CPS

In traditional programming when we run some async operation, the CPU will wait idly for the result.
To eliminate that idly waiting we can create a separate thread for each async operation. But that method increases app complexity because we need to synchronize threads.
In JS there is another method that simplifies async operations.

CPS stands for continuation-passing style.
It is a technique that allows us to define what the program should do after some async operation is done without blocking the program's main thread.

There are three main tools in JS that allow as to use CPS:

- Callbacks
- Promises,
- Async/Await

Callbacks

Callbacks are the most primitive type of CPS. Async functions, that support callbacks receive a callback function as the last argument by convention.

```
asyncOpertaion(data, (err, result) => {
    if (err) {
        return handleError(err);
    }

    handleResult(result);
})
```

Despite that callbacks are very simple, they are very hard to support when we have multiple sequential asynchronous calls.

As you can see with each callback the indentation level is increased and it became hard to find where the operations start and where to end. In JS we call that Callback Hell.

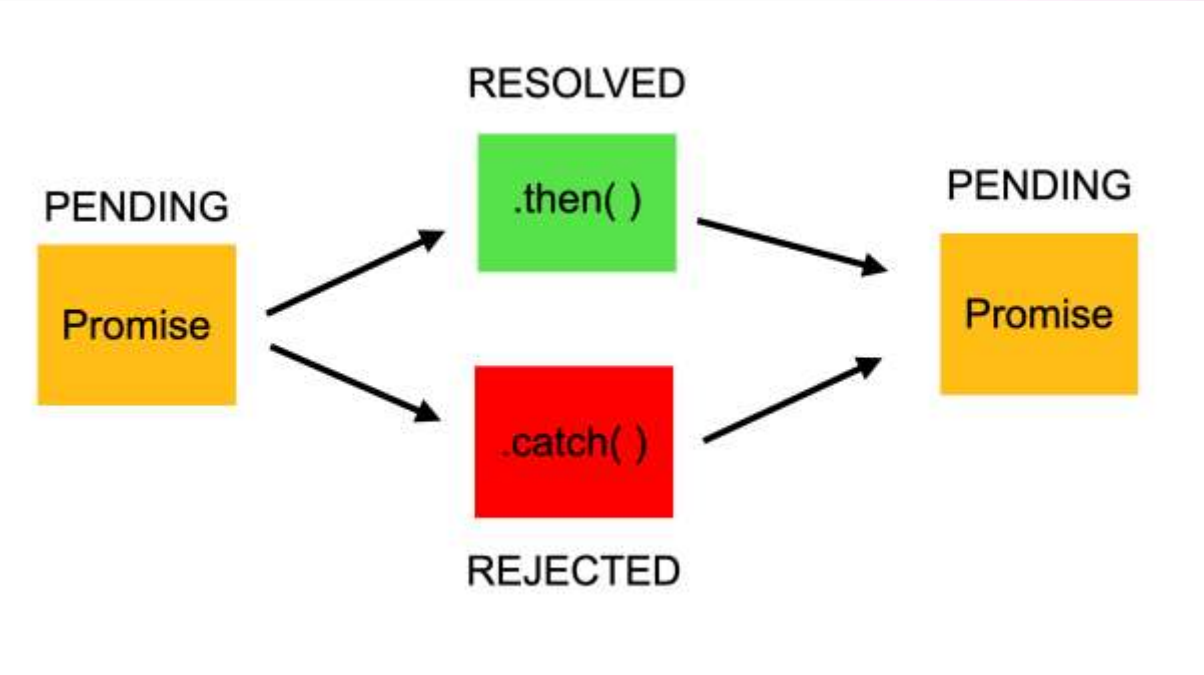
```
asyncOpertaion1(data, (err1, result1) => {
    // err1 handling ----
    asyncOpertaion2(result1, (err2, result2) => {
        // err2 handling ----
        asyncOpertaion3(result2, (err3, result3) => {
            // err3 handling ----
            asyncOpertaion4(result3, (err4, result4) => {
                // err4 handling ----
            })
        })
    })
});
```

Promises

Promises are another type of CPS that hasn't a Callback Hell issue. A promise is an object returned from an async function.

```
const promise = asyncOpertaion(data);
```

The promise object can be in one of three states: **pending**, **resolved**, **rejected**.



