Introduction to JavaScript for...in loop

The `for...in` loop over the enumerable properties that are keyed by strings of an object. Note that a property can be keyed by a string or a symbol.

```
var person = {
    firstName: 'John',
    lastName: 'Doe',
    ssn: '299-24-2351'
};


for(var prop in person) {
    console.log(prop + ':' + person[prop]);
}
```

```
Output:
firstName:John
lastName:Doe
ssn:299-24-2351
```

The for...in loop & Inheritance

When you loop over the properties of an object that inherits from another object, the `for...in` statement goes up in the prototype chain and enumerates over inherited properties. Consider the following example:

```
var decoration = {
    color: 'red'
};

var circle = Object.create(decoration);
circle.radius = 10;


for(const prop in circle) {
    console.log(prop);
}
```

```
Output:
radius
color
```

The circle object has its own prototype that references the decoration object. Therefore, the for...in loop displays the properties of the circle object and its prototype.

```
for(const prop in circle) {
    if(circle.hasOwnProperty(prop)) {
        console.log(prop);
    }
}
```

Output:

```
radius
```

The for...in loop and Array

It’s good practice to not use the for...in to iterate over an array, especially when the order of the array elements is important.

```
const items = [10 , 20, 30];
let total = 0;

for(const item in items) {
    total += items[item];
}
console.log(total);
```

Hence, the for...in will not work correctly. For example:

```
// somewhere else
Array.prototype.foo = 100;

const items = [10, 20, 30];
let total = 0;

for (var prop in items) {
  console.log({ prop, value: items[prop] });
  total += items[prop];
}
console.log(total);
```

Output:

```
{ prop: '0', value: 10 }
{ prop: '1', value: 20 }
{ prop: '2', value: 30 }
{ prop: 'foo', value: 100 }
160
```

or another example

```
var arr = [];
// set the third element to 3, other elements are `undefined`
arr[2] = 3;

for (let i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

The output shows three elements of the array, which is correct:

```
for (const key in arr) {
    console.log(arr[key]);
}
```

Output:

```
3
```

Introduction to JavaScript enumerable properties

Enumerable properties are iterated using the for...in loop or Objects.keys() method.

In JavaScript, an object is an unordered list of key-value pairs. The key is usually a string or a symbol. The value can be a value of any primitive type (string, boolean, number, undefined, or null), an object, or a function.

The following example creates a new object using the object literal syntax:

```
const person = {
    firstName: 'John',
    lastName: 'Doe
};
```

The person object has two properties: firstName and lastName.

An object property has several internal attributes including value, writable, enumerable and configurable. See the Object properties for more details.

The enumerable attribute determines whether or not a property is accessible when the object’s properties are enumerated using the for...in loop or Object.keys() method.

By default, all properties created via a simple assignment or via a property initializer are enumerable. For example:

```
const person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.age = 25;

for (const key in person) {
    console.log(key);
console.log(person[key]);
}
```

```
Output:
firstName
lastName
age
```

In this example:

- The `firstName` and `lastName` are enumerable properties because they are created via a property initializer.
- The `age` property is also enumerable because it is created via a simple assignment.

To change the internal `enumerable` attribute of a property, you use the `Object.defineProperty()` method. For example:

```
const person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.age = 25;

Object.defineProperty(person, 'ssn', {
    enumerable: false,
    value: '123-456-7890'
});

for (const key in person) {
    console.log(key);
}
Output:
firstName
lastName
age
```

In this example, the ssn property is created with the enumerable flag sets to false, therefore it does not show up in the for...in loop.

ES6 provides a method propertyIsEnumerable() that determines whether or not a property is enumerable. It returns true if the property is enumerable; otherwise false. For example:

```
const person = {
    firstName: 'John',
    lastName: 'Doe'
};

person.age = 25;

Object.defineProperty(person, 'ssn', {
    enumerable: false,
    value: '123-456-7890'
});


console.log(person.propertyIsEnumerable('firstName')); // => true
console.log(person.propertyIsEnumerable('lastName')); // => true
console.log(person.propertyIsEnumerable('age')); // => true
console.log(person.propertyIsEnumerable('ssn')); // => false
```

JavaScript for...of loop

The syntax of the for...of loop is:

