NodeJS
środowisko i technologia ServerSide

PAWEŁ ŁUKASZUK

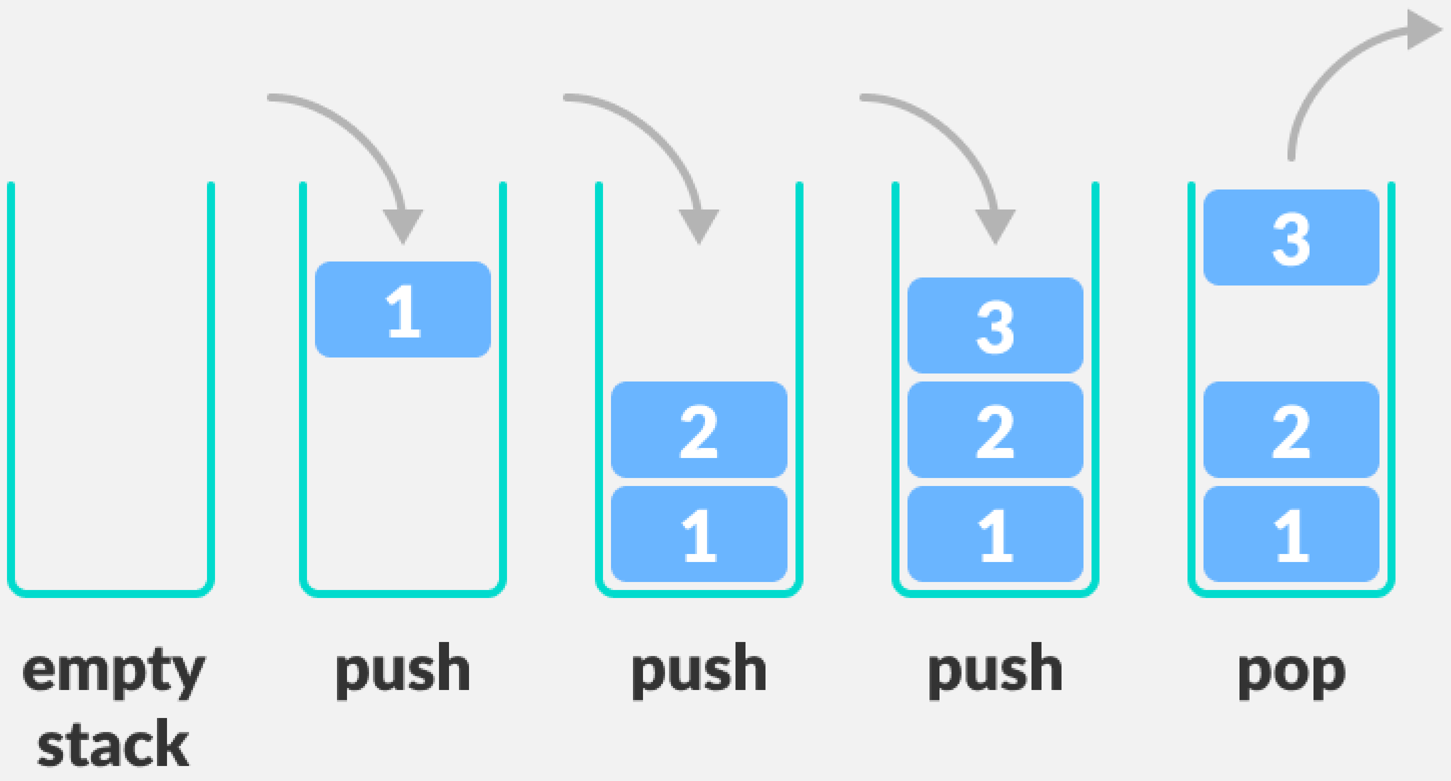
1064

Stack

In computer science, a stack is an abstract data structure that serves as a collection of elements, with two main principal operations:

- push, which adds an element to the collection
- pop, which removes the most recently added element that was not yet removed.

Stack



LIFO

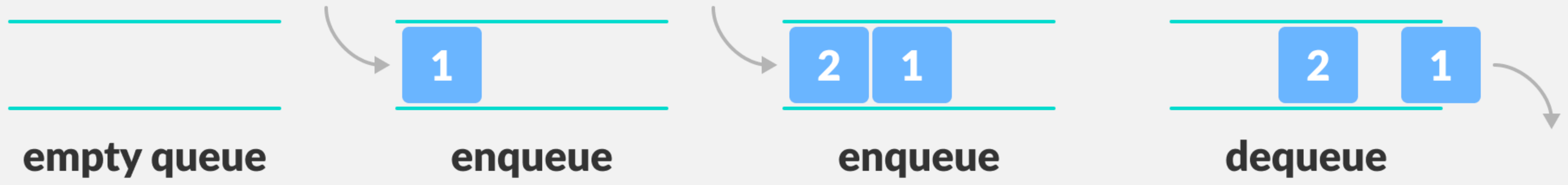
LAST IN – FIRST OUT

Queue

In computer science, a queue is an abstract data structure that serves as a collection of elements, with two main principal operations:

- enqueue, which adds an element to the end collection
- dequeue, which removes the last recently added element

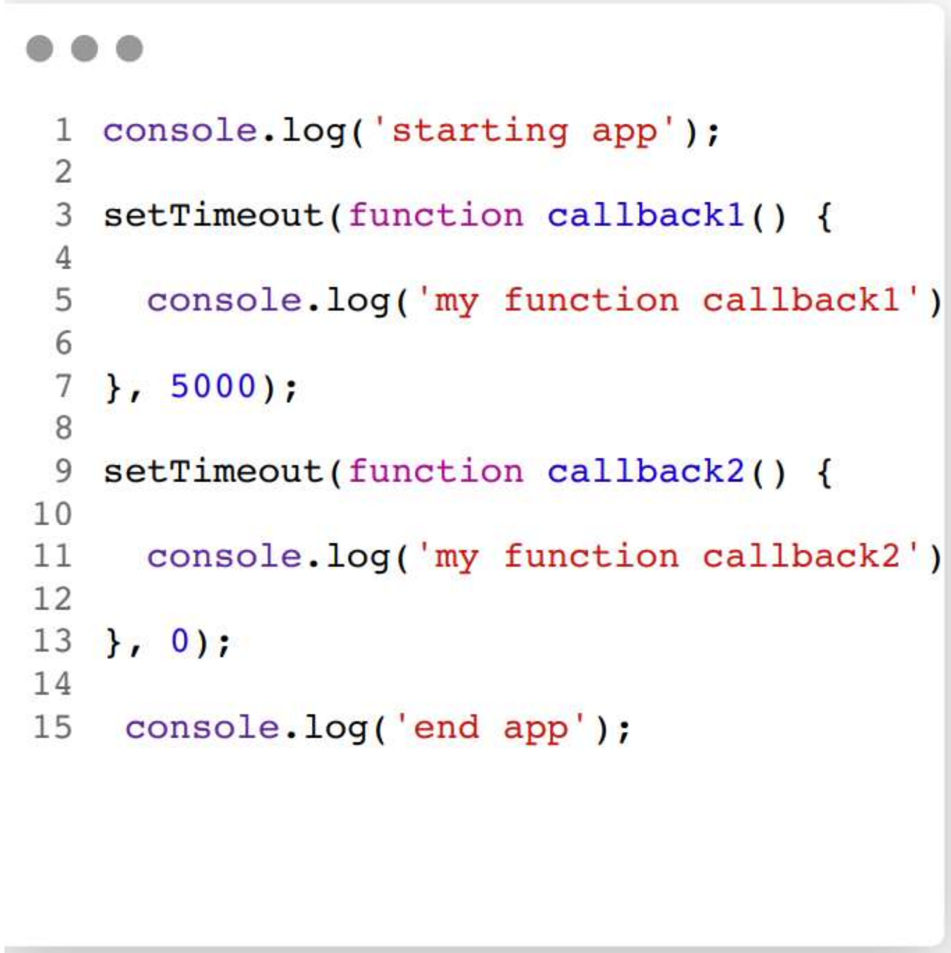
Queue



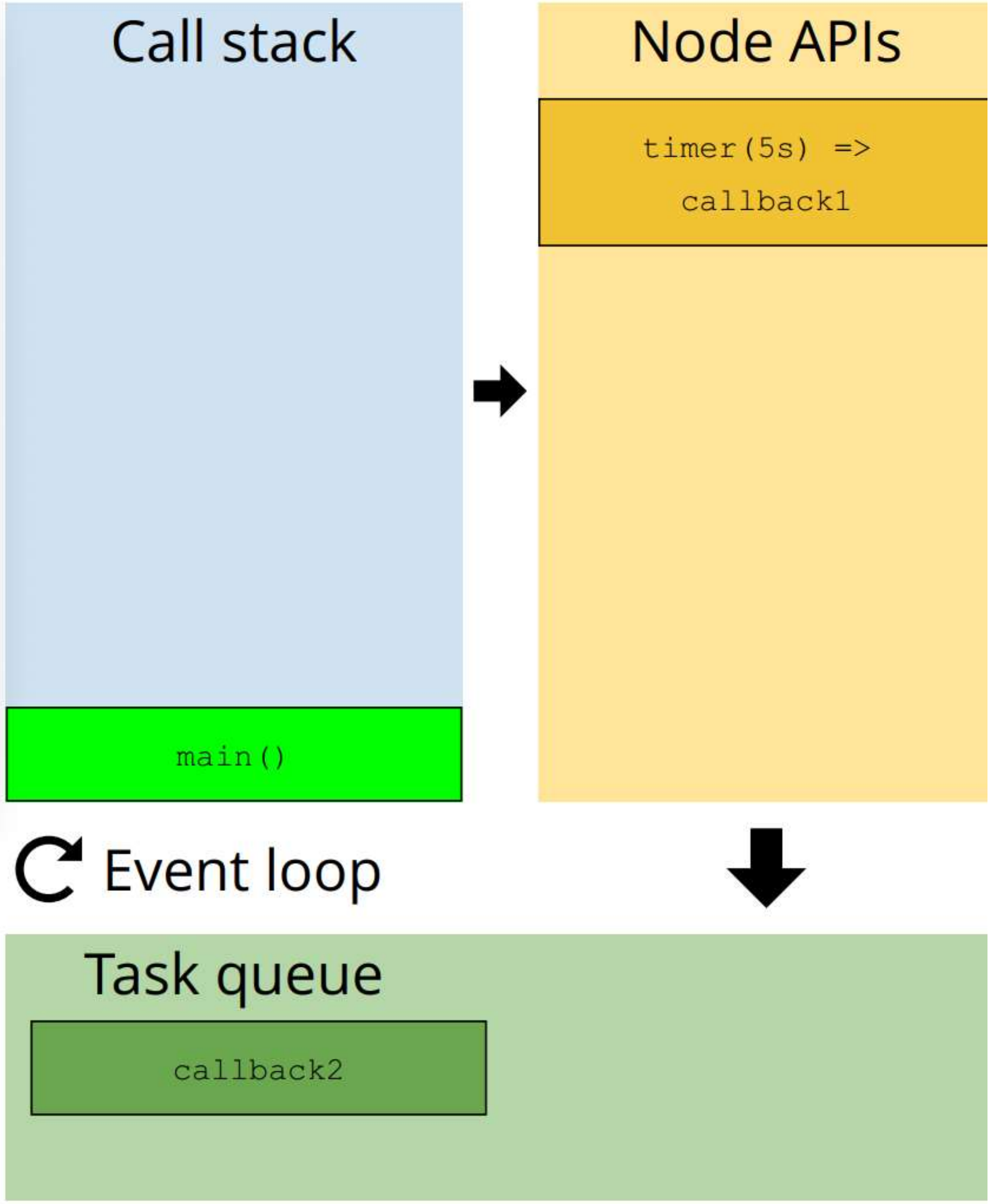
FIFO

FIRST IN – FIRST OUT

Async review



```
console.log('starting app');

setTimeout(function callback1() {

  console.log('my function callback1')

}, 5000);

setTimeout(function callback2() {

  console.log('my function callback2')

}, 0);

 console.log('end app');
```



Event loop

The event loop is what allows Node.js to perform non-blocking I/O operations by offloading operations to the system kernel whenever possible.

When Node.js starts, it initializes the event loop, processes the provided input script (or drops into the REPL) which may make async API calls, schedule timers then begins processing the event loop.

The loop gives priority to the call stack, and it first processes everything it finds in the call stack, and once there's nothing in there, it goes to pick up things in the task queue.

Call stack

A call stack is a mechanism for an interpreter to keep track of its place in a script that calls multiple functions — what function is currently being run and what functions are called from within that function, etc.

- when a script calls a function, the interpreter adds it to the call stack and then starts carrying out the function
- any functions that are called by that function are added to the call stack further up, and run where their calls are reached
- when the current function is finished, the interpreter takes it off the stack and resumes execution where it left off in the last code listing

Friendly reminder

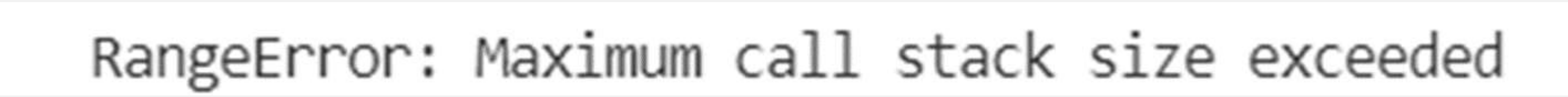
The Call stack is

not an infinite resource

Stack overflow

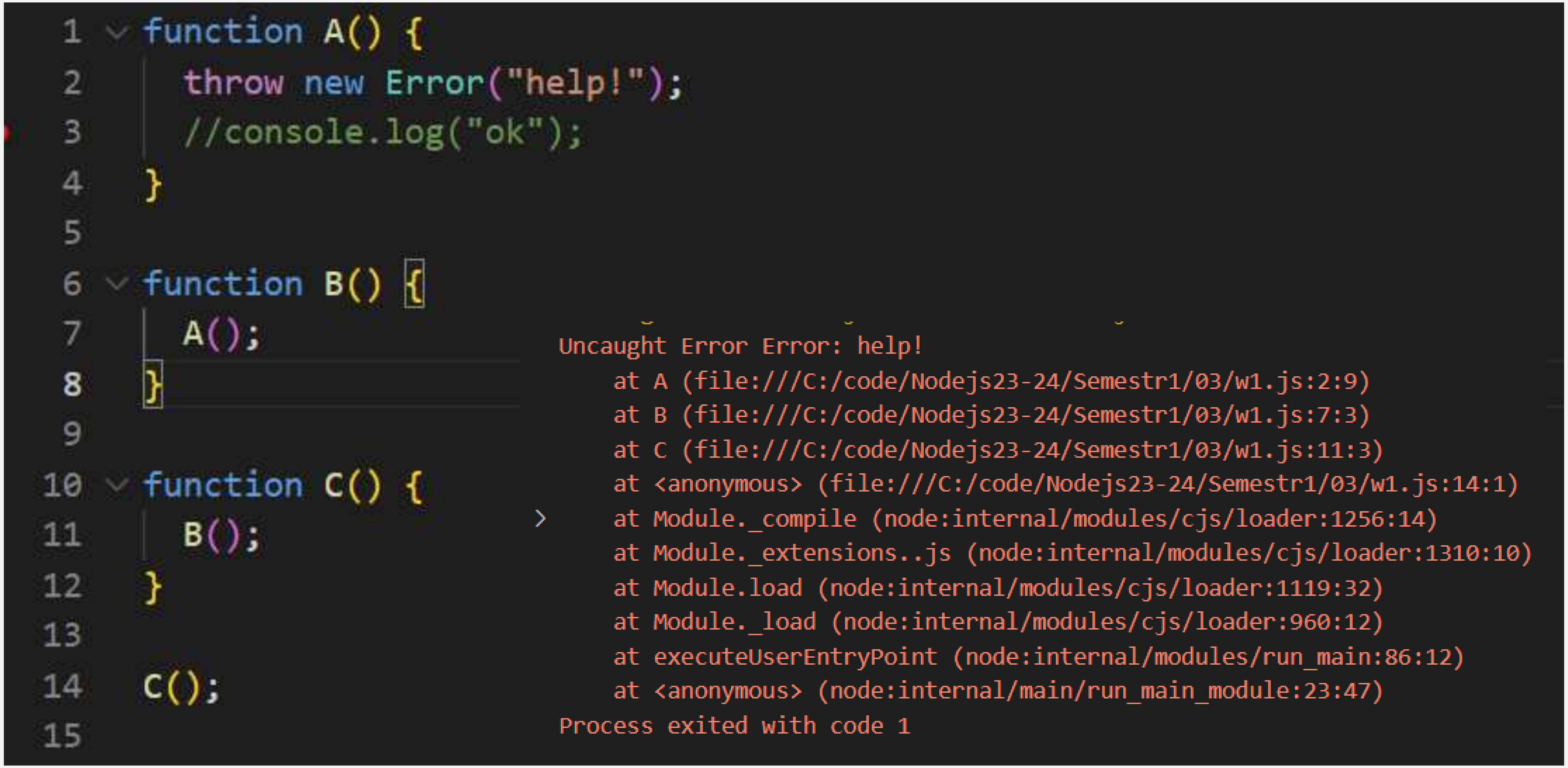
If the stack takes up more space than it had assigned to it, it results in a "stack overflow" error.

