Async JS Exercises

Agenda

This is mostly a revision class. We have used callbacks and promises already in levelZero. In this class, we will understand some terminologies associated with it. There aren't new concepts.

We will also see async/await but that's just syntax over promises.

how to do this ?

```
watchTheWholeSessionNow()
.then(tryAgainYourself)
.then(discussWithYourTeam)
.then(() => {
if (conceptsStillNotClear === true) {
   askInDoubtClearingSessions();
}})
```

COPY

section 1: Callbacks

Where have you seen callbacks ?

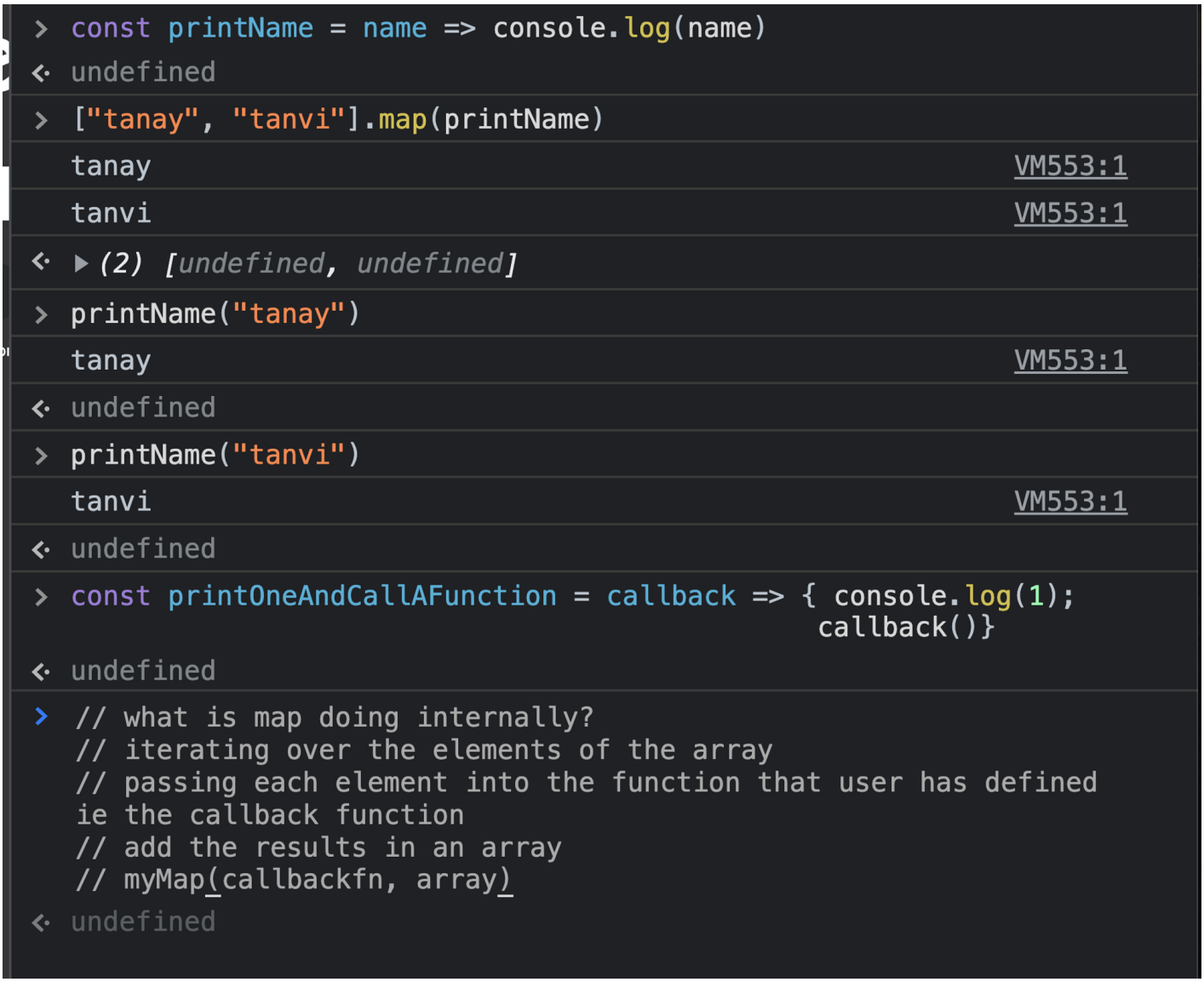
Challenge

We have seen callbacks in levelZero. Can you tell us where?

understanding

We have used callbacks all along in vanillaJS and in React too. However, it's time to understand the other side of this.

write your own function which takes a callback



challenge

Write a function `strLength()` which takes

1. Your name
2. A function that it will call with the length of your name

Now, call this function with the two parameters.

