsText within ElementsHTML attributes

▼ Different Data types in JS?

Number, string , Boolean , object , undefined , null

▼ Primitive Data types?

- primitive data type is data that is not an object and has no methods and is immutable.
- There are 7 primitive data types: string, number, bigint, boolean, undefined, symbol, and null. ... All primitives are immutable, i.e., they cannot be altered.

- All primitive data type interact by value

▼ What is Hoisting?

In the creation phase of the code, memory space is set up for variables and functions. This is called hoisting. Hoisting is when the JavaScript interpreter moves all variable and function declarations to the top of the current scope. It's important to keep in mind that only the actual declarations are hoisted, and that assignments are left where they are.

```
console.log(addtoSum);
var addtoSum = 12;

//Result
addtoSum is undefined

//what happens above?
- During the creation phase, all of the variables get hoisted as undefined
and later on value is assigned to them when js engine reads line by line,
in this example addtoSum first automatically got hoisted as undefined and so in console it
will print as undefined and after console.log addtosum gets assigned to 12.

This is a very important topic for interview, please be very good with this.
If you need more indepth knowledge please check articles on medium.
```

▼ What is scope chain?

When a variable is used in JavaScript, the JavaScript engine will try to find the variable's value in the current scope. If it could not find the variable, it will look into the outer scope and will continue to do so until it finds the variable or reaches global scope.

```
let mytestScope = 20;

function outer(){
  funtion inner(){
    console.log(mytestScope);
  }
}

//This is an example of scopechain

- Here we try to access mytestScope within inner(). So, what happens in the background is that it tries to find
the variable with in its scope, if it is not there it will look for the variable in the parent scope, and if it
doesnt even find there it will move to the global scope and there it will find the variable.

This is how scope chain works when you are trying to access any variable.
```

▼ Difference between arrow function and normal function?

- Normal function depends on how the this keyword is called or on which object is it called.
- Arrow function only depends on where the function is called.
- We can make an new instance of a function(constructor) using new keyword for normal functions.
- With arrow function there is no constructer involved.

▼ Difference between Object.freeze() and Object.seal()?

- Object.freeze() - Read only
- Object.seal() - Read and Update only

▼ Deep Copy Vs Shallow Copy?

Primitive data type are deep copy by default

Non primitive have shallow copy and Deep copy

- Ways to copy an object
- Assignment operator (=) // shallow copy
- JSON.parse(JSON.stringify(obj)). // only work for object and not for functions inside object // deep copy
- Object.assign({}, obj). // partial deep copy but cannot deep copy nested object
- Spread operator - {…obj} // partial deep copy .. can’t copy nested obj

————Total deep copy using spread ————————————————

```
Let copiedValue = {…originalValue}

copiedValue = {

    …copiedValue,

    name: “Alisha”,

    address: {

    …copiedValue.address,

    city: “Goa”

  }

}
```

▼ What is Async/Await?

- The newest way to write asynchronous code in JavaScript.
- It is non blocking (just like promises and callbacks).
- Async/Await was created to simplify the process of working with and writing chained promises.
- Async functions return a Promise. If the function throws an error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

```
(async function() {
  const result = await asyncOperation(params);
  // Called when the operation completes
})();

const sendGetRequest = async () => {
    try {
        const resp = await axios.get('your url', {
            headers: {
                'authorization': 'Bearer YOUR_JWT_TOKEN_HERE'
            }
        });

        console.log(resp.data);
    } catch (err) {
        // Handle Error Here
```

```
        console.error(err);
    }
};
```

▼ ES6 Features

1. Let and const
2. Arrow functions
3. Template Literals
4. Default parameters
5. Object / Array destructure
6. Spread

▼ Explain event bubbling and how one may prevent it

Event bubbling is the concept in which an event triggers at the deepest possible element, and triggers on parent elements in nesting order. As a result, when clicking on a child element one may exhibit the handler of the parent activating.

One way to prevent event bubbling is using `event.stopPropagation()` or `event.cancelBubble` on IE < 9.