Stack overflow



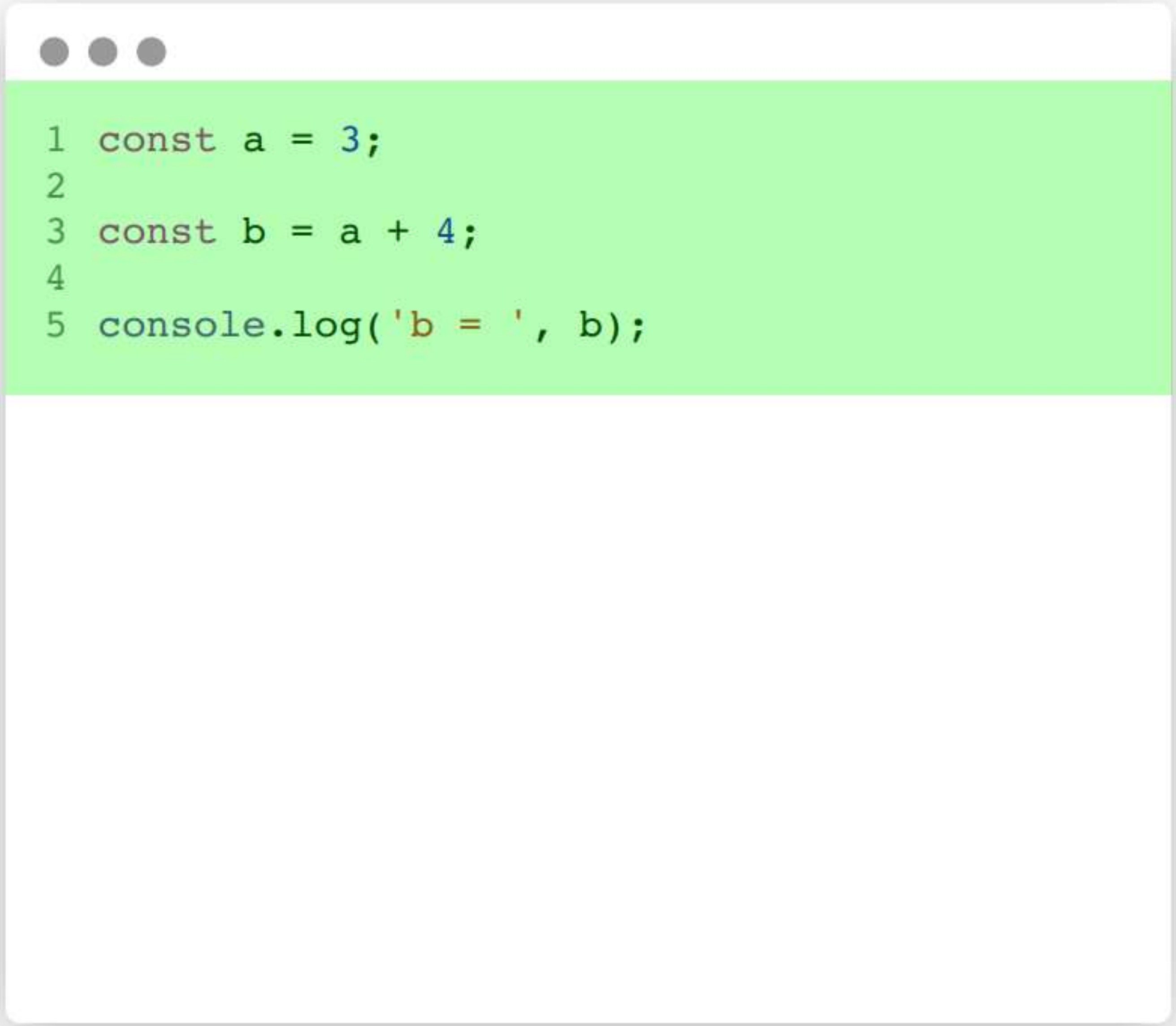
```
RangeError: Maximum call stack size exceeded
```

Stack trace

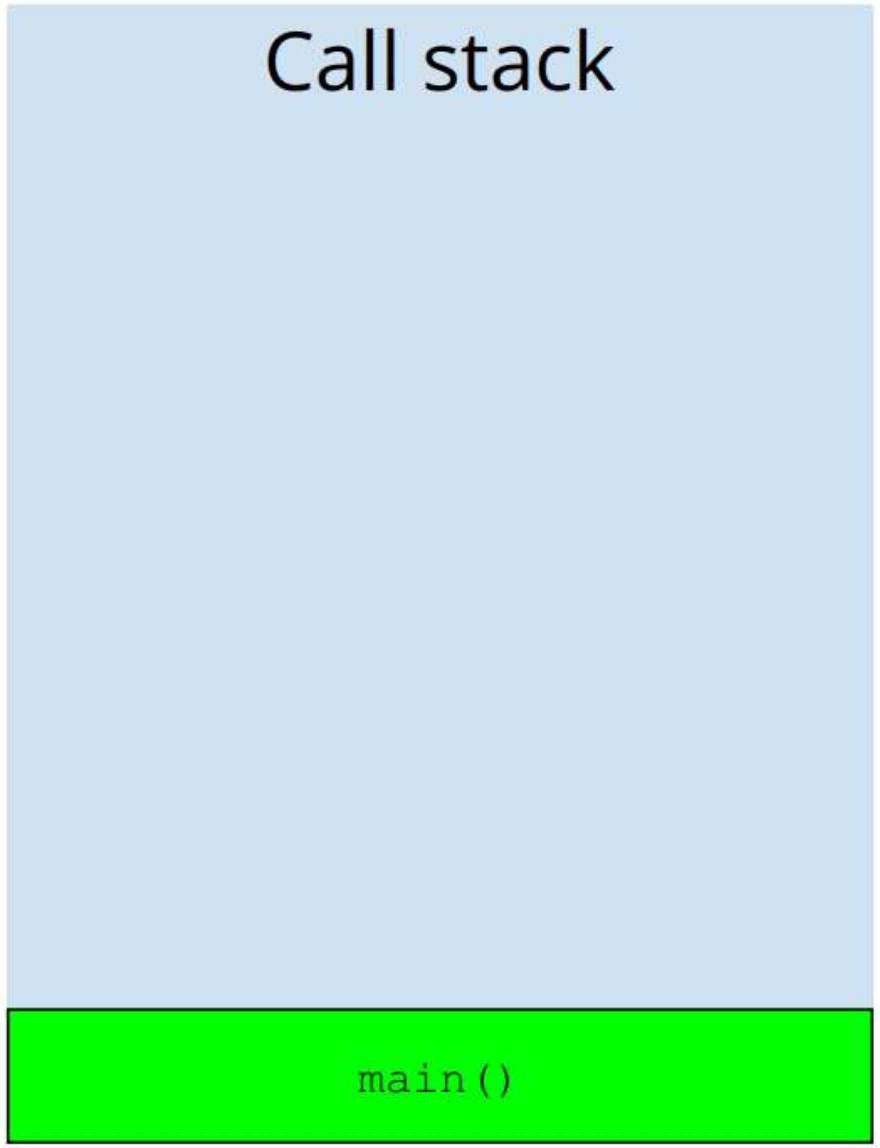


```javascript
function A() {
  throw new Error("help!");
  //console.log("ok");
}

function B() {
  A();
}

function C() {
  B();
}

C();
```

```
Uncaught Error Error: help!
    at A (file:///C:/code/Nodejs23-24/Semestr1/03/w1.js:2:9)
    at B (file:///C:/code/Nodejs23-24/Semestr1/03/w1.js:7:3)
    at C (file:///C:/code/Nodejs23-24/Semestr1/03/w1.js:11:3)
    at <anonymous> (file:///C:/code/Nodejs23-24/Semestr1/03/w1.js:14:1)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at executeUserEntryPoint (node:internal/modules/run_main:86:12)
    at <anonymous> (node:internal/main/run_main_module:23:47)
Process exited with code 1
```



```
const a = 3;

const b = a + 4;

console.log('b = ', b);
```

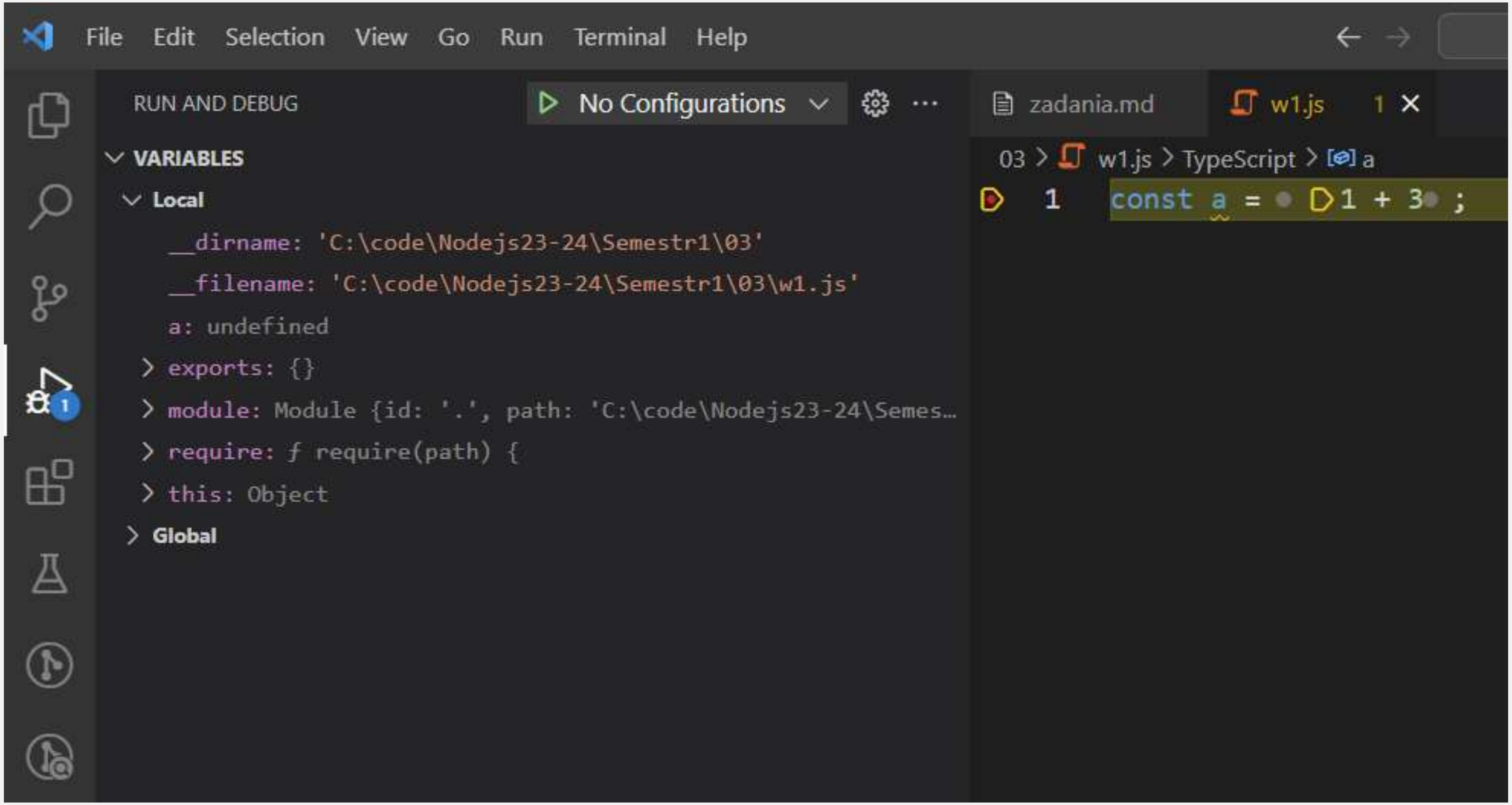


Hidden wrapping function/module wrapper

Before a module's code is executed, Node.js will wrap it with a function wrapper that looks like:

```
(function(exports, require, module, __filename, __dirname) {
    // Module code actually lives in here
});
```

Hidden wrapping function/module wrapper



```
VARIABLES
  Local
    __dirname: 'C:\code\Nodejs23-24\Semestr1\03'
    __filename: 'C:\code\Nodejs23-24\Semestr1\03\w1.js'
    a: undefined
  > exports: {}
  > module: Module {id: '.', path: 'C:\code\Nodejs23-24\Semes…
  > require: ƒ require(path) {
  > this: Object
> Global
```

```
const a = 1 + 3 ;
```

Hidden wrapping function/module wrapper

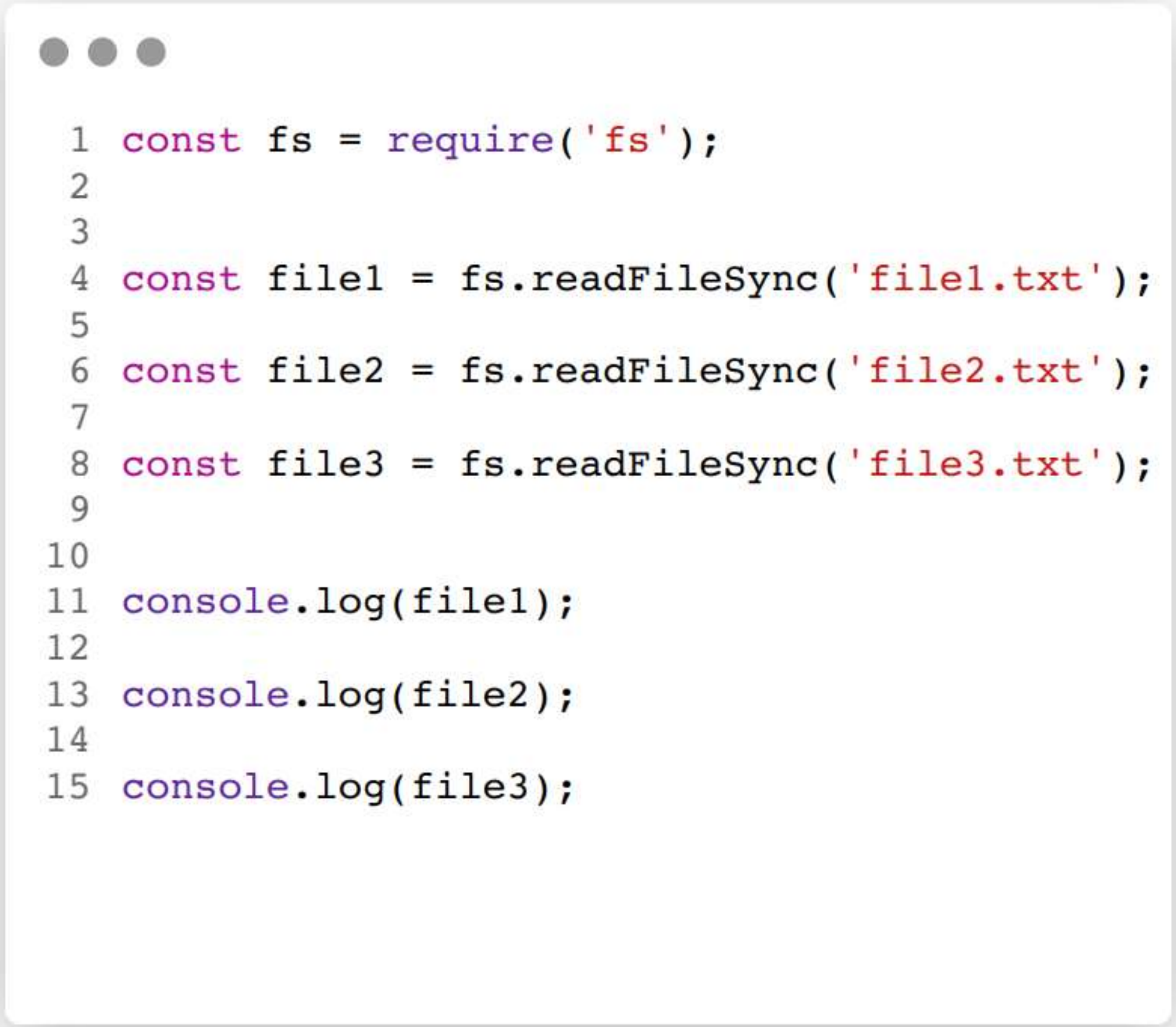
Benefits of wrapping function:

- keeps top-level variables (defined with var, const, or let) scoped to the module rather than the global object.
- helps to provide some global-looking variables that are actually specific to the module, such as:
- module and exports objects that the implementor can use to export values from the module.
- convenience variables __filename and __dirname, containing the module's absolute filename and directory path.

https://nodejs.org/api/modules.html#modules_the_module_wrapper

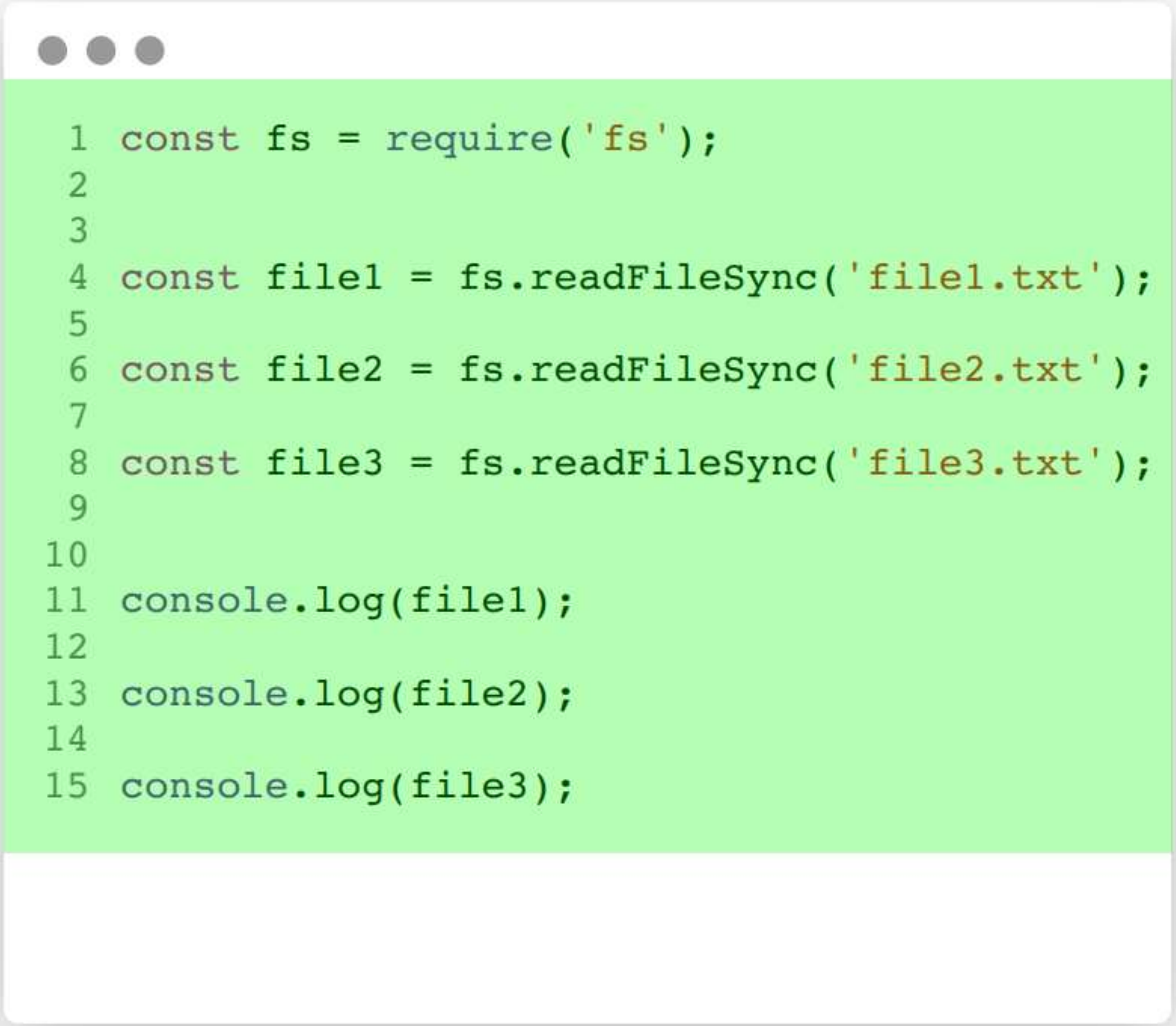
Stack – more advanced example

```
const fs = require('fs’);
const file1 = fs.readFileSync('file1.txt’);
const file2 = fs.readFileSync('file2.txt’);
const file3 = fs.readFileSync('file3.txt’);
console.log(file1);
console.log(file2);
console.log(file3);
```

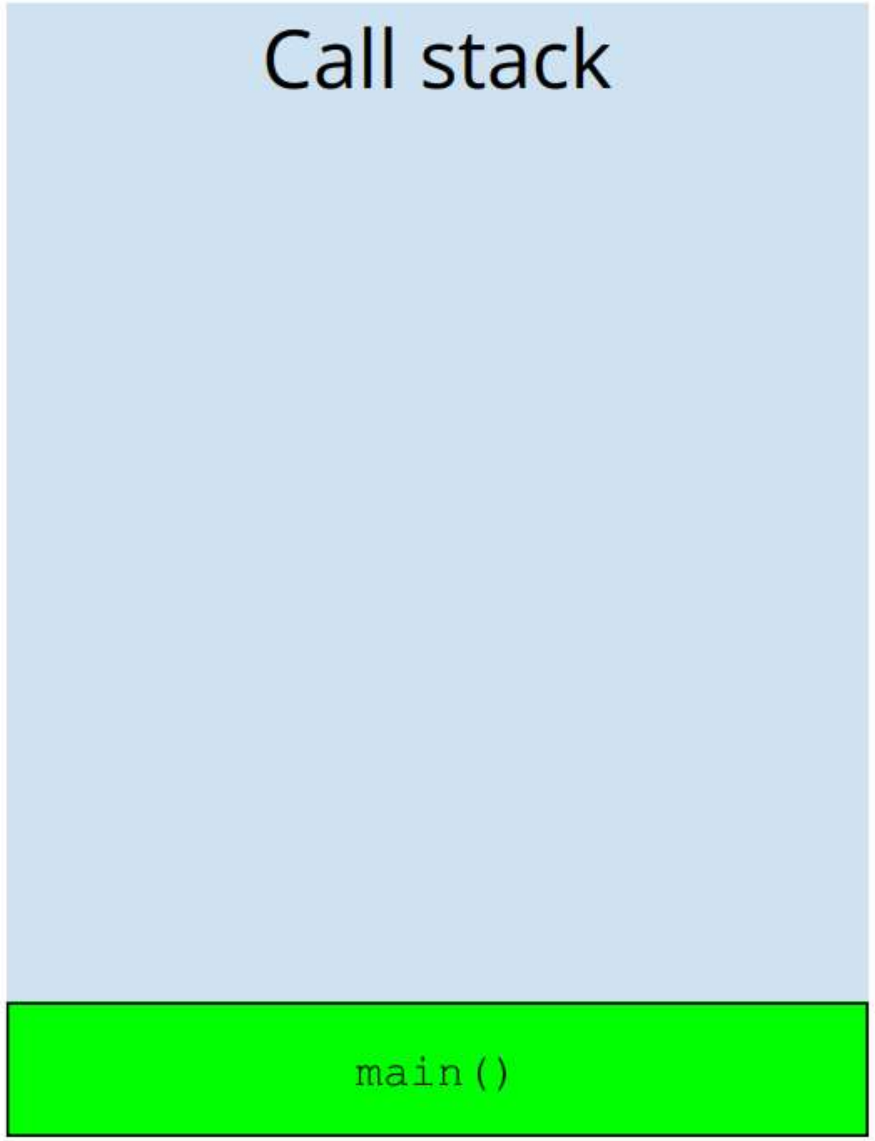


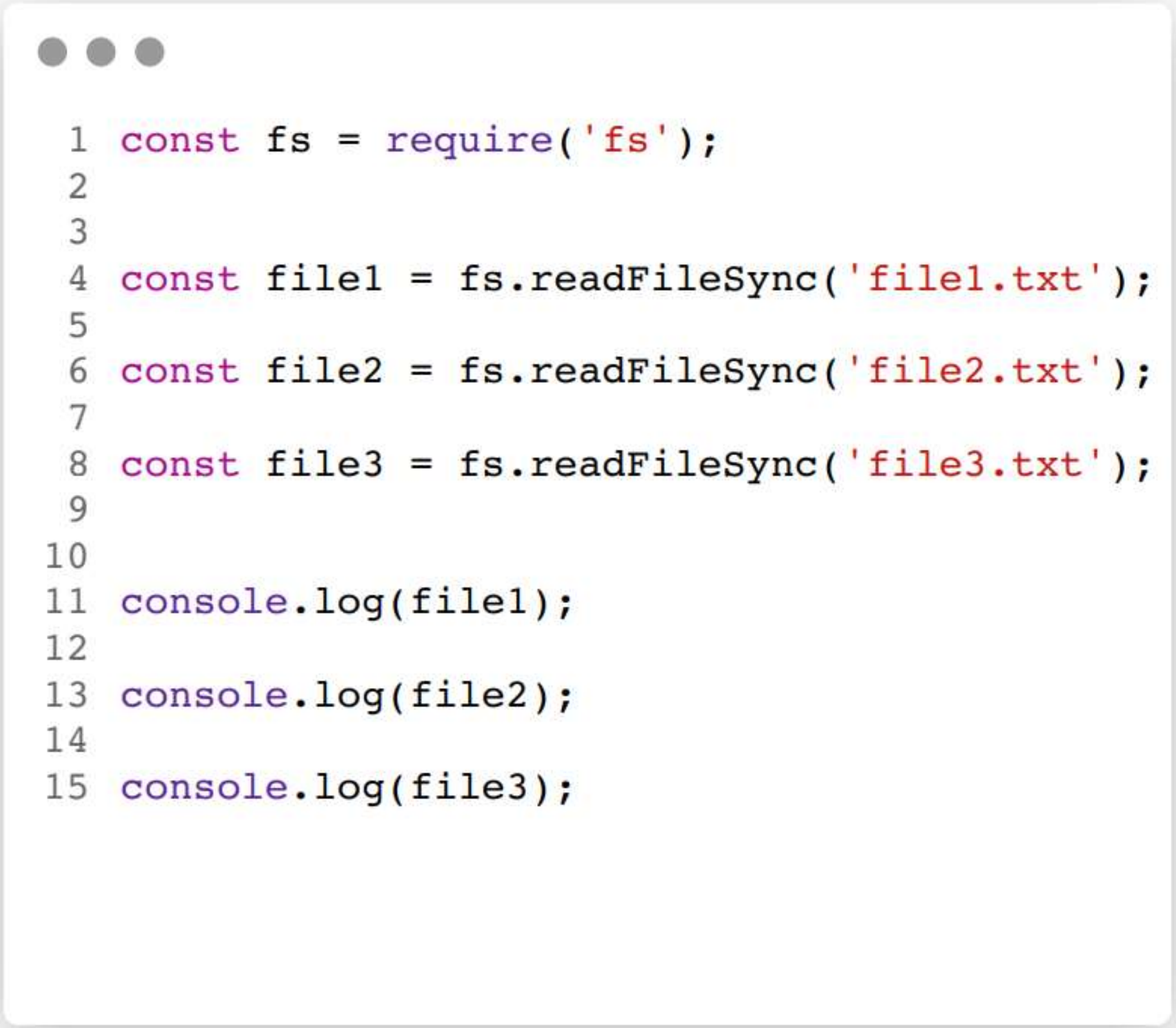
```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```

Call stack

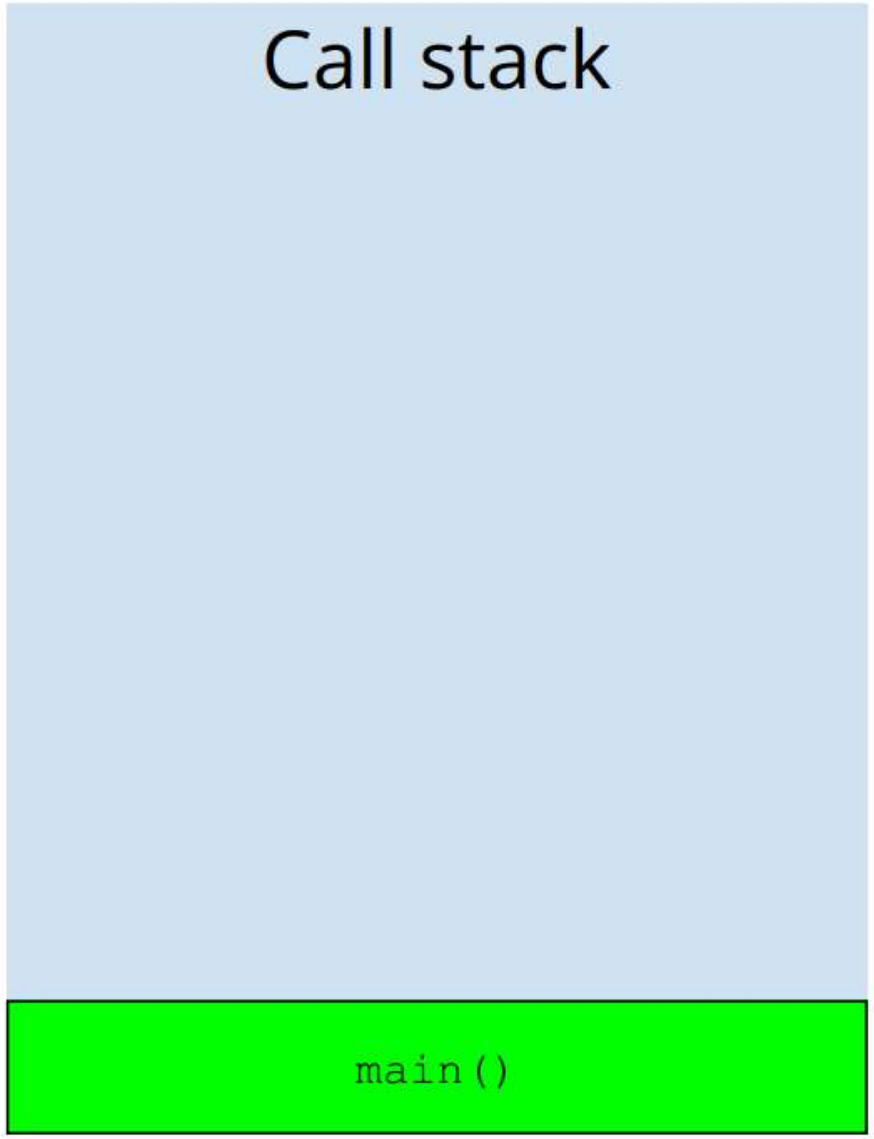


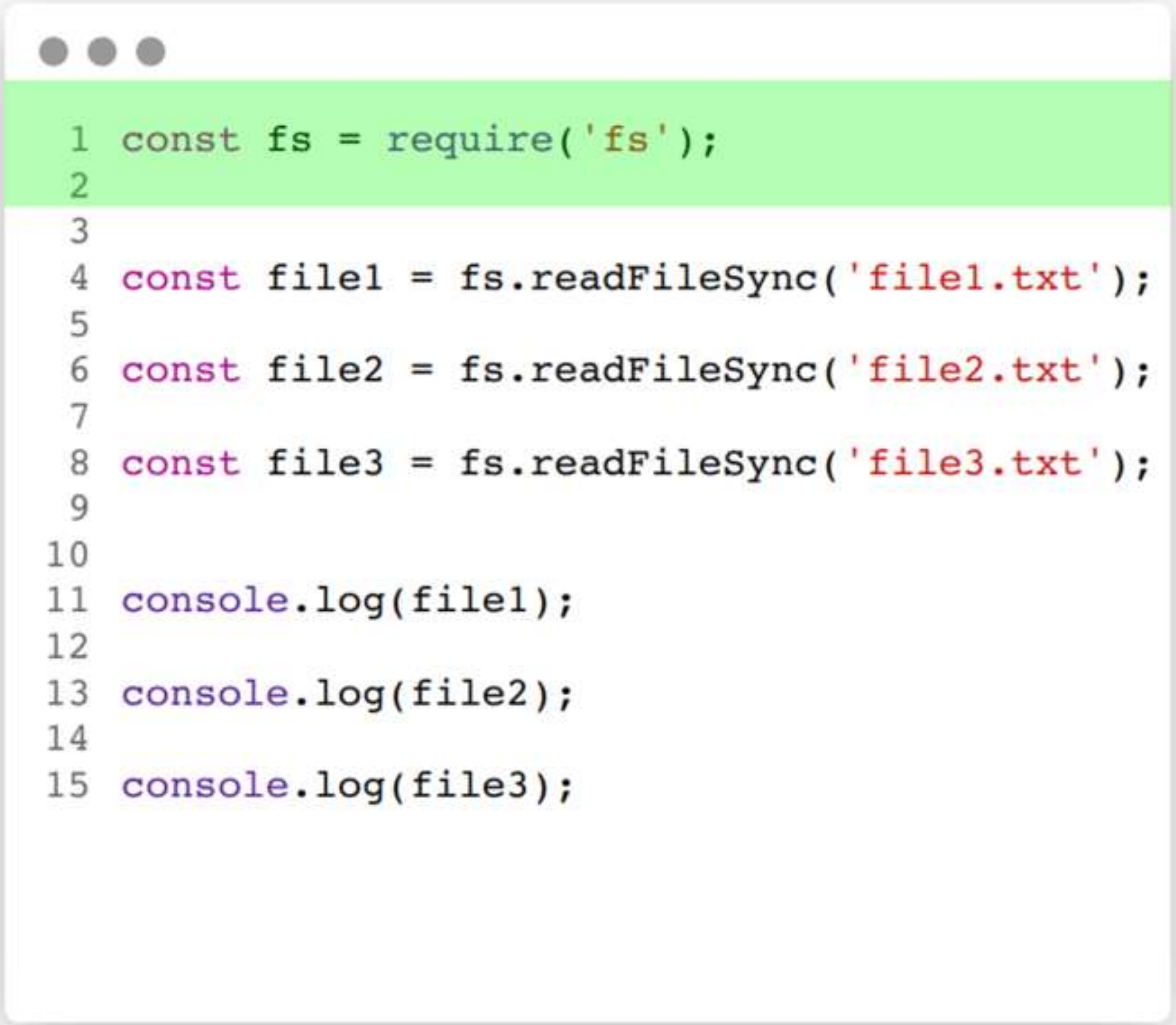
```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```



