Callbacks & Promises in JS

Presentation by

Hung Nguyen Huy

Contents

Introduction

JavaScript is a **single-threaded** and **asynchronous** programming language.

one thread == one call stack == one thing at a time

Asynchronous vs Synchronous

- Synchronous Processing

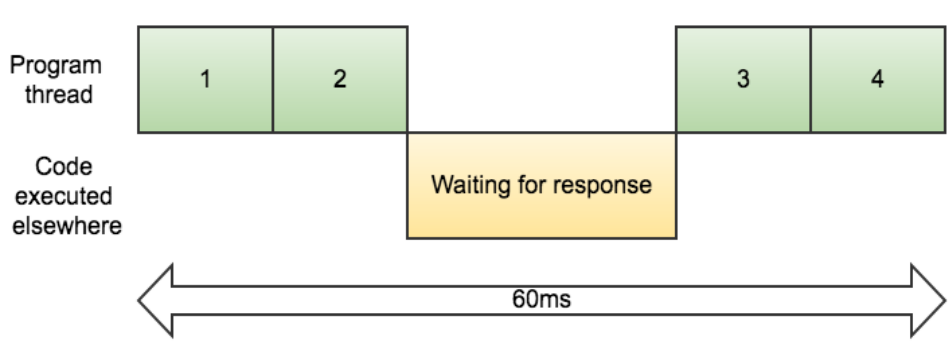
  In ***synchronous*** programs, if you have two lines of code (L1 followed by L2), then L2 **can not** begin running **until L1 has finished executing**.

- Asynchronous Processing

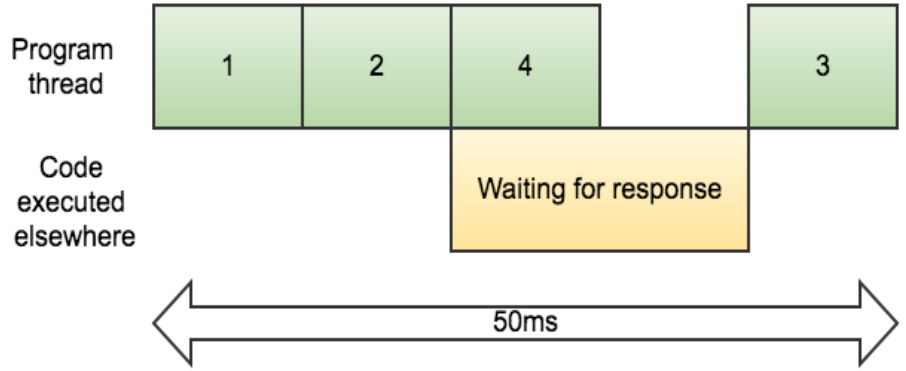
  In ***asynchronous*** programs, you can have two lines of code (L1 followed by L2), where L1 schedules some task to be run **in the future**, but L2 runs **before** that task completes.

Asynchronous vs Synchronous

Synchronous



Asynchronous



Callbacks

A callback function is **a function passed into another function as an argument**, which is then invoked inside the outer function to complete some kind of routine or action.

Async callbacks & Callback hell

- Asynchronous callback

  Callbacks are often used to continue code execution after an asynchronous operation has completed - these are called **asynchronous callbacks.**

- Callback hell

  Chain of asynchronous operations

  => Hard to read & debug code.

Promises arrive in JavaScript!

Getting rid of callback hell!

What is Promise?

Promise is an object which represents the eventual completion or failure of an **asynchronous operation**, and its resulting value.

Constructing a Promise

- We use *new Promise* to construct the promise
- We give the constructor a **executor** function which does the actual work.
- Executor function is passed with 2 arguments: **resolve** and **reject** functions.
- Resolve function fulfills the promise and reject function rejects the promise.

```
new Promise(
    /* executor */
    function(resolve, reject) {
        ...
    }
);
```

Promise states

A Promise is in one of these 3 states:

- ***pending***: initial state, neither fulfilled nor rejected.
- ***fulfilled***: meaning that the operation completed successfully.
- ***rejected***: meaning that the operation failed.

Using the Promise object

- A pending promise can either be **fulfilled with a value**, or **rejected with a reason (error).**
- When either of these options happens, the associated handlers queued up by a promise's ***then*** method are called.

then() and catch()

- then()

  Takes up to two arguments: callback functions for the success and failure cases of the Promise. It **returns a Promise**.

```
p.then(onFulfilled[, onRejected]);


p.then(function(value) {
  // fulfillment
}, function(reason) {
  // rejection
});
```

- catch()