▼ What is Coercion

JavaScript is a **dynamically typed language**: we don't specify what types certain variables are. Values can automatically be converted into another type without you knowing, which is called *implicit type coercion*. **Coercion** is converting from one type into another.

ex- sum(1, “2”) → 12

▼ Implement a general Debouncing function

```
const debounce = (func, time) => {
    let timer;
    return () => {
      clearTimeout(timer);
     timer = setTimeout(() => {
        func();
      }, time);
    };
  };

  const debounceTest = () => {
    console.log("testing debounce");
  };

  console.log(debounce(debounceTest, 1000));
```

▼ Implement a general Memoization function

```
const Memo = (func) => {
    let cache = {}
    return (...args)=>{
      const value = JSON.stringify(args);
      if(value in cache){
        console.log("From Cache")
```

```
        return cache[value]
      } else {
        //caching it first
        console.log("without Cache")
        cache[value] = func.apply(this, args)
        return cache[value]
      }
    }
  }

const memoized = Memo(function here)
memoized(function arguments)
```

▼ Implement your own Array.prototype.flat()

So here we will implement out own flat() and call it a myflat()

```
const arr = [[1,3,4,6],[4,5,2,8],[7,3,8,3,9]];

Array.prototype.myflat = () => {
  let newArr = [];
  for( let i = 0 ; i < arr.length; i++){
    for (let j = 0 ; j < arr[i].length ; j++){
      newArr.push(arr[i][j])
    }
  }
  return newArr
}

console.log(arr.myflat());
```

▼ Implement your own throttle

```
const throttle = (func, limit) => {
  let inThrottle;
  return function() {
    const args = arguments;
    const context = this;
    if (!inThrottle) {
      func.apply(context, args);
      inThrottle = true;
      setTimeout(() => inThrottle = false, limit);
    }
  }
}

//func -> It will be any function that you want to throttle
//limit -> It is the time gap between each throttle
```

▼ **What is the drawback of declaring methods directly in JavaScript objects?**

One of the drawbacks of declaring methods directly in JavaScript objects is that they are very memory inefficient. When you do that, a new copy of the method is created for each instance of an object. Here's an example:

```
var Employee = function (name, company, salary) {
  this.name = name || "";
  this.company = company || "";
  this.salary = salary || 5000;

  // We can create a method like this:
```

```
  this.formatSalary = function () {
      return "$ " + this.salary;
  };
};

// Alternatively we can add the method to Employee's prototype:
Employee.prototype.formatSalary2 = function() {
    return "$ " + this.salary;
}

//creating objects
var emp1 = new Employee('Yuri Garagin', 'Company 1', 1000000);
var emp2 = new Employee('Dinesh Gupta', 'Company 2', 1039999);
var emp3 = new Employee('Erich Fromm', 'Company 3', 1299483);
```

In this case each instance variable `emp1`
, `emp2`
, `emp3`
has its own copy of the `formatSalary`
method. However the `formatSalary2`
will only be added once to `Employee.prototype`
.

▼ **What are Service Workers and when can you use them?**

It's a technology that allows your web application to use cached resources first, and provide default experience offline, before getting more data from the network later. This principle is commonly known as Offline First.

Service Workers actively use promises. A Service Worker has to be installed,activated and then it can react on fetch, push and sync events.

▼ **What is the difference between a method and a function in javascript?**

In JS, that difference is quite subtle. A function is a piece of code that is called by name and function itself not associated with any object and not defined inside any object. It can be passed data to operate on (i.e. parameter) and can optionally return data (the return value).

```
// Function statement
function myFunc() {
  // Do some stuff;
}

// Calling the function
myFunc();


// Methods
var obj1 = {
  attribute: "xyz",
  myMethod: function () {  // Method
    console.log(this.attribute);
  }
};

// Call the method
obj1.myMethod();
```

▼ **What are the ways of creating objects in JavaScript ?**

Method 1: Function based

```
function Employee(fName, lName, age, salary){
    this.firstName = fName;
    this.lastName = lName;
    this.age = age;
    this.salary = salary;
  }

  // Creating multiple object which have similar property but diff value assigned to object property.
  var employee1 = new Employee('John', 'Moto', 24, '5000$');
  var employee2 = new Employee('Ryan', 'Jor', 26, '3000$');
  var employee3 = new Employee('Andre', 'Salt', 26, '4000$');
```

Method 2: Object Literal

```
var employee = {
  name : 'Nishant',
  salary : 245678,
  getName : function(){
    return this.name;
  }
}
```

Method 3: From `Object` using `new` keyword

```
var employee = new Object(); // Created employee object using new keywords and Object()
employee.name = 'Nishant';
employee.getName = function(){
  return this.name;
}
```

Method 4:** Using `Object.create`

`Object.create(obj)` will create a new object and set the `obj` as its prototype. It's a modern way to create objects that inherit properties from other objects. `Object.create` function doesn't run the constructor. You can use `Object.create(null)` when you don't want your object to inherit the properties of `Object`

.