At first, the promise has a **pending** state. then when the async operation finishes, it will change its state to **resolved** in case of success or **rejected** in case of failure.

sourcemind

Promise

We can add listeners to promise state changes. To add listeners for the resolved state we use then and use catch for rejected.

```
promise
    .then((result) => {
        console.log(result);
    })
    .catch((err) => {
        console.log(err)
    });
```

Promise chaining can allow us to avoid callback hell.

```
asyncOpertaion1(data)
    .then(result1 => {
        return asyncOpertaion2(result1)
    })
    .then(result2 => {
        return asyncOpertaion3(result2)
    })
    .then(result3 => {
        return asyncOpertaion4(result3)
    })
    .then(result4 => { /* .   */})
    .catch(err => {
        handleError(err)
    });
```

sourcemind

Promisify

Suppose we have an old library with callback based functions and we want to use that functions in promise way. To convert callback based async function to promise we can wrap that function into promise constructor.

Promise constructor take one function with to parameters: **resolve**, **reject**. They both are functions. If we call resolve, the promise will be resolvedand if we call reject the promise will be rejected.

```
new Promise((resolve, reject) => {

});
```

Now let's promisify callback based async function. We will call reject when there is an error in the callback, otherwise, we will call resolve.

```
const promisifiedFunc = (data) => {
    return new Promise((resolve, reject) => {
        asyncFun(data, (err, result) => {
            if( err) {
                return reject(err);
            }

            resolve(result);
        })
    })
}

promisifiedFunc(data)
    .then(() => {})
    .catch(() => {})
```

sourcemind

Async / Await

Async/await is a more concise and elegant way of working with promises. They let us write async code like ordinary synchronous code. Let's write code from previous example by using async/await.

```
async function myFun(data) {
    try {
        const result1 = await asyncOpertaion1(data);
        const result2 = await asyncOpertaion2(result1);
        const result3 = await asyncOpertaion3(result2);
        const result4 = await asyncOpertaion4(result3);
        // ---- handle result4
    } catch (err) {
        handleError(err)
    }
}

myFun(data);
```

Here we put await keywords before functions that return a promise. The await keyword will wait until the promise is resolved or rejected. If the Promise is resolved, then await will return the promise value. If the promise is rejected await will throw an exception. We can use await only in functions that are marked with async keyword.