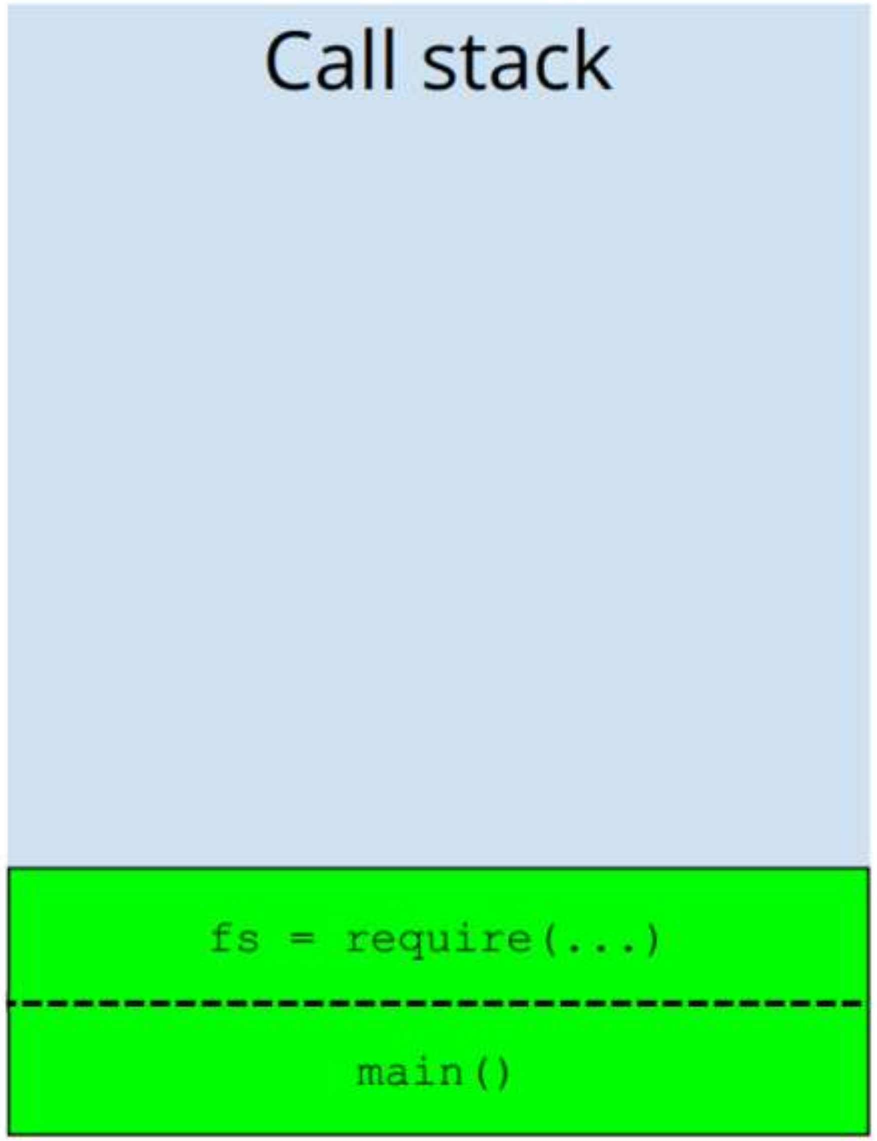


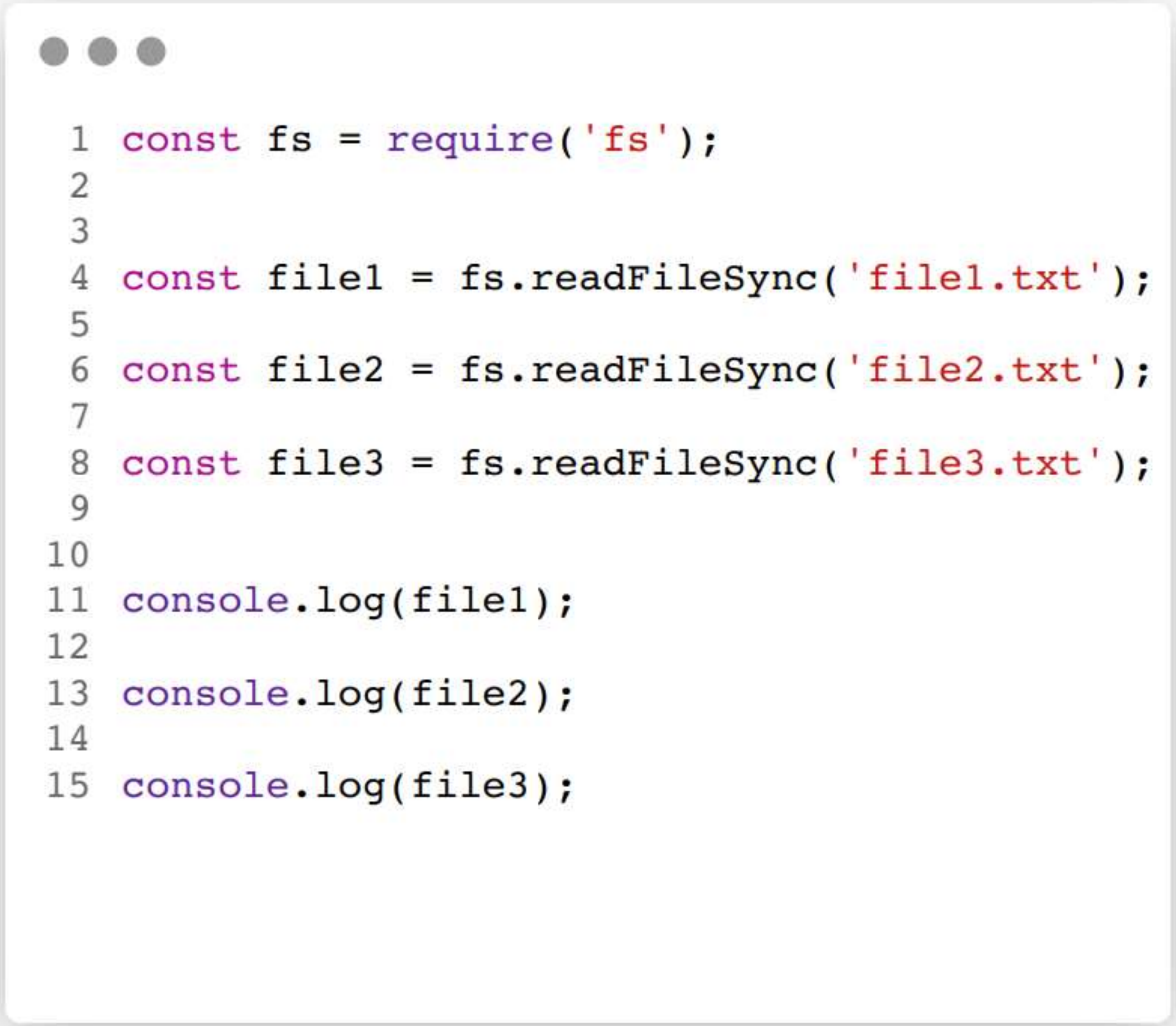
```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```



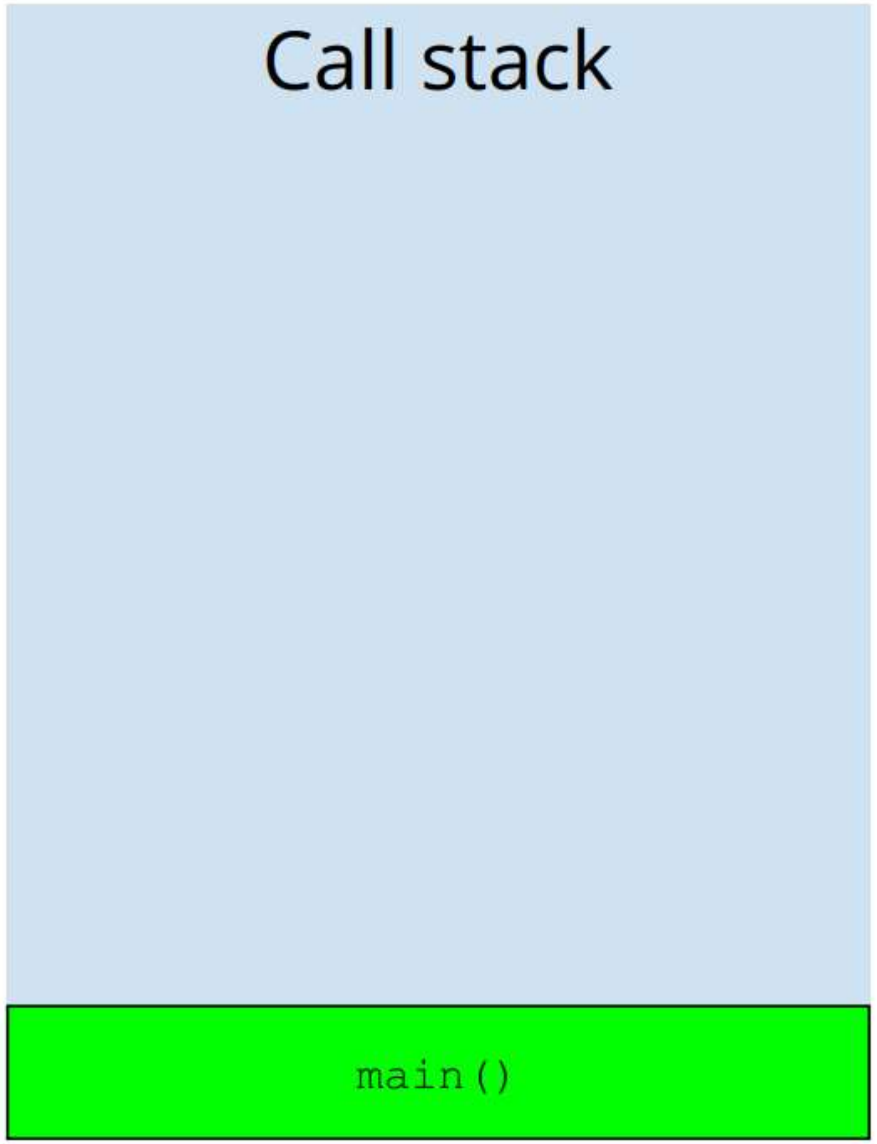


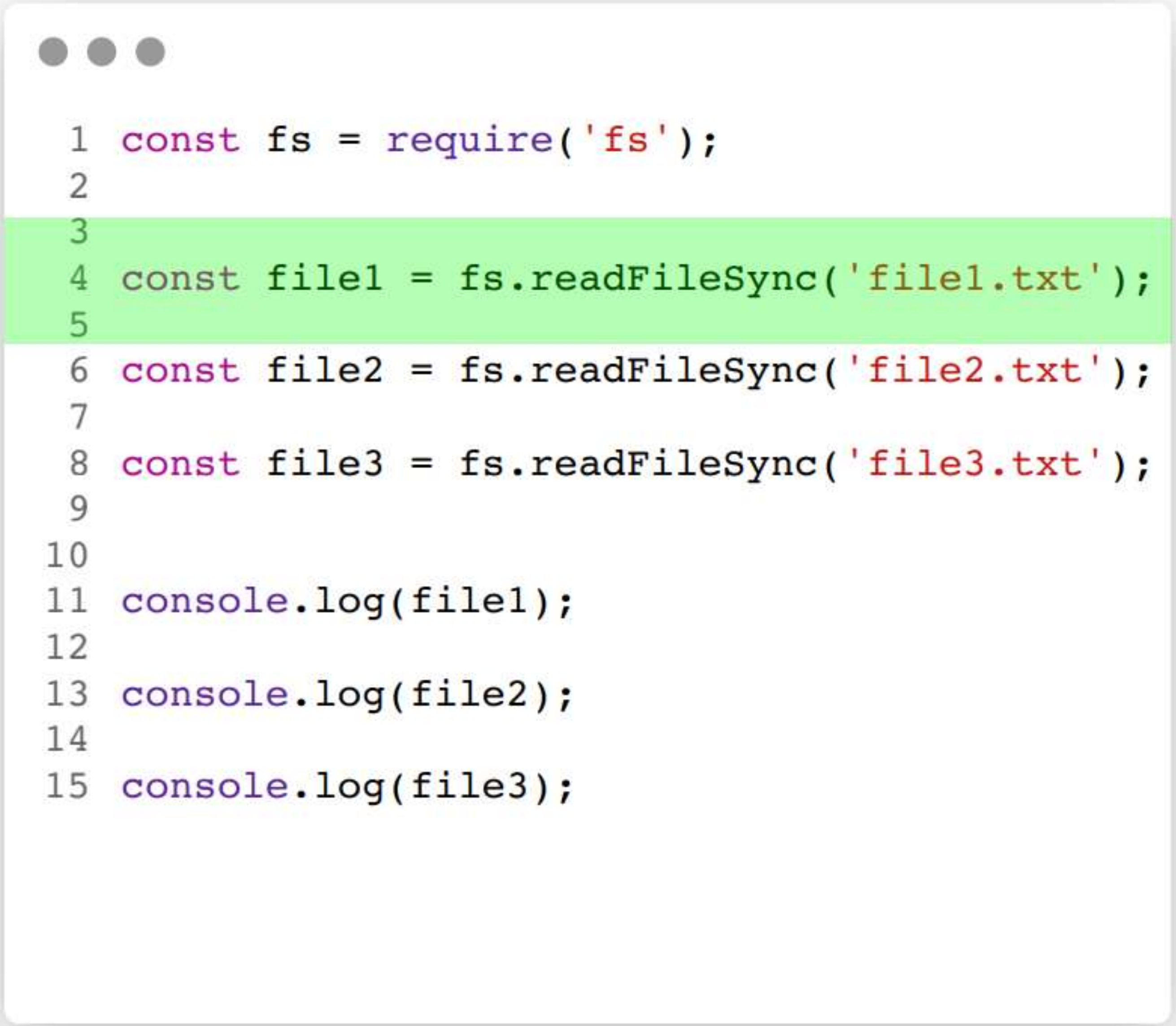
```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```