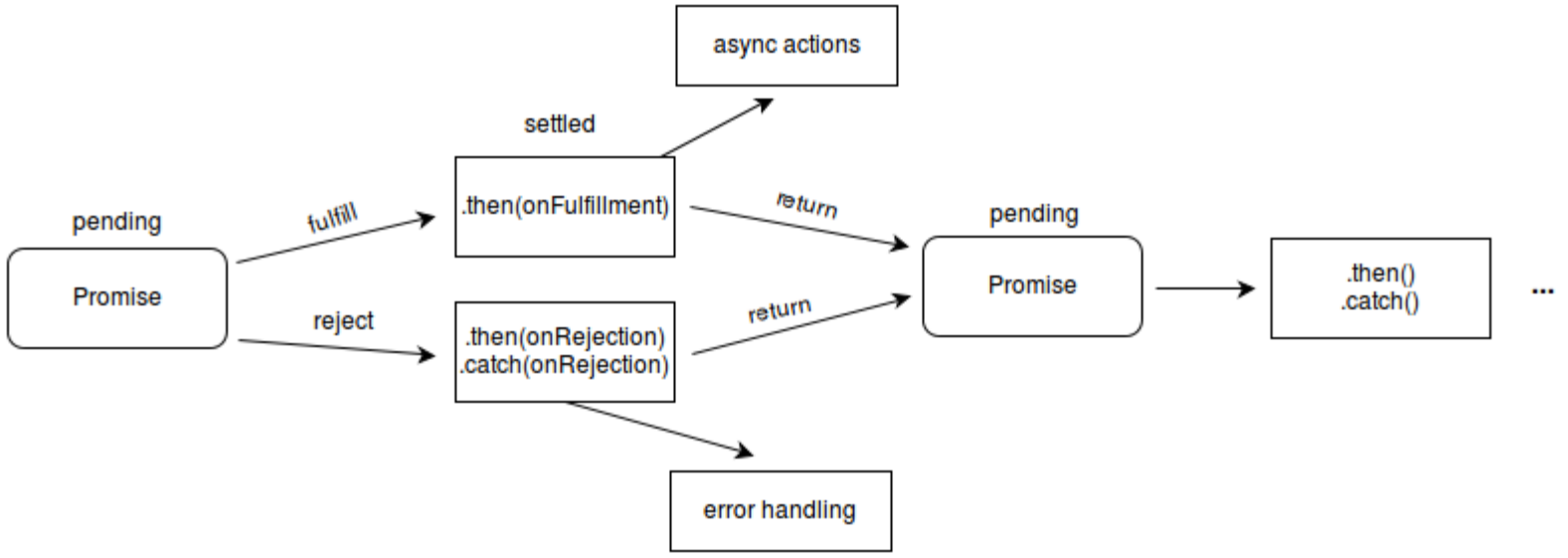
  Take only one argument to deal with rejected cases only. It **returns a Promise**.

```
p.catch(onRejected);


p.catch(function(reason) {
   // rejection
});
```

Promises chaining & transformation

As the ***then()*** and ***catch()*** methods return promises, they can be chained.



Error handling

- with then()

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}, function(error) {
  console.log("Failed!", error);
})
```

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}).then(undefined, function(error) {
  console.log("Failed!", error);
})
```

- with catch()

```
get('story.json').then(function(response) {
  console.log("Success!", response);
}).catch(function(error) {
  console.log("Failed!", error);
})
```

Error handling

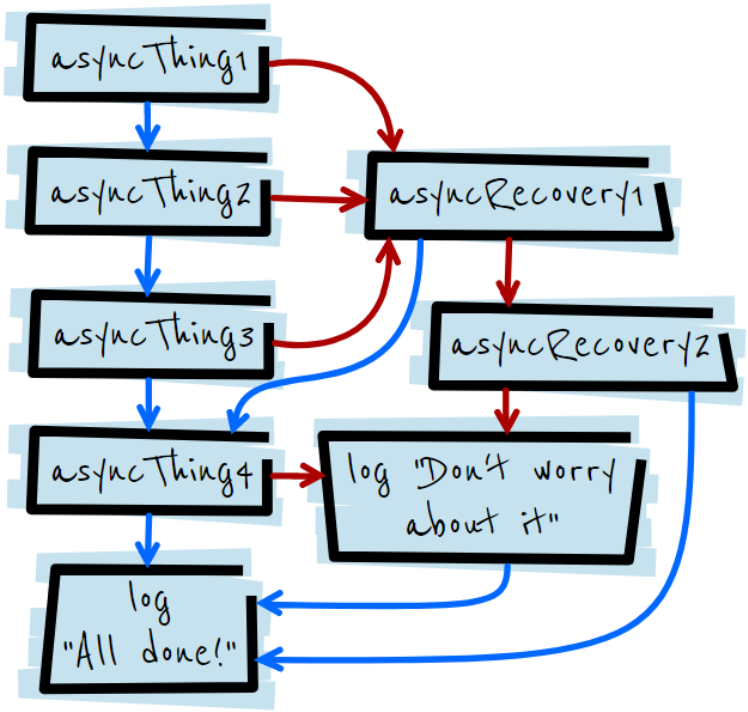
- Different between *then(undefined, func)* and *catch(func)*?

  Promise rejections skip forward to the next then() with a rejection callback (or catch()).

  With then(func1, func2), func1 or func2 will be called, never both. But with then(func1).catch(func2), both will be called if func1 rejects, as they're separate steps in the chain.

Error handling

```
asyncThing1().then(function() {
  return asyncThing2();
}).then(function() {
  return asyncThing3();
}).catch(function(err) {
  return asyncRecovery1();
}).then(function() {
  return asyncThing4();
}, function(err) {
  return asyncRecovery2();
}).catch(function(err) {
  console.log("Don't worry about it");
}).then(function() {
  console.log("All done!");
})
```



Promise.all()

Returns a promise that either fulfills when all of the promises in the iterable argument have fulfilled or rejects as soon as one of the promises in the iterable argument rejects.

Promise.race()

Returns a promise that fulfills or rejects as soon as one of the promises in the iterable fulfills or rejects, with the value or reason from that promise.

SUMMARY