```
for (element of iterable) {
    // body of for...of
}
```

The for...of loop was introduced in the later versions of **JavaScript ES6**.

The for..of loop in JavaScript allows you to iterate over iterable objects (arrays, sets, maps, strings etc).

Here,

- **iterable** - an iterable object (array, set, strings, etc).
- **element** - items in the iterable

for...of with Arrays

The for..of loop can be used to iterate over an array. For example,

```
// array
const students = ['John', 'Sara', 'Jack'];

// using for...of
for ( let element of students ) {

    // display the values
    console.log(element);
}
```

Output

```
John
Sara
Jack
```

for...of with Strings

You can use for...of loop to iterate over string values. For example,

```
// string
const string = 'code';

// using for...of loop
for (let i of string) {
    console.log(i);
}
```

for...of with Sets

You can iterate through Set elements using the for...of loop. For example,

```
// define Set
const set = new Set([1, 2, 3]);

// looping through Set
for (let i of set) {
    console.log(i);
}
```

```
Output:
1
2
3
```

for...of with Maps

You can iterate through Map elements using the for...of loop. For example,

```
// define Map
let map = new Map();

// inserting elements
map.set('name', 'Jack');
map.set('age', '27');

// looping through Map
for (let [key, value] of map) {
    console.log(key + '- ' + value);
}
```

```
Output:
name - Jack
age - 27
```

User Defined Iterators

You can create an iterator manually and use the for...of loop to iterate through the iterators. For example,

```
// creating iterable object
```

```
const iterableObj = {

    // iterator method
    [Symbol.iterator]() {
        let step = 0;
        return {
            next() {
                step++;
                if (step === 1) {
                    return { value: '1', done: false};
                 }
                else if (step === 2) {
                    return { value: '2', done: false};
                }
                else if (step === 3) {
                    return { value: '3', done: false};
                }
                return { value: '', done: true };
            }
        }
    }
}

// iterating using for...of
for (const i of iterableObj) {
 console.log(i);
}
```

Output:

```
1
2
3
```

for...of Vs for…in

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

JavaScript static methods

By definition, static methods are bound to a class, not the instances of that class. Therefore, static methods are useful for defining helper or utility methods.

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
};
```

The following adds a static method called createAnonymous() to the Person type:

```
Person.createAnonymous = function (gender) {
    let name = gender == "male" ? "John Doe" : "Jane Doe";
    return new Person(name);
};
```

The createAnonymous() method is considered a static method because it doesn't depend on any instance of the Person type for its property values.

```
var anonymous = Person.createAnonymous();
```

JavaScript static methods in ES6

In ES6, you define static methods using the `static` keyword. The following example defines a static method called `createAnonymous()` for the `Person` class:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
    static createAnonymous(gender) {
        let name = gender == "male" ? "John Doe" : "Jane Doe";
        return new Person(name);
    }
}
```

To invoke the static method, you use the following syntax:

```
let anonymous = Person.createAnonymous(“male");
```

If you attempt to call the static method from an instance of the class, you’ll get an error. For example:

```
let person = new Person('James Doe');
let anonymous = person.createAnonymous(“male");
```

Error:

```
TypeError: person.createAnonymous is not a function
```

Calling a static method from the class constructor or an instance method

To call a static method from a class constructor or an instance method, you use the class name, followed by the `.` and the static method:

```
className.staticMethodName();
```

Alternatively, you can use the following syntax:

```
this.constructor.staticMethodName();
```

Introduction to the JavaScript static properties

Like a static method, a static property is shared by all instances of a class. To define static property, you use the `static` keyword followed by the property name like this:

```
class Item {
  static count = 0;
}
```

To access a static property, you use the class name followed by the `.` operator and the static property name. For example:

```
console.log(Item.count); // 0
```

To access the static property in a static method, you use the class name followed by the `.` operator and the static property name. For example:

```
class Item {
  static count = 0;
  static getCount() {
    return Item.count;
  }
}

console.log(Item.getCount()); // 0
```

To access a static property in a class constructor or instance method, you use the following syntax:

```
className.staticPropertyName;
```

or

```
this.constructor.staticPropertyName;
```

The following example increases the `count` static property in the class constructor:

```
class Item {
  constructor(name, quantity) {
    this.name = name;
    this.quantity = quantity;
    this.constructor.count++;
  }
  static count = 0;
  static getCount() {
    return Item.count;
  }
}
```

When you create a new instance of the `Item` class, the following statement increases the `count` static property by one:

```
this.constructor.count++;
```

Introduction to the JavaScript call() method

The `Function.prototype` type has the `call()` method with the following syntax:

```
functionName.call(thisArg, arg1, arg2, ...);
```

In this syntax, the `call()` method calls a function `functionName` with the arguments (`arg1`, `arg2`, …) and the `this` set to `thisArg` object inside the function.

- The `thisArg` is the object that the `this` object references inside the function `functionName`.
- The `arg1`, `arg2`, .. are the function arguments passed into the `functionName`.

The `call()` method returns the result of calling the `functionName()`.

The following example defines the `add()` function and calls it normally:

```
function add(x, y) {
  return x + y;
}

let result = add(10, 20);
console.log(result); // 30
```

The following calls the `add()` function but use the `call()` method instead:

```
function add(x, y) {
  return x + y;
}

let result = add.call(this, 10, 20);
console.log(result); // 30
```

Consider the following example:

```
var greeting = 'Hi';

var messenger = {
    greeting: 'Hello'
}

function say(name) {
    console.log(this.greeting + ' ' + name);
}
```

Inside the `say()` function, we reference the `greeting` via the `this` value. If you just invoke the `say()` function via the `call()` method as follows:

```
say.call(this,’John');
```

It’ll show the following output to the console:

```
"Hi John”
```

However, when you invoke the `call()` method of `say` function object and pass the `messenger` object as the `this` value:

```
say.call(messenger,’John');
```

The output will be:

```
"Hello John”
```

Using the JavaScript call() method to chain constructors for an object

You can use the `call()` method for chaining constructors of an object. Consider the following example:

```
function Box(height, width) {

  this.height = height;

  this.width = width;

}

function Widget(height, width, color) {

  Box.call(this, height, width);

  this.color = color;

}

let widget = new Widget('red', 100, 200);

console.log(widget);
```

Output:

```
Widget { height: 'red', width: 100, color: 200 }
```

In this example:

- First, initialize the `Box` object with two properties: `height` and `width`.
- Second, invoke the `call()` method of the `Box` object inside the `Widget` object, set the `this` value to the `Widget` object.

Using the JavaScript call() method for function borrowing

The following example illustrates how to use the call() method for borrowing functions:

```
const car = {
  name: 'car',
  start() {
    console.log('Start the ' + this.name);
  },
  speedUp() {
    console.log('Speed up the ' + this.name);
  },
  stop() {
    console.log('Stop the ' + this.name);
  },
};

const aircraft = {
  name: 'aircraft',
  fly() {
    console.log('Fly');
  },
};

car.start.call(aircraft);
car.speedUp.call(aircraft);
aircraft.fly();
```

Output:

```
Start the aircraft
Speed up the aircraft
Fly
```

How it works.

First, define a car object with one property name and three methods `start`, `speedUp`, and `stop`:

```
const car = {
  name: 'car',
  start() {
    console.log('Start the ' + this.name);
  },
  speedUp() {
    console.log('Speed up the ' + this.name);
  },
  stop() {
    console.log('Stop the ' + this.name);
  },
};
```

Second, define the aircraft object with one property name and a method:

```
const aircraft = {
  name: 'aircraft',
  fly() {
    console.log('Fly');
  },
};
```

Third, call the start() and speedUp() method of the car object and the fly() method of the aircraft object. However, passing the aircraft as the first argument into the start() and speedUp() methods:

```
car.start.call(aircraft);
car.speedUp.call(aircraft);
aircraft.fly();
```

Inside the start() and speedUp() methods, the this references the aircraft object, not the car object. Therefore, the this.name returns the 'aircraf' string. Hence, the methods output the following message:

```
Start the aircraft
Speed up the aircraft
```

Technically, the aircraft object borrows the start() and speedUp() method of the car object. And function borrowing refers to an object that uses a method of another object.

The following example illustrates how the arguments object borrows the filter() method of the Array.prototype via the call() function:

```
function isOdd(number) {
  return number % 2;
}

function getOddNumbers() {
  return Array.prototype.filter.call(arguments, isOdd);
}

let results = getOddNumbers(10, 1, 3, 4, 8, 9);
console.log(results);
```

Output:

```
[ 1, 3, 9 ]
```

How it works.

First, define the `isOdd()` function that returns true if the number is an odd number:

```
function isOdd(number) {
  return number % 2;
}
```

Second, define the `getOddNumbers()` function that accepts any number of arguments and returns an array that contains only odd numbers:

```
function getOddNumbers() {
  return Array.prototype.filter.call(arguments, isOdd);
}
```

In this example, the `arguments` object borrows the filter() method of the `Array.prototype` object.

Third, call the `getOddNumbers()` function:

```
let results = getOddNumbers(10, 1, 3, 4, 8, 9);
console.log(results);
```

What are callbacks

In JavaScript, functions are first-class citizens. Therefore, you can pass a function to another function as an argument.

By definition, a callback is a function that you pass into another function as an argument for executing later.

The following defines a filter() function that accepts an array of numbers and returns a new array of odd numbers:

Introduction to JavaScript bind() method

The bind() method returns a new function, when invoked, has its this sets to a specific value.

The following illustrates the syntax of the bind() method:

```
fn.bind(thisArg[, arg1[, arg2[, ...]]])
```

```
function filter(numbers) {
  let results = [];
  for (const number of numbers) {
    if (number % 2 != 0) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];
console.log(filter(numbers));
```

How it works.

- First, define the filter() function that accepts an array of numbers and returns a new array of the odd numbers.
- Second, define the numbers array that has both odd and even numbers.
- Third, call the filter() function to get the odd numbers out of the numbers array and output the result.

If you want to return an array that contains even numbers, you need to modify the filter() function. To make the filter() function more generic and reusable, you can:

- First, extract the logic in the if block and wrap it in a separate function.
- Second, pass the function to the filter() function as an argument.

Here's the updated code:

```
function isOdd(number) {
  return number % 2 != 0;
}

function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];
console.log(filter(numbers, isOdd));
```

The result is the same. However, you can pass any function that accepts an argument and returns a boolean value to the second argument of the filter() function.

For example, you can use the filter() function to return an array of even numbers like this:

```
function isOdd(number) {
  return number % 2 != 0;
}
function isEven(number) {
  return number % 2 == 0;
}
```

```
function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));
console.log(filter(numbers, isEven));
```

By definition, the `isOdd` and `isEven` are callback functions or callbacks. Because the `filter()` function accepts a function as an argument, it's called a *high-order function*.

A callback can be an anonymous function, which is a function without a name like this:

```
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, function (number) {
  return number % 2 != 0;
});

console.log(oddNumbers);
```

n this example, we pass an anonymous function to the filter() function instead of using a separate function.

In ES6, you can use an arrow function like this:

```
function filter(numbers, callback) {
  let results = [];
  for (const number of numbers) {
    if (callback(number)) {
      results.push(number);
    }
  }
  return results;
}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, (number) => number % 2 != 0);

console.log(oddNumbers);
```

JavaScript single-threaded model

JavaScript is a single-threaded programming language. This means that JavaScript can do only one thing at a single point in time.

The JavaScript engine executes a script from the top of the file and works its way down. It creates the execution contexts, pushes, and pops functions onto and off the call stack in the execution phase.

If a function takes a long time to execute, you cannot interact with the web browser during the function's execution because the page hangs.

A function that takes a long time to complete is called a blocking function. Technically, a blocking function blocks all the interactions on the webpage, such as mouse click.

An example of a blocking function is a function that calls an API from a remote server.