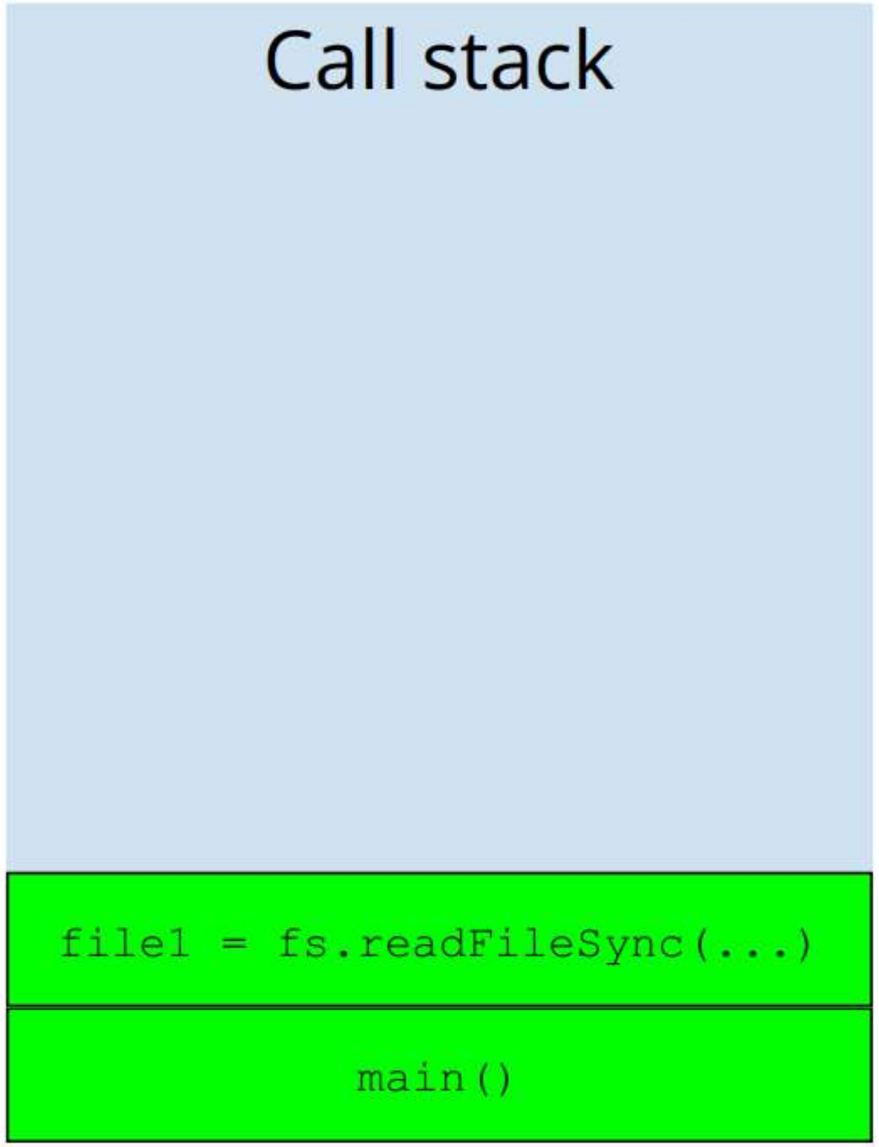


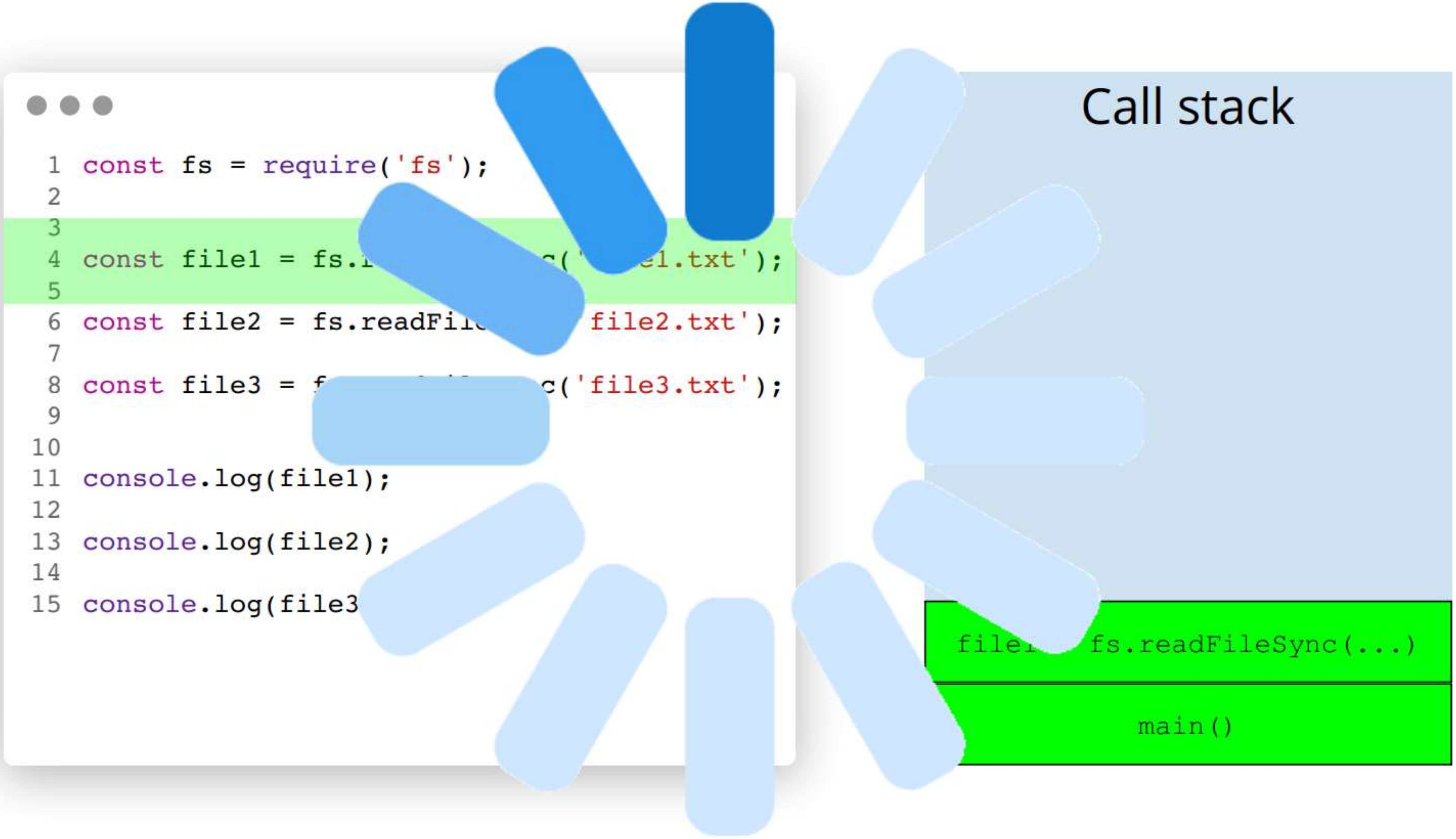
```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```





```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```





```
const fs = require('fs');


const file1 = fs.readFileSync('file1.txt');

const file2 = fs.readFileSync('file2.txt');

const file3 = fs.readFileSync('file3.txt');


console.log(file1);

console.log(file2);

console.log(file3);
```

I/O operations

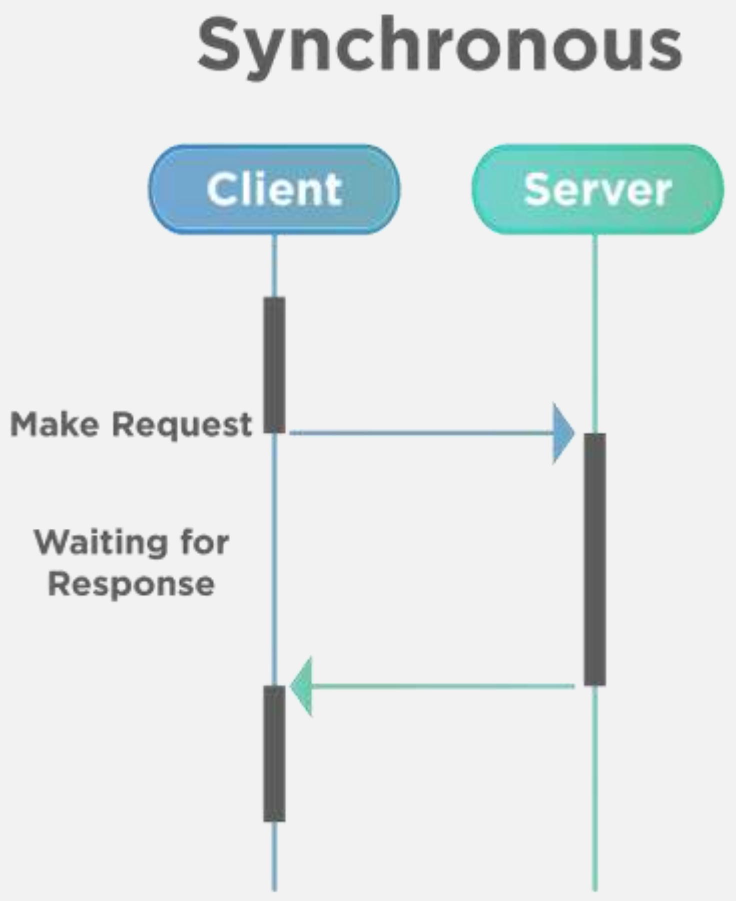
Input/output (I/O) operations are way slower than CPU in-memory operations. Difference comes from nature of these processes and their associated hardware constraints.

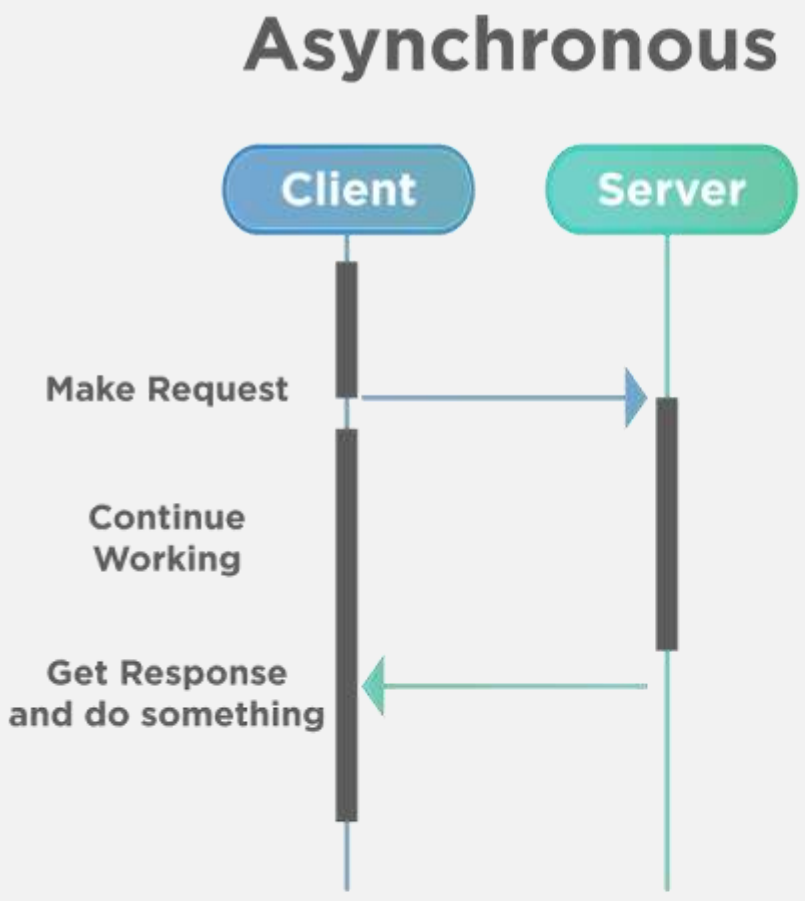
I/O operations involve communication with external devices, such as hard drives or network interfaces, which inherently possess physical limitations and slower data transfer rates compared to the rapid access within the CPU's RAM.

Mechanical delays, the nature of I/O devices contribute to the slower pace of I/O operations.

In contrast, CPU in-memory operations benefit from electronic components, high internal bus speeds, and optimized memory hierarchies, allowing for swift data retrieval and manipulation.