1. Your name. This is simple.
2. A function. This function will get the length of the string. Use that length to print a message: `OMG! my name is X char long!`

```
> const strLength = (name, cb) => cb(name.length)
< undefined
> strLength("tanay", (lengthOfName) => console.log(`OMG! my name is
  ${lengthOfName} char long!`))
  OMG! my name is 5 char long!                                VM2729:1
< undefined
> const printNameMessage = (lengthOfName) => console.log(`OMG! my
  name is ${lengthOfName} char long!`);
< undefined
> strLength("tanay", printNameMessage)
  OMG! my name is 5 char long!                                VM2851:1
< undefined
>
```

understanding

When you're designing your API. A callback is nothing but a function that the user of your API will give you. In return, you give your user a contract saying that you'll call her function with X data.

Understand that this data can be calculated *synchronously or in an asynchronous* manner. It doesn't matter as long as you give that data.

ex3: function which takes two callbacks

challenge

Write a function `willThanosKillMe()` . This function will take three parameters

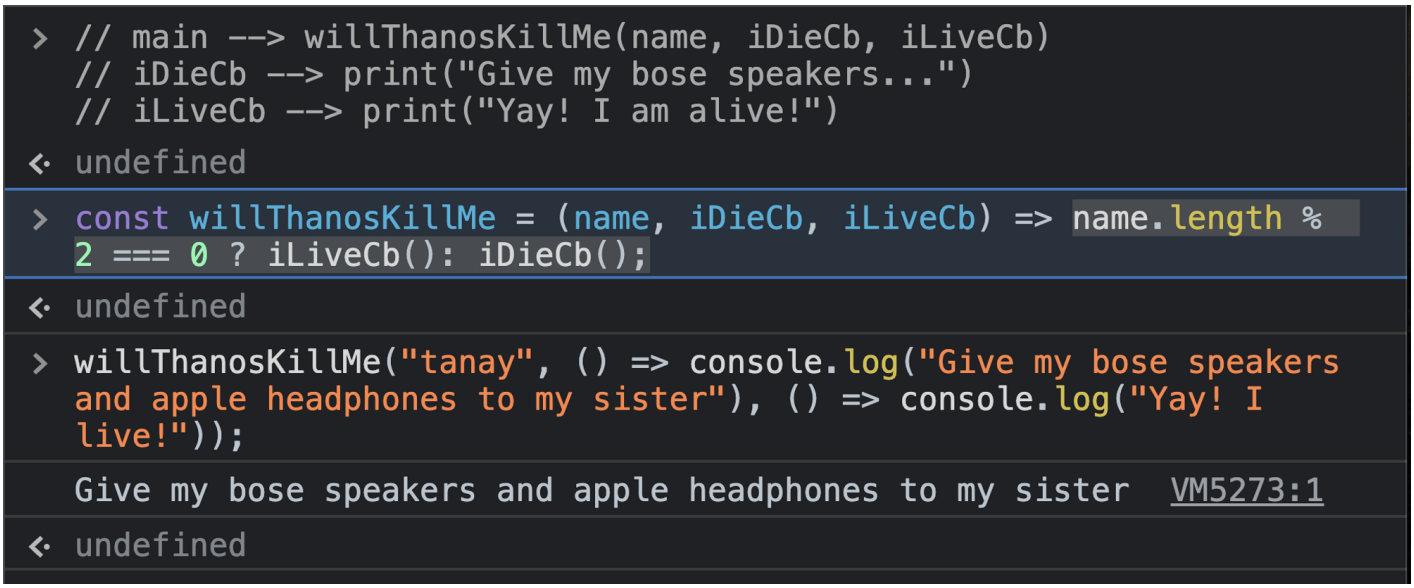
1. Your name
2. Function to call if Thanos doesn't kill you.
3. Function to call if Thanos does kill you.

This function will call success callback if your name has even characters. You won't die.