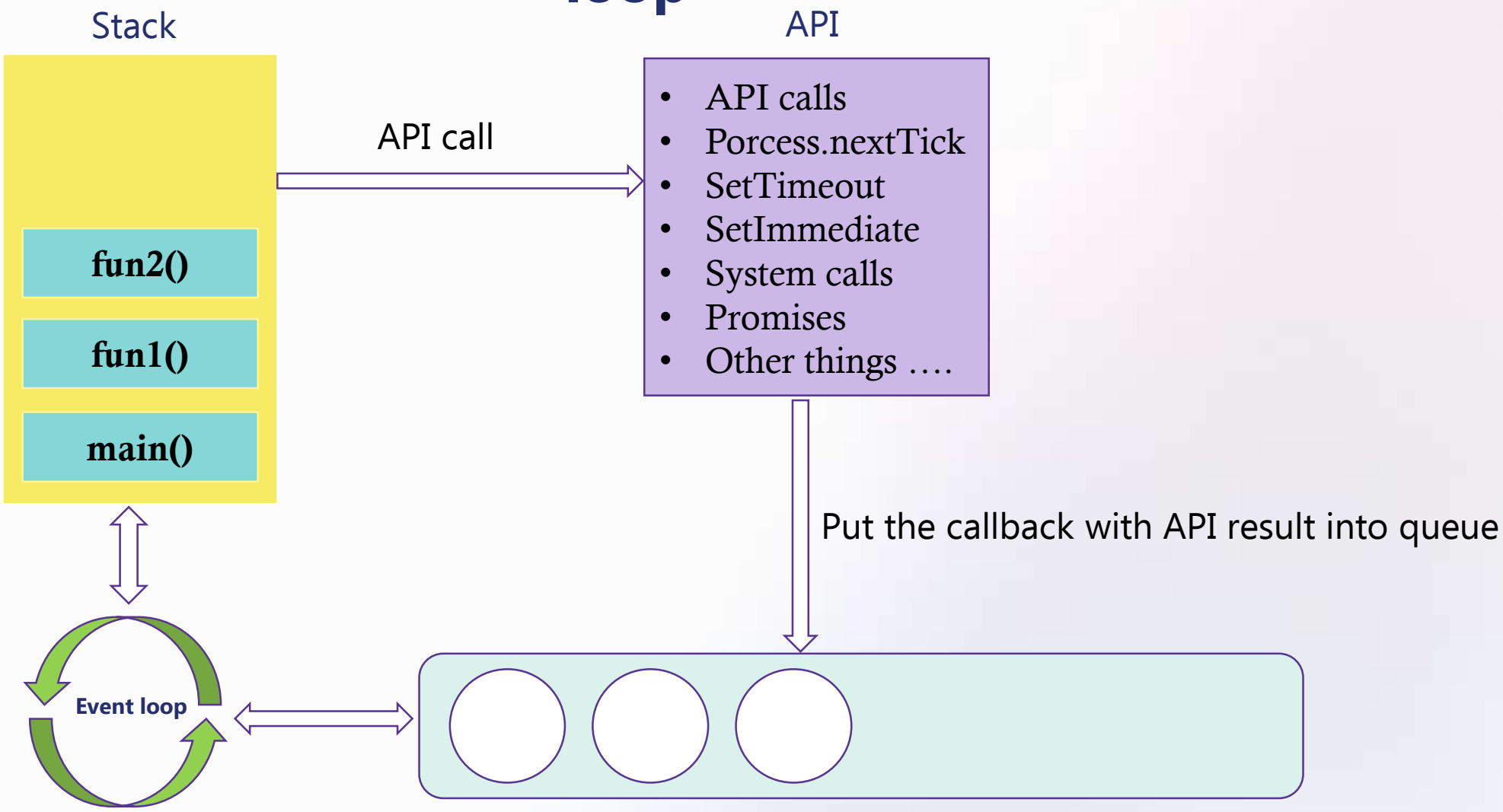
Event Emitter

Node js has a built-in events module that exports the EventEmitter constructor. The instance of EventEmitter can emit events and we can add listeners to that events.

```
const EventEmmiter = require('events');

const em = new EventEmmiter();

em.on('my event', data => {
    console.log(data);
});

em.emit('my event', 'hello');
```

To add a listener we use **on** function, which receives 2 parameters: event name and callback, that receives event data.
To emit an event we use **emit** function, which receives 2 parameters: event name and event data.

Event loop



GLOBALS

- Like a browser, there is a global object in Node JS. Instead of a window, we call it global. We can access global object's properties without prefix. The well-known globals like setTimeout, setImmediate, process, Buffer, Nuber, String, and others are global object properties.

```
console.log(global.Number === Number);          // true
console.log(global.process === process);        // true
console.log(global.Buffer === Buffer);          // true
console.log(global.setTimeout === setTimeout); // true
```

Process

The process object provides information about, and control over, the current Node.js process.

We can add listeners to process events like exit, beforeExit, uncaughtException, and others.

```
process.on('exit', () => {
    console.log('exited');
})
```

process alls provide functions that we can use to get and set process related info: getgid(), getuid(), getgroups(), setgid(), setuid(), setgroups().
Also, there are properties that contain process information: pid, ppid, platform, env, execArgv, execPath, and others.
The most used property of the process is env, which we use to get environment variables.

```
process.env.PRIVATE_KEY
```

File System

To work with file system we can use fs or fs/promise. The fs module provide synchronous and async callback based API. The fs/promise module provide async promise based API.
Lets first consider synchronous API.

fs.readdirSync will return the content of directory specified by first parameter. here __dirname is a variable that returns the directory name of the current module. There is also __filename variable that returns the full name of the current module.
fs.mkdirSync and fs.rmdirSync functions accordingly create and delete directories.

```
const fs = require('fs');
const files = fs.readdirSync(__dirname);
```

fs.readSync function read the file specified by the first parameter and returns the Buffer object. Buffer is a special type that represents a fixed-length sequence of bytes. To convert the buffer to a string we can use the toString function.

```
const file = fs.readFileSync(__filename);
console.log(file.toString());
```

fs.writeFileSync function used to write data into a file. By default, it will create a file, if it does not exist. We can configure that and other things by options parameter.

```
fs.writeFileSync(`${__dirname}/testfile`, 'Hello');
```

fs/promises

fs/promises module is very similar to the fs module, but instead, it works with promises.
Let's rewrite previous examples by using the fs/promises module.

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

fs.readdir(__dirname)
    .then(files => { });

fs.readFile(__filename)
    .then(file => { });

fs.writeFile(`${__dirname}/testFile`, "hello")
    .then(() => { });
```