Sync and Async operations



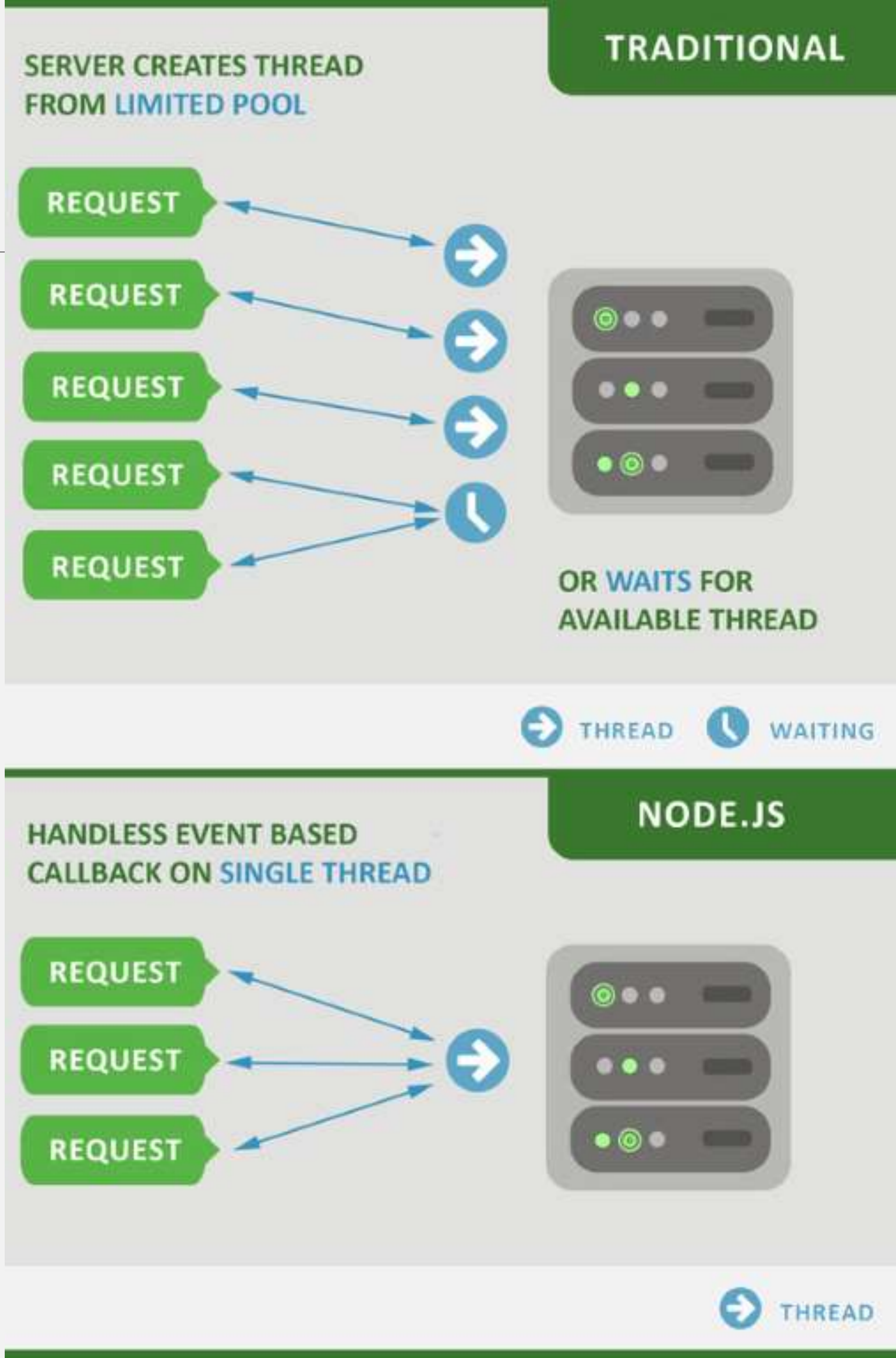


Single/Multi threading

Multithreaded processes allow the execution of multiple parts of a program at the same time. These are lightweight processes available within the process.

Single threaded processes contain the execution of instructions in a single sequence. In other words, one command is processes at a time.

Node.js is singlethreaded.


TRADITIONAL
SERVER CREATES THREAD FROM LIMITED POOL
REQUEST
REQUEST
REQUEST
REQUEST
REQUEST
OR WAITS FOR AVAILABLE THREAD
THREAD
WAITING
NODE.JS
HANDLESS EVENT BASED CALLBACK ON SINGLE THREAD
REQUEST
REQUEST
REQUEST
THREAD

API

Application Programming Interface

An API is a defined specification of possible interactions with a software component.

API doesn't have to explain what happens inside. Can describe only inputs and returned values.

Node.js APIs

Timers

File System

Network

…

Callback

A Callback is simply a function passed as an argument to another function which will then use it (call it back)

Callback function allows other code to run in the meantime.

Node.js used to make heavy use of callbacks - all the APIs of Node.js are written ir such a way that they support callbacks. It allows Node.js to process a large number of requests without waiting for any function to return the result which makes Node.js highly scalable.

For example: In Node.js, when a function start reading file, it returns the control to execution environment immediately so that the next instruction can be executed. Once file I/O gets completed, callback function will get called to avoid blocking or wait for File I/O.

Callback in SetTimeout

```
setTimeout(function () {
     console.log('my function callback');
 }, 5000);
```

Multiple callbacks with delay

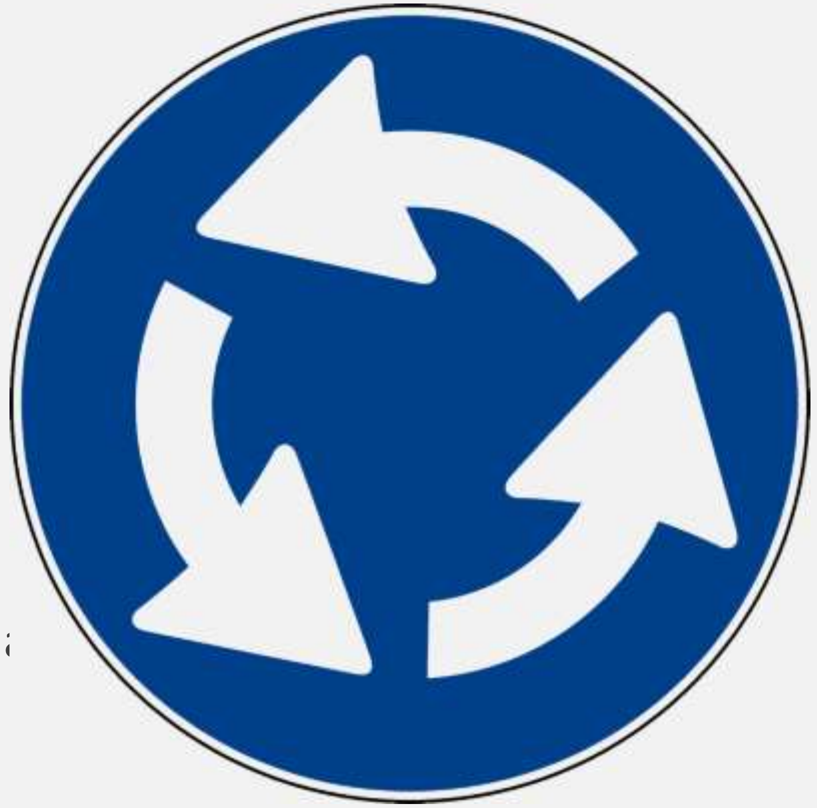
```
console.log('starting app’);

setTimeout(function callback1() {
    console.log('my function callback1')
}, 2000);

setTimeout(function callback2() {
    console.log('my function callback2')
}, 0);

console.log('end app’);
```

Multiple callbacks with small delay

```
console.log('starting app’);

setTimeout(function callback1() {
    console.log('my function callback1')
}, 2);

setTimeout(function callback2() {
    console.log('my function callback2')
}, 0);

console.log('end app’);
```

Event loop

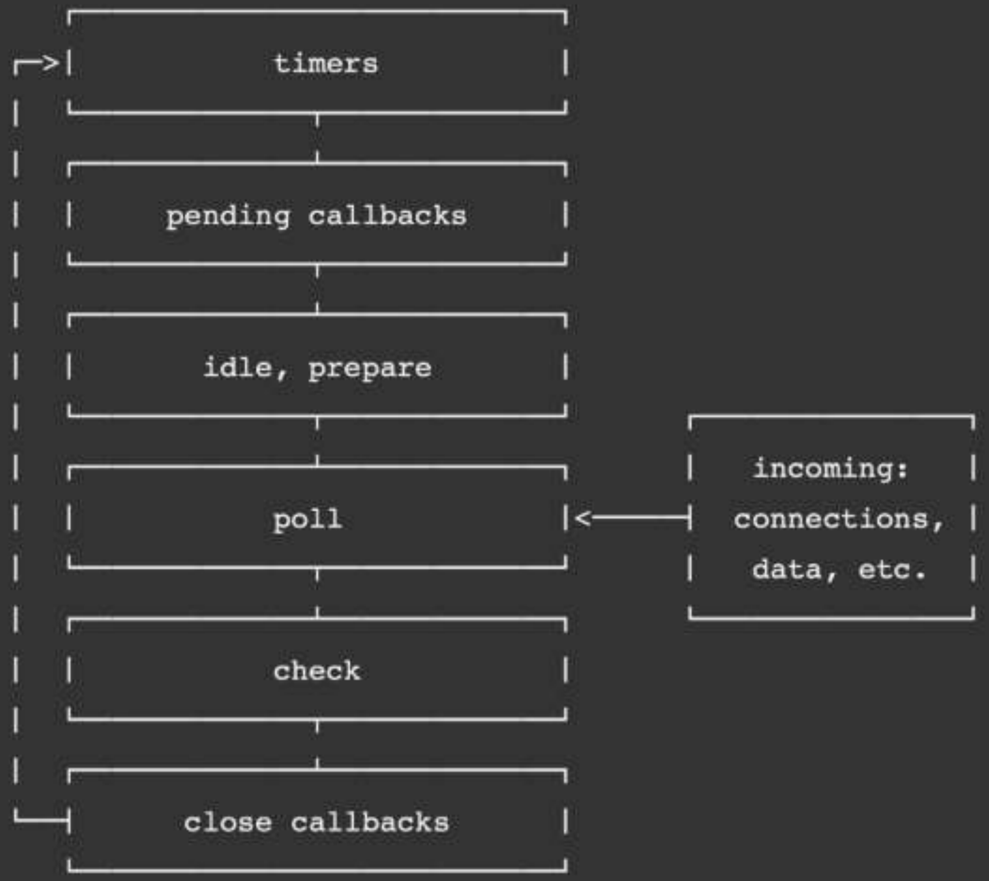
- endless loop (end with program)
- initialized on Node.js start
- executes tasks only when the call stack is empty
- waits for tasks, executes them and then sleeps until it receives more tasks
- allows us to use callbacks and promises
- executes the tasks starting from the oldest first (oldest that ready to process)



Event loop in details

Deep dive into event loop (not required for this course)

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/



Summary

Node.js is a single-threaded non-blocking asynchronous concurrent technology

=

Node.js has:

call stack, event loop, callback queue

and some other APIs stuff

Callback example

```
const fs = require("fs");

var myCallbackFunction = function (err, data) {
   console.log(data.toString());
}

fs.readFile("input.txt", myCallbackFunction);

console.log("Program Ended");
```

Callback example #2

```
const fs = require("fs");

fs.readFile("input.txt", function (err, data) {
   console.log(data.toString());
});

console.log("Program Ended");
```

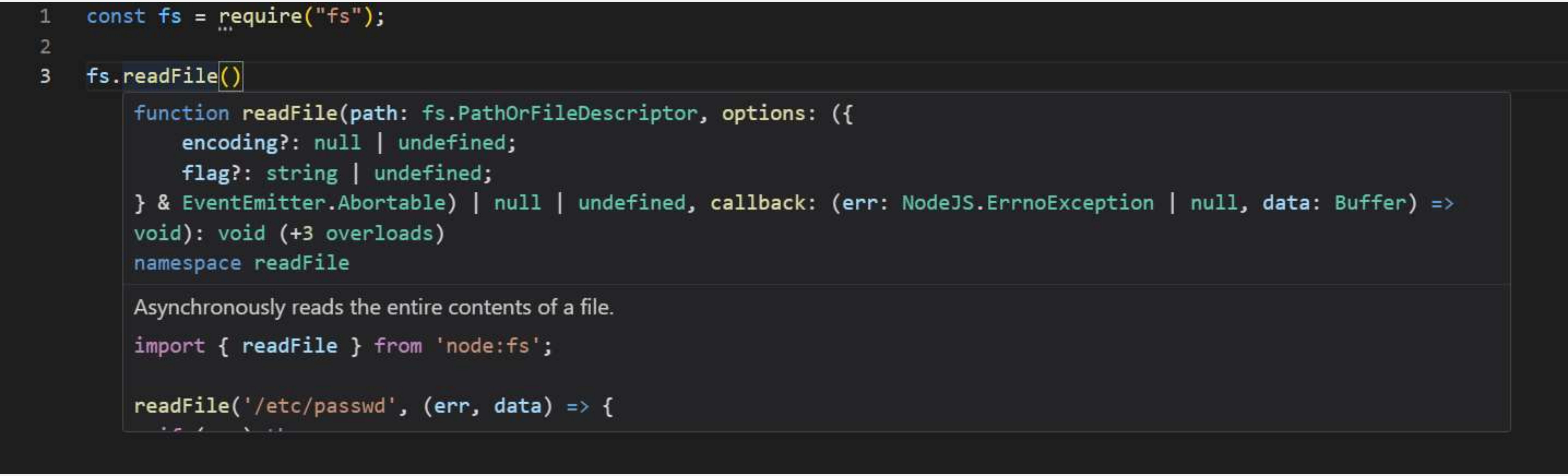
Error-First callback pattern

In Node.js, it is considered standard practice to handle errors in asynchronous functions by returning them as the first argument to the current function's callback.

If there is an error, the first parameter is passed an Error object with all the details. Otherwise, the first parameter is null.

```
var callback = function (error, retval) {
    if (error) {    // something went wrong
        console.log(error); // log error
        return; // and leave
    }

    console.log(retval); // ok, we can process returned value
}
```

Error-First callback pattern



```
const fs = require("fs");

fs.readFile()
```

```
function readFile(path: fs.PathOrFileDescriptor, options: ({
    encoding?: null | undefined;
    flag?: string | undefined;
} & EventEmitter.Abortable) | null | undefined, callback: (err: NodeJS.ErrnoException | null, data: Buffer) =>
void): void (+3 overloads)
namespace readFile
```

Asynchronously reads the entire contents of a file.

```
import { readFile } from 'node:fs';

readFile('/etc/passwd', (err, data) => {
```

Nested callback

```
const makeBurger = () => {
    getBeef(function (beef) {
        cookBeef(beef, function (cookedBeef) {
            getBuns(function (buns) {
                // Put patty in bun
            })
        })
    })
}
```

Nested callback - example

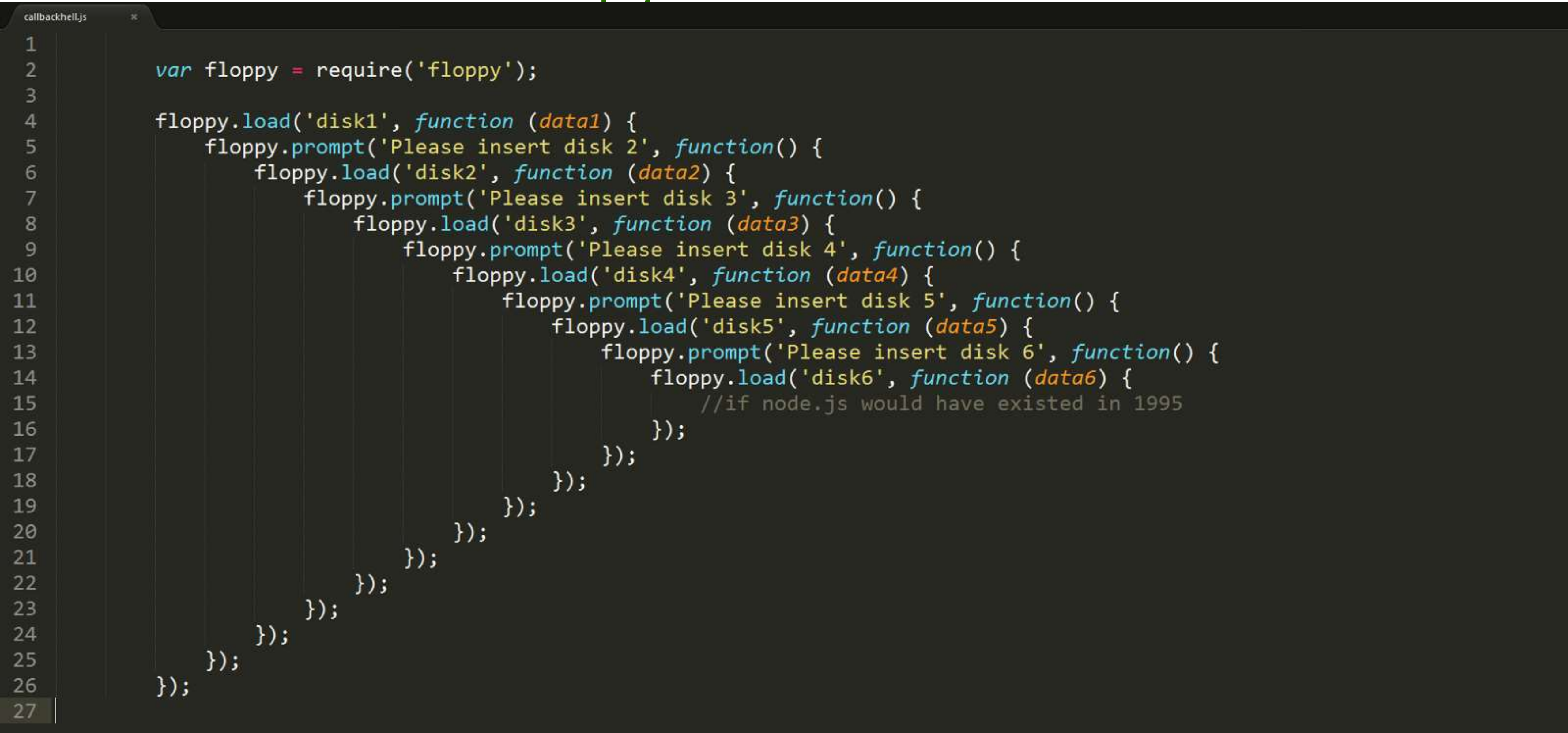
We would like to add couple of songs to a playlist on Spotify.