```
function task(message) {
    // emulate time consuming task
    let n = 10000000000;
    while (n > 0){
        n--;
    }
    console.log(message);
}

console.log('Start script...');
task('Download a file.');
console.log(‘Done!');
```

n this example, we have a big `while` loop inside the `task()` function that emulates a time-consuming task. The `task()` function is a blocking function.

The script hangs for a few seconds (depending on how fast the computer is) and issues the following output:

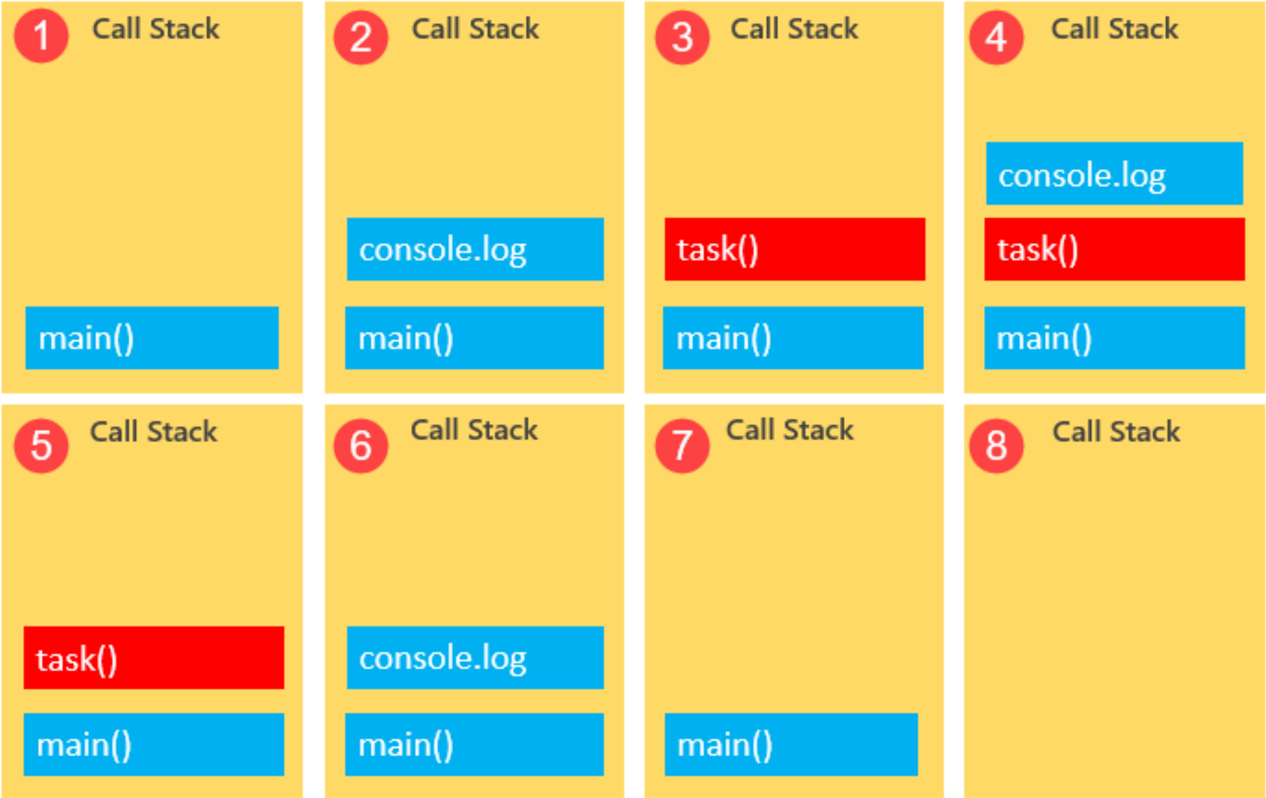
Output:

```
Start script...
Download a file.
Done!
```

To execute the script, the JavaScript engine places the first call `console.log()` on top of the call stack and executes it. Then, it places the `task()` function on top of the call stack and executes the function.

However, it’ll take a while to complete the `task()` function. Therefore, you’ll see the message `'Download a file.'` a little time later. After the `task()` function completes, the JavaScript engine pops it off the call stack.

Finally, the JavaScript engine places the last call to the `console.log('Done!')` function and executes it, which will be very fast.



Callbacks to the rescue

To prevent a blocking function from blocking other activities, you typically put it in a callback function for execution later. For example:

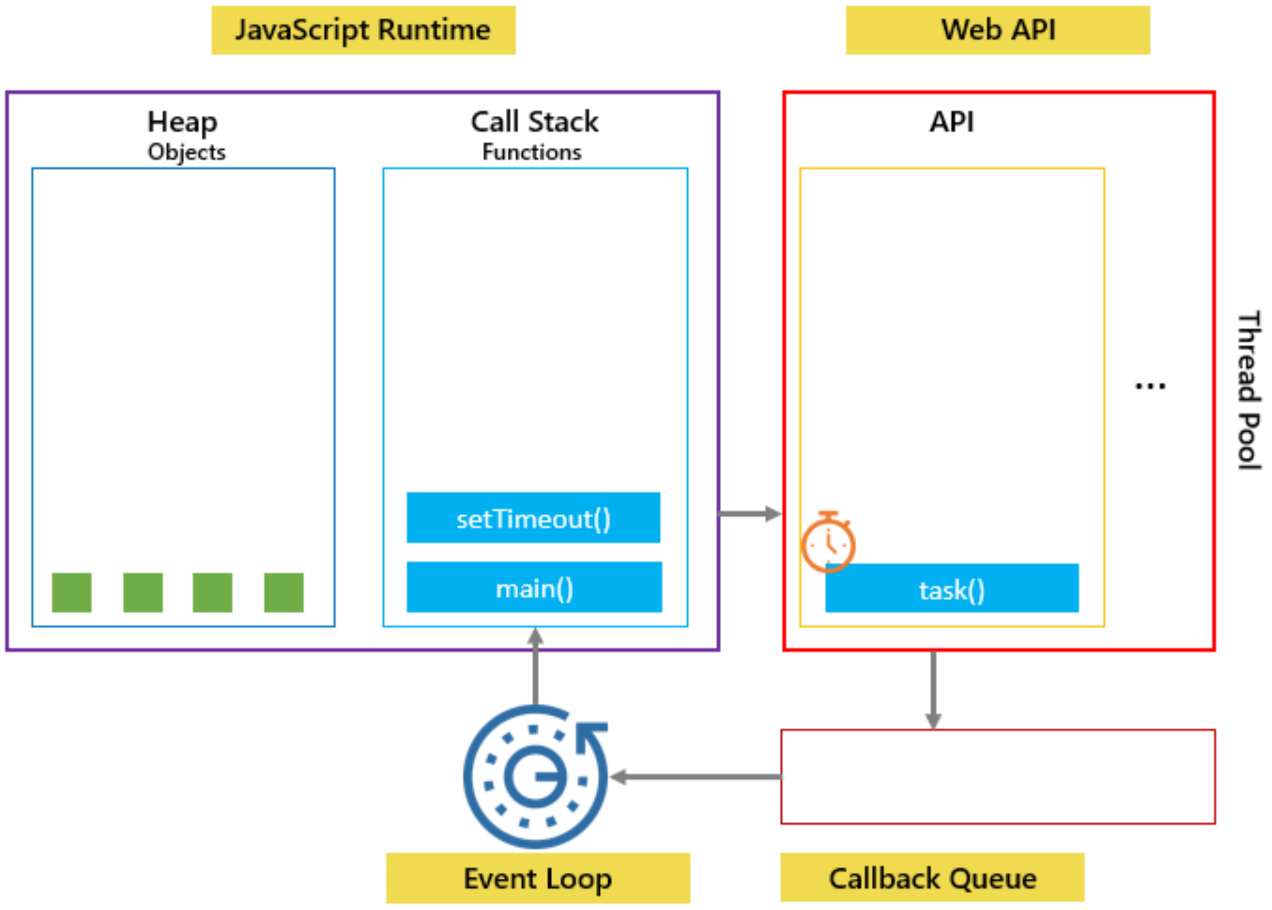
```
console.log('Start script...');