▼ Write a function called deepClone which takes an object and creates a object copy of it.

```
function deepClone(object){
  var newObject = {};
  for(var key in object){
    if(typeof object[key] === 'object'  && object[key] !== null ){
     newObject[key] = deepClone(object[key]);
    }else{
     newObject[key] = object[key];
    }
  }
  return newObject;
}


//sample object
var personalDetail = {
  name : 'Nishant',
  address : {
    location: 'xyz',
    zip : '123456',
    phoneNumber : {
      homePhone: 8797912345,
```

```
      workPhone : 1234509876
    }
  }
}

//calling deepClone function
deepClone(personalDetail)
```

▼ **How to use constructor functions for inheritance in JavaScript?**

Let say we have `Person` class which has name, age, salary properties and **incrementSalary()** method.

```
function Person(name, age, salary) {
  this.name = name;
  this.age = age;
  this.salary = salary;
  this.incrementSalary = function (byValue) {
    this.salary = this.salary + byValue;
  };
}
```

Now we wish to create Employee class which contains all the properties of Person class and wanted to add some additional properties into Employee class.

```
function Employee(company){
  this.company = company;
}

//Prototypal Inheritance
Employee.prototype = new Person("Sandip", 24,5000);
```

In the example above, **Employee** type inherits from **Person**. It does so by assigning a new instance of `Person` to `Employee` prototype. After that, every instance of `Employee` inherits its properties and methods from `Person.`

```
//Prototypal Inheritance
Employee.prototype = new Person("Sandip", 24,5000);

var emp1 = new Employee("PubicisSapient");

console.log(emp1 instanceof Person); // true
console.log(emp1 instanceof Employee); // true
```

Constructor inheritance

```
//Defined Person class
function Person(name){
  this.name = name || "Sandip";
}

var obj = {};

// obj inherit Person class properties and method
Person.call(obj); // constructor inheritance

console.log(obj); // Object {name: "Sandip"}
```

Here we saw calling **Person.call(obj)** define the name properties from `Person` to `obj`

```
console.log(name in obj); // true
```

▼ **Write code for merge two JavaScript Object dynamically.**

Let say you have two objects

```
var person = {
  name : 'Sandip',
  age  : 26
}

var address = {
  addressLine1 : 'Some Location x',
  addressLine2 : 'Some Location y',
  city : 'Kolkata'
}
```

Write merge function which will take two object and add all the own property of second object into first object.

```
function merge(toObj, fromObj) {
  // Make sure both of the parameter is an object
  if (typeof toObj === 'object' && typeof fromObj === 'object') {
    for (var pro in fromObj) {
      // Assign only own properties not inherited properties
      if (fromObj.hasOwnProperty(pro)) {
        // Assign property and value
        toObj[pro] = fromObj[pro];
      }
    }
  }else{
    throw "Merge function can apply only on object";
  }
}

//calling function
merge(person, address)
```

▼ **What is generator in JS?**

Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances. Generator functions are written using the `function*` syntax. When called initially, generator functions do not execute any of their code, instead returning a type of iterator called a Generator. When a value is consumed by calling the generator's `next` method, the Generator function executes until it encounters the `yield` keyword.

The function can be called as many times as desired and returns a new Generator each time, however each Generator may only be iterated once.

```
function* makeRangeIterator(start = 0, end = Infinity, step = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```

▼ **When should we use generators in ES6?**

To put it simple, generator has two features:

- one can choose to jump out of a function and let outer code to determine when to jump back into the function.
- the control of asynchronous call can be done outside of your code

The most important feature in generators—we can get the next value in only when we really need it, not all the values at once. And in some situations it can be very convenient.

▼ **Difference between .map() and .forEach()? which one to choose over other?**

`forEach`

- Iterates through the elements in an array.
- Executes a callback for each element.
- Does not return a value.

```
const a = [1, 2, 3];
const doubled = a.forEach((num, index) => {
  // Do something with num and/or index.
});

// doubled = undefined
```

`map`

- Iterates through the elements in an array.
- "Maps" each element to a new element by calling the function on each element, creating a new array as a result.

```
const a = [1, 2, 3];
const doubled = a.map(num => {
  return num * 2;
});

// doubled = [2, 4, 6]
```

The main difference between `.forEach` and `.map()` is that `.map()` returns a new array. If you need the result, but do not wish to mutate the original array, `.map()` is the clear choice. If you simply need to iterate over an array, `forEach` is a fine choice.

▼ **What is the Temporal Dead Zone in ES6?**

In ES6 `let` and `const` are hoisted (like `var`, `class` and `function`), but there is a period between entering scope and being declared where they cannot be accessed. **This period is the temporal dead zone (TDZ)**

```
//console.log(aLet)  // would throw ReferenceError

let aLet;
console.log(aLet); // undefined
aLet = 10;
console.log(aLet); // 10
```