Here are the steps that we need:

- Retrieve temporary access token
- Retrieve user’s id using the access token that we just got
- Create a brand new empty playlist
- Try to look for the song on Spotify for every song on the list
- Since we got the user’s id from step 2 as well as the playlist’s id from step 3, we should now be able to add songs to the playlist on Spotify

```
post("https://accounts.spotify.com/api/token", {}, urlencode({
    grant_type: 'authorization_code',
    code: getParam(tab.url, 'code'),
    redirect_uri: "https://www.lukaszuk.net/spotify.html",
    client_id: ":)",
    client_secret: ":)",
}), function (response) {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    get("https://api.spotify.com/v1/me", {
        Authorization: tokenType + ' ' + accessToken
    }, null, function (response) {
        var userId = response.id;
        post("https://api.spotify.com/v1/users/" + userId + "/playlists", {
            Authorization: tokenType + ' ' + accessToken,
            "Content-type": "application/json"
        }, JSON.stringify({
            name: localStorage.playlistTitle
        }), function (response) {
            var playlistId = response.id;
            var songs = JSON.parse(localStorage.songs);
            var i = 0;
            for (key in songs) {
                get("https://api.spotify.com/v1/search", {
                    Authorization: tokenType + ' ' + accessToken
                }, "q=" + songs[key].title + "%20album:" + songs[key].album + "%20artist:
" + songs[key].artist + "&type=track", function (response) {
                    if (response.tracks.items.length) {
                        var uri = response.tracks.items[0].uri;
                        post("https://api.spotify.com/v1/users/" + userId + "/playlists/"
 + playlistId + "/tracks", {
                            Authorization: tokenType + ' ' + accessToken,
                            "Content-type": "application/json"
                        }, JSON.stringify({
                            uris: [uri]
                        }), function (response) {
                            // song has been added to the playlist
                        });
                    }
                });
            };
        });
    });
});
```

Callback hell / pyramide of doom



```

var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function() {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function() {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function() {
                                floppy.load('disk5', function (data5) {
                                    floppy.prompt('Please insert disk 6', function() {
                                        floppy.load('disk6', function (data6) {
                                            //if node.js would have existed in 1995
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});

```

Solution #1 - comments

General rule says that you should avoid putting comments in your code.

Sometimes using comments is justified and can bring benefits.

```
// function get(url, header, param, success) {...}

// function post(url, header, param, success) {...}

// Retrieve temporary access token

post("https://accounts.spotify.com/api/token", {}, urlencode({

    grant_type: 'authorization_code',

    code: getParam(tab.url, 'code'),
    redirect_uri: "https://www.lukaszuk.net/sfy.html",
    client_id: ":)",
    client_secret: ":)",

}), function (response) {

    var tokenType = response.token_type;

    var accessToken = response.access_token;

    // Retrieve user's id using the access token that we just got

    get("https://api.spotify.com/v1/me", {

        Authorization: tokenType + ' ' + accessToken
    }, null, function (response) {

        var userId = response.id;

        // Create a brand new empty playlist

        post("https://api.spotify.com/v1/users/" + userId + "/playlists", {

            Authorization: tokenType + ' ' + accessToken,

            "Content-type": "application/json"

        }, JSON.stringify({

            name: localStorage.playlistTitle

        }), function (response) {

            var playlistId = response.id;

            var songs = JSON.parse(localStorage.songs);

            var i = 0;

            // Try to look for the song on Spotify for every song on the list

            for (key in songs) {

                get("https://api.spotify.com/v1/search", {

                    Authorization: tokenType + ' ' + accessToken

                }, "q=" + songs[key].title + "%20album:" + songs[key].album + "%20artist:" +
songs[key].artist + "&type=track", function (response) {

                    if (response.tracks.items.length) {

                        var uri = response.tracks.items[0].uri;

                        // Since we got the user's id from step 2 as well as the playlist's i
d from step 3, we should now be able to add songs to the playlist on Spotify

                        post("https://api.spotify.com/v1/users/" + userId + "/playlists/" + p
laylistId + "/tracks", {

                            Authorization: tokenType + ' ' + accessToken,

                            "Content-type": "application/json"

                        }, JSON.stringify({

                            uris: [uri]

                        }), function (response) {

                            // song has been added to the playlist

                        });

                    }

                });

            };

        });

    });

});
```

Solution #2 - smaller functions

Splitting long function into multiple smaller functions is always good idea.

Small pieces of code are:

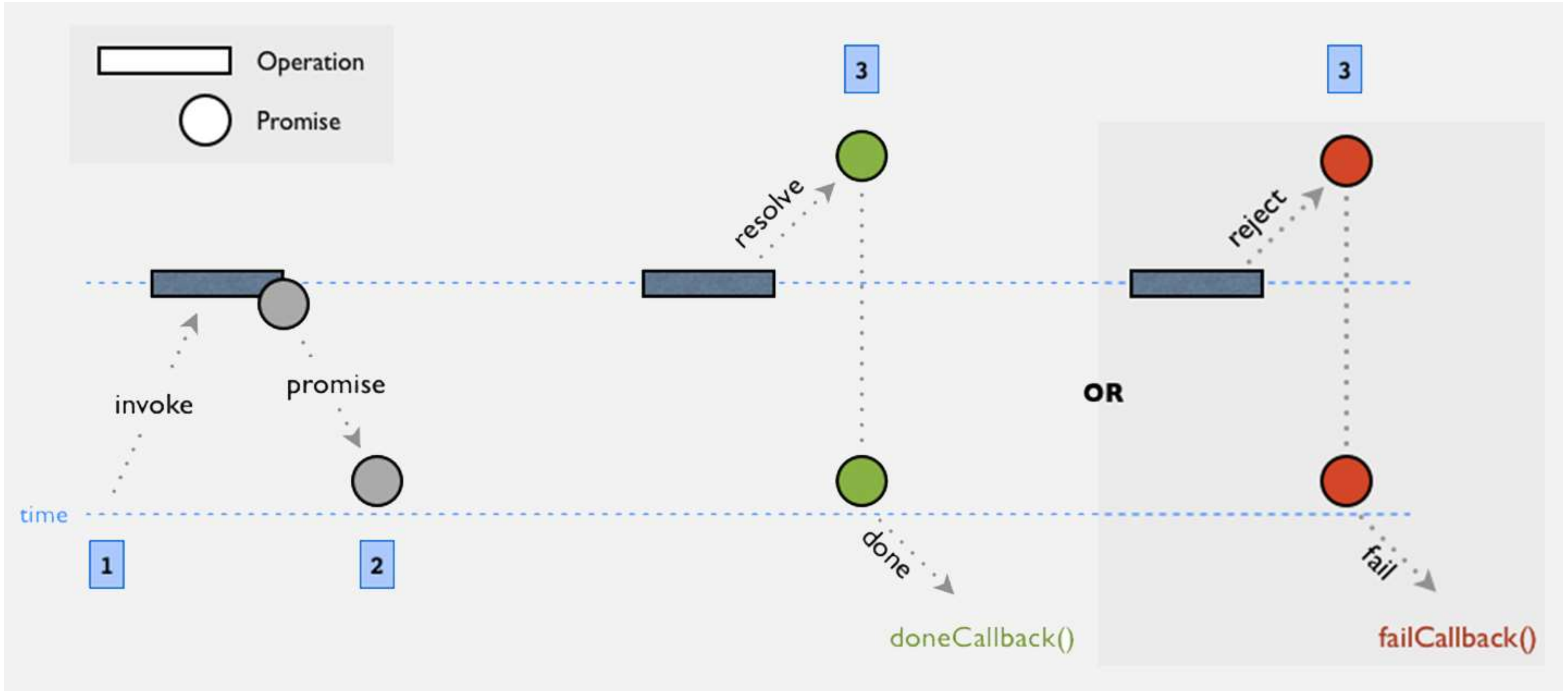
- easier to read
- easier to understand
- easier to change
- easier to test
- easier to reuse

```
// function get(url, header, param, success) {...}
// function post(url, header, param, success) {...}

var tokenType, accessToken, userId, playlistId, songs = JSON.parse(localStorage.songs);

function retrieveAccessToken(callback) {
    post("https://accounts.spotify.com/api/token", {}, urlencode({
        grant_type: 'authorization_code',
        code: getParam(tab.url, 'code’),
        redirect_uri: "https://www.lukaszuk.net/sfy.html",
        client_id: ":)",
        client_secret: ":)",
    }), function (response) {
        callback(response);
    });
}

function retrieveUserId(response, callback) {
    tokenType = response.token_type;
    accessToken = response.access_token;
    get("https://api.spotify.com/v1/me", {
        Authorization: tokenType + ' ' + accessToken
    }, null, function (response) {
        callback(response);
    });
}

function createANewPlaylist(response, callback) {
    userId = response.id;
    post("https://api.spotify.com/v1/users/" + userId + "/playlists", {
        Authorization: tokenType + ' ' + accessToken,
        "Content-type": "application/json"
    }, JSON.stringify({
        name: localStorage.playlistTitle
    }), function (response) {
        callback(response);
    });
}

function searchASong(key, callback) {
    get("https://api.spotify.com/v1/search", {
        Authorization: tokenType + ' ' + accessToken
    }, "q=" + songs[key].title + "%20album:" + songs[key].album + "%20artist:" + songs[key].
artist + "&type=track", function (response) {
        callback(response);
    });
}

function addASongToThePlaylist(uri, callback) {
    post("https://api.spotify.com/v1/users/" + userId + "/playlists/" + playlistId + "/track
s", {
        Authorization: tokenType + ' ' + accessToken,
        "Content-type": "application/json"
    }, JSON.stringify({
        uris: [uri]
    }), function (response) {
        callback(response);
    });
}

function addAllSongsToPlayList(response, callback) {
    playlistId = response.id;
    var i = 0;
    for (key in songs) {
        searchASong(key, function (response) {
            if (response.tracks.items.length) {
                addASongToThePlaylist(response.tracks.items[0].uri, function (response) {
                    i++;
                });
            }
        });
    }
    callback(i);
}

retrieveAccessToken(function (response) {
    retrieveUserId(response, function (response) {
        createANewPlaylist(response, function (response) {
            addAllSongsToPlayList(response, function (total) {
                console.log("There are " + total + " out of " + songs.length + " songs been
added to the playlist!!!");
            });
        });
    });
});
```

Solution #3 - using promises

Promise - class that allows you to create objects, representing value or failure of async operation.

Promise represents an operation that is not yet finished, but it is expected to end in the future.



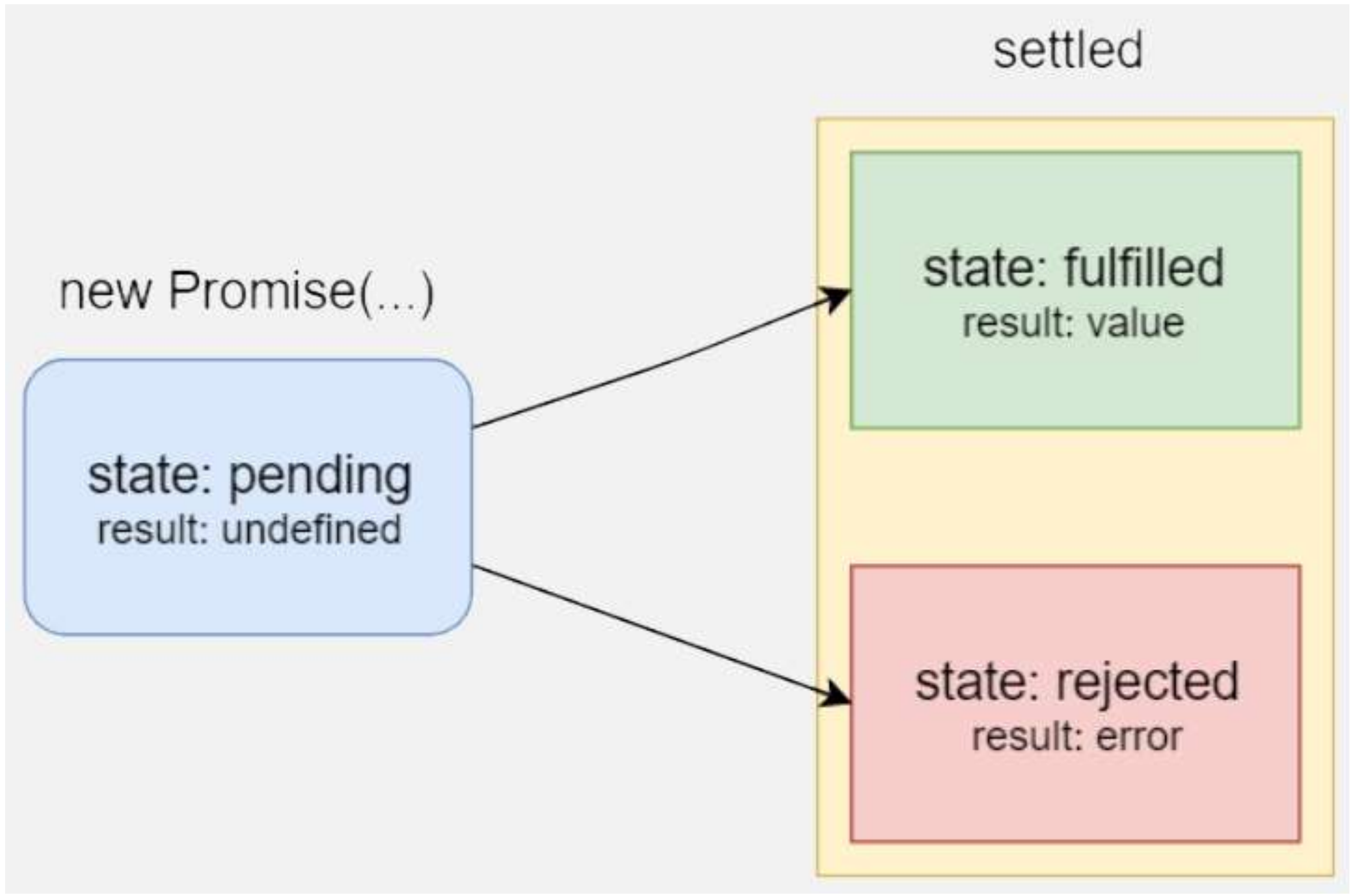
Callback vs Promise

- Callback is a function, Promise is an object
- Callback accepts parameters, Promise return value
- Callback supports success and error, Promise handles nothing but pass on values
- Callback can be called many times, Promise is only called once

Promise

Promise states:

- pending
- settled:
  - fulfilled
  - rejected



Create Promise

```
const myPromise = new Promise((resolve, reject) => {
    /* some logic */
    if (/* some condition */) {
        resolve('all works fine');
    } else {
        reject('error');
    }
});
```

Converting Callbacks into Promises

In practice, callbacks would probably be written for you already.