But if your name has odd number of characters then you're going to die. Sorry! :(

Now, call this function with the given parameters. The success function should console a happy message: `Yay! I am alive!` and the failure function should console your will: `Give my bose speakers and apple headphones to my sister`

P.S. Sorry for the grim example. ☹️



understanding

Say in your function you were calling a server on Titan to see whether Thanos is going to kill your or not. Obviously that network call would take time. Making the entire program wait for one network call wouldn't look good na! Hence, the callbacks.

ex4: use setTimeOut()

challenge

- Learn how to use `setTimeout()`
- Now, write a function that takes a message and a delay and print that message after the delay.

```
> setTimeout(() => console.log("Tanay"), 3000)
< 2
  Tanay                                                    VM6255:1
> // Journey of setTimeout
  // First goes to callstack
  // callstack sees setTimeout, I don't know this, it gives it to
  webAPIs
  // setTimeout --> fn, timer. Let me wait till 3 seconds //
  guarantee of 3 seconds waiting here
  // job queue aka task queue
  // event loop will see! Wow! there's a new task on job queue
  // event loop will check if callstack is empty
  // if it's empty, fn will be executed
  // if it's busy, fn will wait job queue // guarantee of timer
  expires in this case
< undefined
```

understanding

`setTimeout()` places the callback function on the event queue after the given time.

remember

- It doesn't guarantee time
- `setTimeout(fn, 0)` is used to do calculation when callstack is empty. Therefore not blocking the render.

ex5: predict outputs

1.

```
setTimeout(() => console.log("A"), 0);
setTimeout(() => console.log("C"), 0);
setTimeout(() => console.log("B"), 0);
```

COPY

2.

```
setTimeout(() => console.log("A"), 1000);
setTimeout(() => console.log("B"), 400);
setTimeout(() => console.log("C"), 1300);
```

COPY

```
> // Question 2
  // 0ms
  // Callstack: A
  // WebAPIs: Send A, 1000ms
  // Job Queue?

  // Callstack: B
  // WebAPIs: Send B, 400ms
  // Job Queue? Nope

  // Callstack: C
  // WebAPIs: Send C, 1300ms
  // Job Queue? Nope

  // 399ms
  // Callstack: nothing
  // Job Queue: nothing


  // Three timers are running in the webAPIs
  // 400ms
  // B is done
  // Job Queue: B
  // Callstack: B --> executed


  // 999ms
  // Callstack: nothing
  // Job Queue: nothing

  // 1000ms
  // A is done
  // Job Queue: A
  // Callstack: A --> executed
<· undefined
```

3.

```
console.log("A");
setTimeout(() => console.log("B"), 0);
console.log("C");
```

COPY

```
> // Question 3
  // 0ms // line 1
  // Callstack --> console.log("A")
  // Console --> Print A
  // Job Queue --> No
  // WebAPIs --> No

  // 0ms // line 2
  // Callstack --> setTimeout
  // Console --> A
  // Job Queue --> No
  // WebAPIs --> fn, 0

  // 0ms // line 3
  // Callstack --> console.log("C")
  // console --> A C
  // Job Queue --> fn ==> console.log("B")
  // WebAPIs --> nothing

  // 0ms
  // Callstack --> console.log("B")
  // Job Queue --> No
<- undefined
```

```
> // WebAPIs --> No
  // Console --> A C B
<- undefined
```

homework

If you didn't predict this right, discuss that particular number with your group after the class. This means your understanding from last class is not solid yet.

promises

ex6: print data on success

A fake Promise for class exercises.

```
function fakeFetch(msg, shouldReject) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (shouldReject) {
        reject(`error from server: ${msg}`);
      }
      resolve(`from server: ${msg}`);
    }, 3000);
  });
}
```

```
// Internally inside promise
function Promise(cb) {
  let state = "pending"
  function resolve(msg) { state = "fulfilled" }
  function reject(msg) { state = "reject" }

  cb(resolve, reject)

}
```