setTimeout(() => {
    task('Download a file.');
}, 1000);

console.log('Done!');
```

In this example, you'll see the message `'Start script...'` and `'Done!'` immediately. And after that, you'll see the message `'Download a file'`.

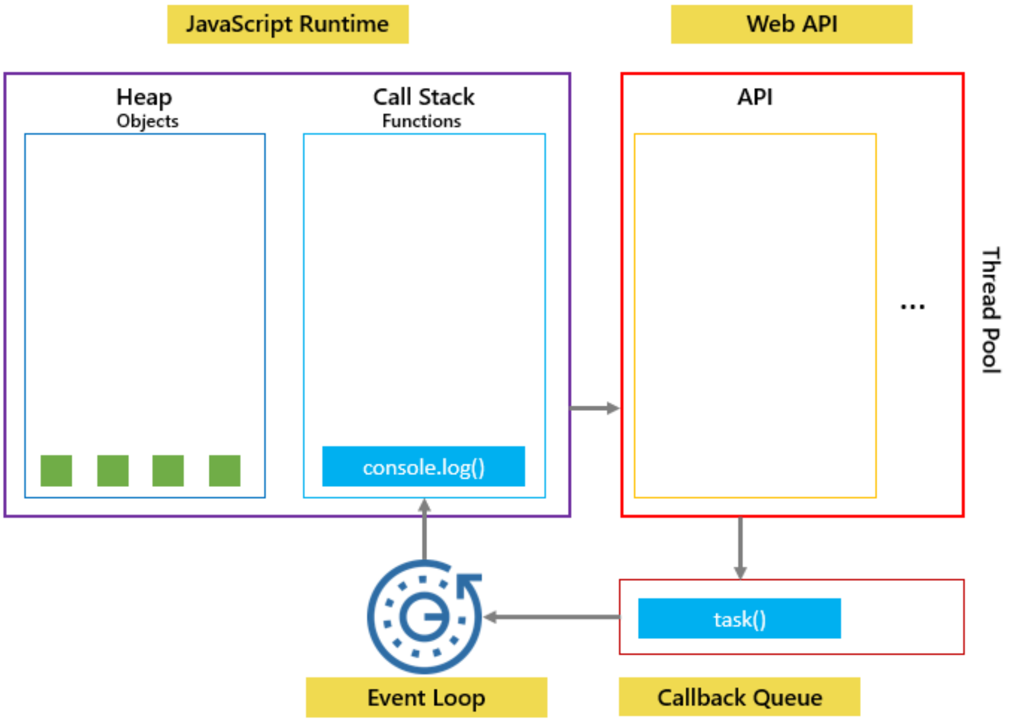
Output:

Start script...
Done!
Download a file.

In our example, when calling the setTimeout() function, the JavaScript engine places it on the call stack, and the Web API creates a timer that expires in 1 second.

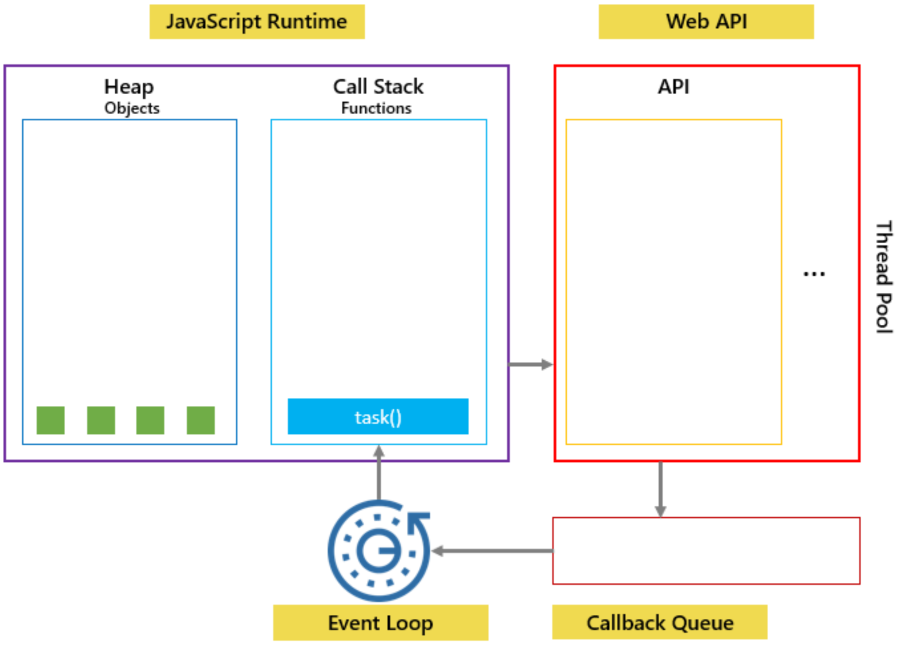


Then JavaScript engine place the task() function is into a queue called a callback queue or a task queue:



The event loop is a constantly running process that monitors both the callback queue and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callback queue is empty, nothing will happen



See another example:

```
console.log('Hi!');

setTimeout(() => {
   console.log('Execute immediately.');
}, 0);

console.log('Bye!');
```

In this example, the timeout is 0 second, so the message 'Execute immediately.' should appear before the message 'Bye!'. However, it doesn't work like that.

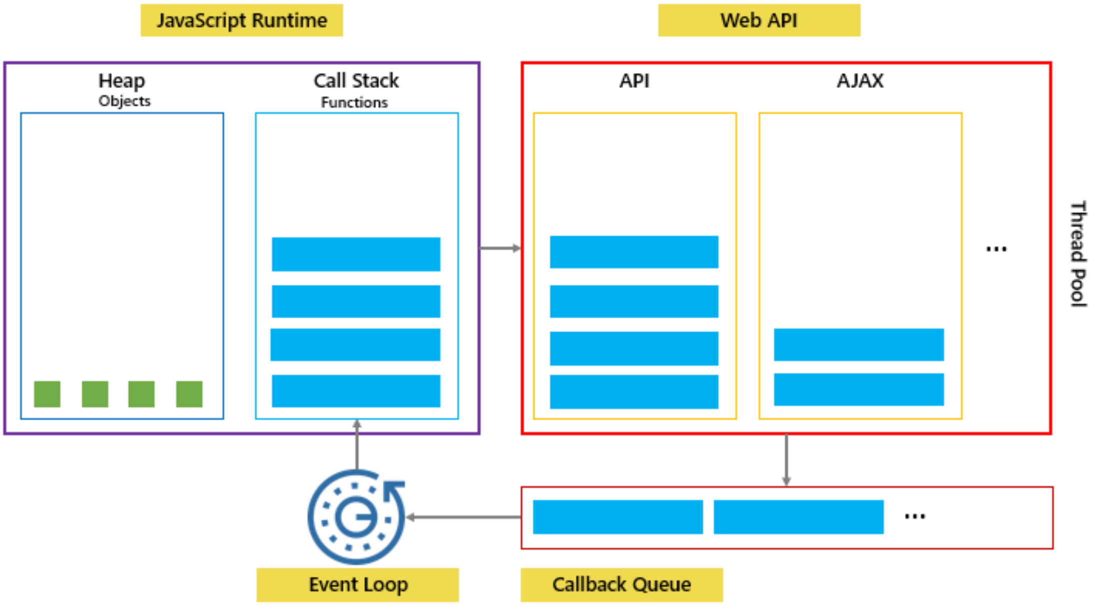
The JavaScript engine places the following function call on the callback queue and executes it when the call stack is empty. In other words, the JavaScript engine executes it after the console.log('Bye!').

```
console.log('Execute immediately.');
```

Here's the output:

```
Hi!
Bye!
Execute immediately.
```

The following picture illustrates JavaScript runtime, Web API, Call stack, and Event loop:


JavaScript Runtime
Web API
Heap
Objects
Call Stack
Functions
API
AJAX
...
Thread Pool
...
Event Loop
Callback Queue