If you use Node, each function that contains a callback will have the same syntax:

- the callback would be the last argument
- the callback will always have two arguments. And these arguments are in the same order. (Error first, followed by whatever you're interested in).

If your callback has the same syntax, you can use libraries like:

es6-promisify or Node.js util.promisify.

Converting Callbacks into Promises

```
//callback
const fs = require('fs');



const data = 'ala ma kota';
fs.writeFile('some-file.txt’,
   data,
  () => {
      console.log('file saved’);
  }
)
```

```
//promise
const util = require('util');
const fs = require('fs');

const writePromise = util.promisify(fs.writeFile);
const data = 'ala ma kota';
writePromise('some-file.txt', data)
    .then(() => {
        console.log('file saved');
    });
```

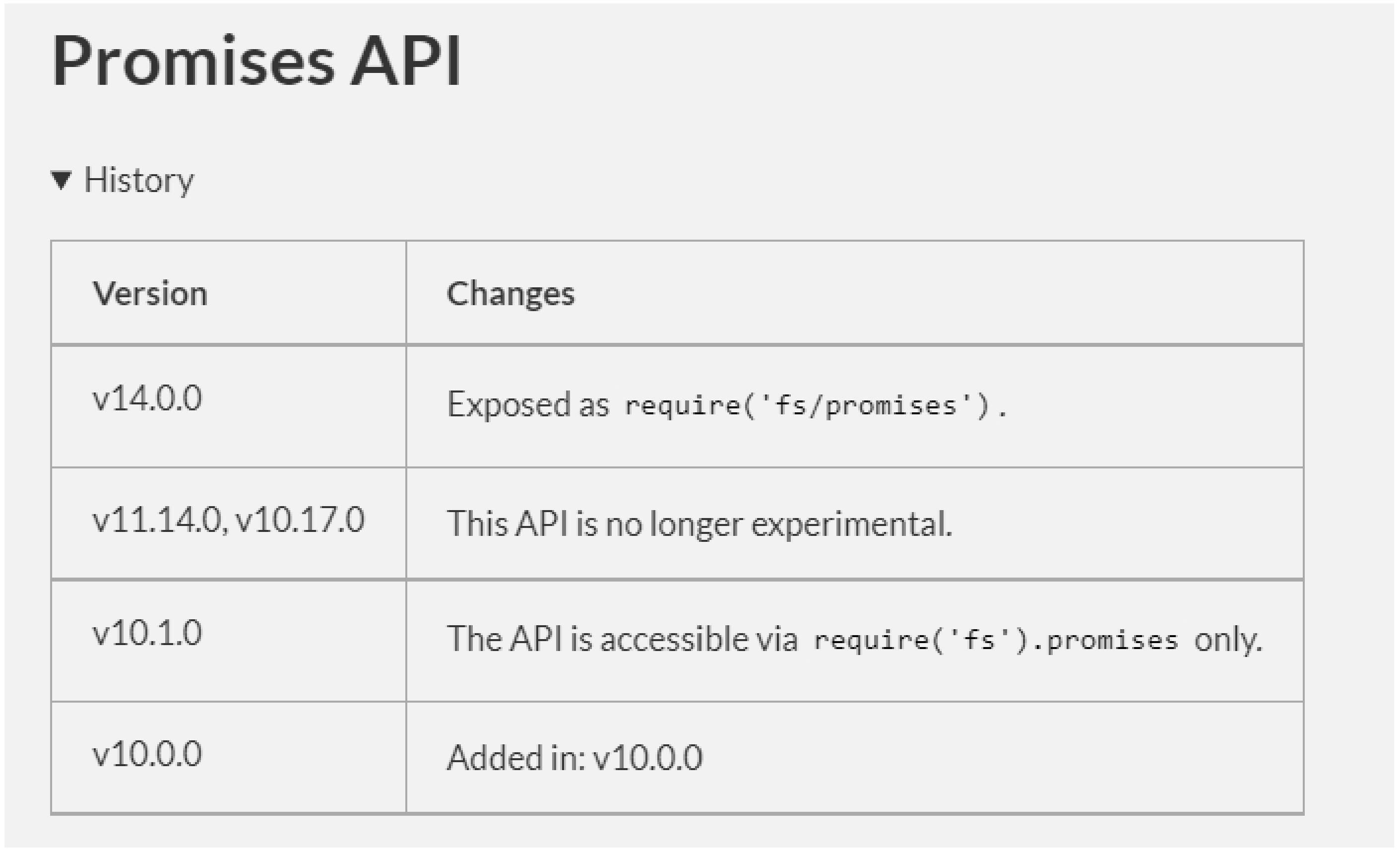
Promise syntax

```
retrieveAccessToken(tab.url)
    .then(retrieveUserInfo)
    .then(createAPlaylist)
    .then(getAllSongsInfo)
    .then(prepareToaddAllSongsToPlaylist)
    .then(addAllSongsToPlaylist)
    .catch(error => {
        // error handling
    });
```

```javascript
const retrieveAccessToken = url => {
    return new Promise(resolve => {
        post("https://accounts.spotify.com/api/token", {}, ur
lencode({
           grant_type: 'authorization_code',
           code: getParam(url, 'code’),
           redirect_uri: "https://www.lukaszuk.net/sfy.html",
           client_id: ":)",
           client_secret: ":)",
       }), response => {
           resolve(response);
       });
    })
};

const retrieveUserInfo = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    return new Promise(resolve => {
        get("https://api.spotify.com/v1/me", {
            Authorization: tokenType + ' ' + accessToken
        }, null, response => {
            response['token_type'] = tokenType
            response['access_token'] = accessToken;
            return resolve(response);
        });
    });
};

const createAPlaylist = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    var userId = response.id;
    return new Promise(resolve => {
        post("https://api.spotify.com/v1/users/" + userId + "
/playlists", {
            Authorization: tokenType + ' ' + accessToken,
            "Content-type": "application/json"
        }, JSON.stringify({
            name: localStorage.playlistTitle
        }), response => {
            response['token_type'] = tokenType
            response['access_token'] = accessToken;
            response['userId'] = userId;
            return resolve(response);
        });
    });
};

const searchASong = response => {
    return new Promise(resolve => {
        get("https://api.spotify.com/v1/search", {
            Authorization: response.token_type + ' ' + respon
se.access_token
        }, buildSearchQuery(response.song), responseFromSearc
h => {
            resolve(responseFromSearch.tracks.items[0]);
        });
    });
};


const getAllSongsInfo = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    var playlistId = response.id;
    var userId = response.userId;
    var songs = JSON.parse(localStorage.songs);
    var allSearchPromises = [];
    for (key in songs) {
        response['song'] = songs[key];
        allSearchPromises.push(searchASong(response));
    }
    return Promise.all(allSearchPromises).then(function (resp
onse) {
        response['token_type'] = tokenType;
        response['access_token'] = accessToken;
        response['playlistId'] = playlistId;
        response['userId'] = userId;
        return response;
    });
};

const prepareToaddAllSongsToPlaylist = response => {
    var songs = [];
    for (key in response) {
        if (isNumeric(key)) {
            songs.push(response[key].uri);
        }
    }
    return new Promise(resolve => {
        response['songs'] = songs;
        resolve(response);
    });
};

const addAllSongsToPlaylist = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    var playlistId = response.playlistId;
    var userId = response.userId;
    var songs = response.songs;
    return new Promise(resolve => {
        post("https://api.spotify.com/v1/users/" + userId + "
/playlists/" + playlistId + "/tracks", {
            Authorization: tokenType + ' ' + accessToken,
            "Content-type": "application/json"
        }, JSON.stringify({
            uris: songs
        }), function (response) {
            resolve(response);
        });
    });
};

function isNumeric(n) {
    return !isNaN(parseFloat(n)) && isFinite(n);
}

function buildSearchQuery(song) {
    return "q=" + song.title +         "%20album:" + song.alb
um +
        "%20artist:" + song.artist + "&type=track";
}

retrieveAccessToken(tab.url)
    .then(retrieveUserInfo)
    .then(createAPlaylist)
    .then(getAllSongsInfo)
    .then(prepareToaddAllSongsToPlaylist)
    .then(addAllSongsToPlaylist)
    .catch(error => {
        progress.innerHTML += "[WARNING] " + error + "<br>";
    });
```

Node.js v18.13.0 documentation

▼ Table of contents


Promises API

▼ History

Version	Changes
v14.0.0	Exposed as `require('fs/promises')`.
v11.14.0, v10.17.0	This API is no longer experimental.
v10.1.0	The API is accessible via `require('fs').promises` only.
v10.0.0	Added in: v10.0.0

```
const fs = require("fs").promises;
// or
const fs = require("fs/promises");
```

Solution #4 – using async/await

Data return from async functions is automatically wrapped in a promise.

Using await keyword will automatically extract data from promise.

Await keyword can be used only inside functions marked as async

Async/await - syntax

```
// PROMISE
function doWork() {
    asyncAction()
        .then(data => {
            console.log(data);
        });
}

doWork();
```

Async/await - syntax

```
// PROMISE
function doWork() {
    asyncAction()
        .then(data => {
            console.log(data);
        });
}

doWork();
```

```
// ASYNC/AWAIT
async function doWork() {
    const data = await asyncAction();
    console.log(data);
}

doWork();
```

Async/await - syntax with error handling

```
// PROMISE
function doWork() {
    asyncAction()
        .then(data => {
            console.log(data);
        })
        .catch(error => {
            console.log(error);
        });
}

doWork();
```

```
// ASYNC/AWAIT
async function doWork() {
    try {
        const data = await asyncAction();
        console.log(`message = ${data}`);
    } catch (error) {
        console.log(`message = ${error}`);
    }
}

doWork();
```

```
const retrieveAccessToken = url => {
    return new Promise(resolve => {
        post("https://accounts.spotify.com/api/token", {}, ur
lencode({
            grant_type: 'authorization_code',
            code: getParam(url, 'code’),
            redirect_uri: "https://lukaszuk.net/sfy.html",
            client_id: ":)",
            client_secret: ":)",
        }), response => {
            resolve(response);
        });
    })
};

const retrieveUserInfo = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    return new Promise(resolve => {
        get("https://api.spotify.com/v1/me", {
            Authorization: tokenType + ' ' + accessToken
        }, null, response => {
            response['token_type'] = tokenType
            response['access_token'] = accessToken;
            return resolve(response);
        });
    });
};

const createAPlaylist = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    var userId = response.id;
    return new Promise(resolve => {
        post("https://api.spotify.com/v1/users/" + userId + "
/playlists", {
            Authorization: tokenType + ' ' + accessToken,
            "Content-type": "application/json"
        }, JSON.stringify({
            name: localStorage.playlistTitle
        }), response => {
            response['token_type'] = tokenType
            response['access_token'] = accessToken;
            response['userId'] = userId;
            return resolve(response);
        });
    });
};

const searchASong = response => {
    return new Promise(resolve => {
        get("https://api.spotify.com/v1/search", {
            Authorization: response.token_type + ' ' + respon
se.access_token
        }, buildSearchQuery(response.song), responseFromSearc
h => {
            resolve(responseFromSearch.tracks.items[0]);
        });
    });
};


const getAllSongsInfo = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    var playlistId = response.id;
    var userId = response.userId;
    var songs = JSON.parse(localStorage.songs);
    var allSearchPromises = [];
    for (key in songs) {
        response['song'] = songs[key];
        allSearchPromises.push(searchASong(response));
    }
    return Promise.all(allSearchPromises).then(function (resp
onse) {
        response['token_type'] = tokenType;
        response['access_token'] = accessToken;
        response['playlistId'] = playlistId;
        response['userId'] = userId;
        return response;
    });
};

const prepareToaddAllSongsToPlaylist = response => {
    var songs = [];
    for (key in response) {
        if (isNumeric(key)) {
            songs.push(response[key].uri);
        }
    }
    return new Promise(resolve => {
        response['songs'] = songs;
        resolve(response);
    });
};

const addAllSongsToPlaylist = response => {
    var tokenType = response.token_type;
    var accessToken = response.access_token;
    var playlistId = response.playlistId;
    var userId = response.userId;
    var songs = response.songs;
    return new Promise(resolve => {
        post("https://api.spotify.com/v1/users/" + userId + "
/playlists/" + playlistId + "/tracks", {
            Authorization: tokenType + ' ' + accessToken,
            "Content-type": "application/json"
        }, JSON.stringify({
            uris: songs
        }), function (response) {
            resolve(response);
        });
    });
};

function isNumeric(n) {
    return !isNaN(parseFloat(n)) && isFinite(n);
}

function buildSearchQuery(song) {
    return "q=" + song.title + "%20album:" + song.album +
        "%20artist:" + song.artist +  "&type=track";
}

const beginToAddSongsToPlaylist = async () => {
    let response = await retrieveAccessToken(tab.url);
    response = await retrieveUserInfo(response);
    response = await createAPlaylist(response);
    response = await getAllSongsInfo(response);
    response = await prepareToaddAllSongsToPlaylist(response)
;
    response = await addAllSongsToPlaylist(response);
};

beginToAddSongsToPlaylist();
```

Promises vs async/await

```
//PROMISE
retrieveAccessToken(tab.url)
    .then(retrieveUserInfo)
    .then(createAPlaylist)
    .then(getAllSongsInfo)
    .then(prepareToaddAllSongsToPlaylist)
    .then(addAllSongsToPlaylist)
    .catch(error => {
        progress.innerHTML += "[WARNING] " + error + "<br>";
    });
```

```
//ASYNC-AWAIT
const beginToAddSongsToPlaylist = async () => {
    let response = await retrieveAccessToken(tab.url);
    response = await retrieveUserInfo(response);
    response = await createAPlaylist(response);
    response = await getAllSongsInfo(response);
    response = await prepareToaddAllSongsToPlaylist(response);
    response = await addAllSongsToPlaylist(response);
};

beginToAddSongsToPlaylist();
```

Promise.all vs async/await

```
var p1 = new Promise((resolve, reject) => { … });
var p2 = new Promise((resolve, reject) => { … });
var p3 = new Promise((resolve, reject) => { … });

Promise.all([p1, p2, p3]).then(values => {
  console.log(values);
  // result of all promises […, …, …]
});
```

```
:(
```

Promise.any vs async/await

```
var p1 = new Promise((resolve, reject) => { … });
var p2 = new Promise((resolve, reject) => { … });
var p3 = new Promise((resolve, reject) => { … });

Promise.any([p1, p2, p3]).then((value) => {
    console.log(value);
    // result which fulfils first
})
```

```
:(
```

Promise.race vs async/await

```
var p1 = new Promise((resolve, reject) => { … });
var p2 = new Promise((resolve, reject) => { … });
var p3 = new Promise((resolve, reject) => { … });

Promise.race([p1, p2, p3]).then((value) => {
    console.log(value);
    // result which fulfils or rejects first
})
```

```
:(
```

Async/Await

Pros of async/await approach:

- similar pattern is available in other languages: C#, F#, Python, Rust, Scala
- concise and clean
- error handling using common javascript approach
- more accessible intermediate values
- easier debugging

Cons of async/await approach:

- looks less like JavaScript
- slightly less functionality

Async/Await summary

Async/await is really syntactic sugar for promises

because it still uses promises under the hood.

It’s not either/or

You can use both promise chains and async await,

and there’s nothing wrong with that :)