COPY

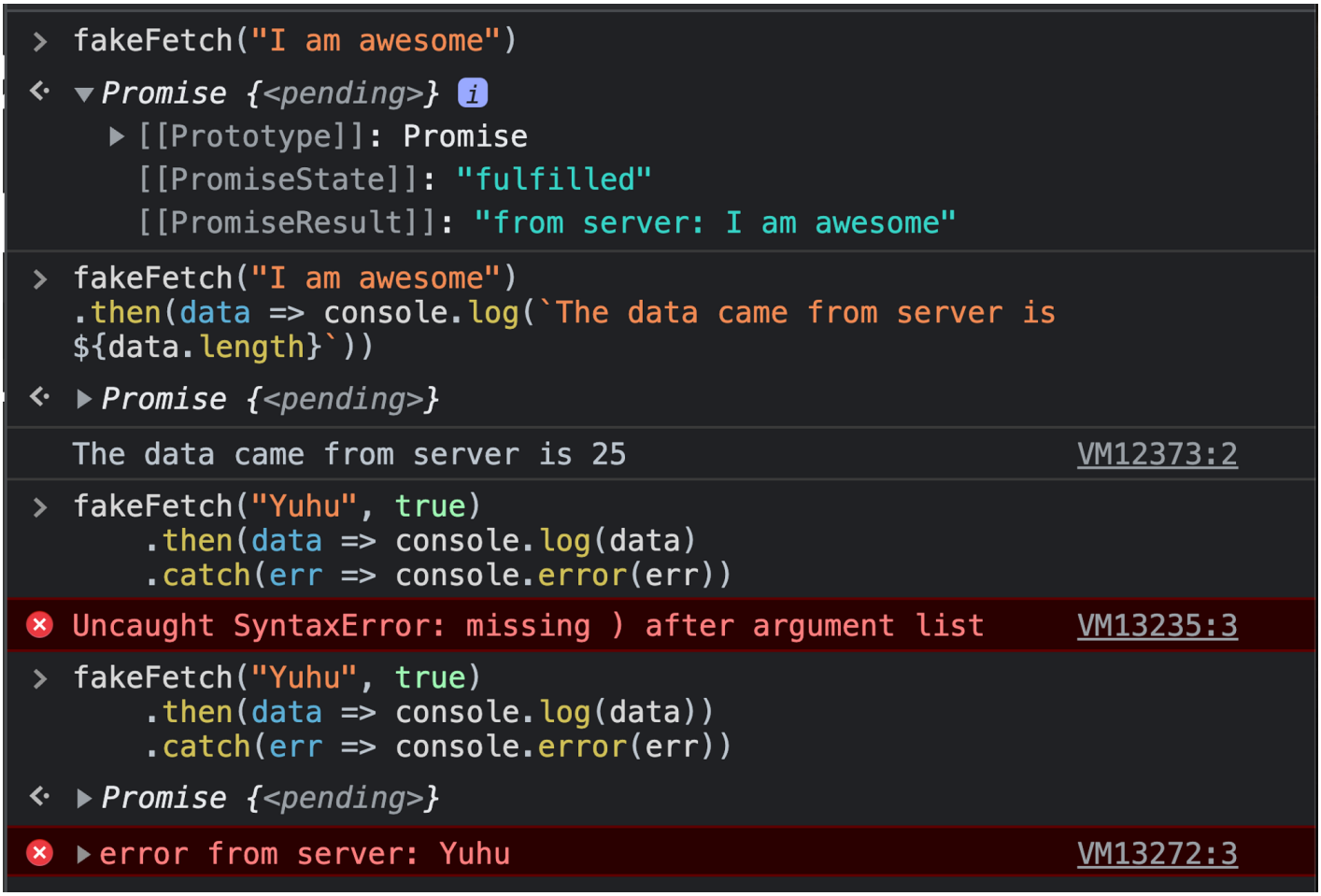
challenge

use the `fakeFetch()` to get data and show on success.

ex7: print data on success, print error on failure

challenge

Call `fakeFetch(msg, true)` and it will reject the promise. Handle the error with the error handler. Show a message using `console.error` for errors.



ex8: chaining

challenge

Create a function `getServerResponseLength(msg)` This function will use `fakeFetch()` internally with the message and return the length of the response received by the server.

Note: Instead of returning the response from the server this should return the length.

Hint: It will return in a promise.

understanding

Whatever you return from `.then()` also becomes a promise. And this is how the chain propagates.

```
> const getServerResponseLength = msg => {
      fakeFetch(msg)
      .then(response => response.length)
  }
< undefined
> getServerResponseLength("Yuhu")
  .then(data => console.log(data))
✕ ▶Uncaught TypeError: Cannot read properties of          VM14432:2
  undefined (reading 'then')
      at <anonymous>:2:1
> const getServerResponseLength = msg => fakeFetch(msg)
                                         .then(response =>
  response.length)
< undefined
> getServerResponseLength("Yuhu")
  .then(data => console.log(data))
< ▶Promise {<pending>}
  17                                                      VM14563:2
```

async-await

Nothing but promises which look better.

Two best practices to keep in mind:

1. Use async-await as much as possible.
2. Always take care of the error handling as well.

```
async function printDataFromServer() {
  const serverData = await anyPromiseWhichWillReturnData()
  console.log(serverData);
}

// notice that function need an `async` keyword.

// Doing this in es6 arrow function would be

const printDataFromServer = async () => {
        try {
                const serverData = await anyPromiseWhichWillReturnData();
          console.log(serverData);
        } catch (err) {
         console.error(err)
        }
}

/**
Note: In arrow the async keyword is used before the ().
While in normal functions, it is used before the `function` keyword itself.
```

COPY

Remind me to give practice question sets for try/catch as well. Even outside promises.

Read exception handling in JavaScript. Maybe write a blog about it. Don’t go deep into it.

ex9: use async-await with fakeFetch

challenge

Given the syntax above. Call `fakeFetch()` with some msg and use await to get the data and then print it.