MORE Javascript!

The Event Loop:

- Imagine that every single tab in your web browser is running on a constant loop. Javascript code is entirely single threaded, that is to say, there are no other threads or processes that get kicked off. While this does seemingly limit the abilities of your program, it does simplify your code (i.e. you don't need to worry about listening for other threads). All you need to think about is how to create code that does not block (e.g. no "event listener loops" that spin forever).
- Expensive calls in javascript (such as network calls) that could block the event loop are non-blocking.

The Event Loop:

- The event loop looks to see if there is anything in the call stack that can be run. Regularly speaking, think of this as a stack trace.

```
1
2  const sayHi = () => console.log("hi");
3
4  const sayGoodbye = () => console.log("goodbye");
5
6
7
8  const interact = () => {
9
10   sayHi();
```

```
  sayGoodbye();

};


interact();

```

outputs:

```
hi

goodbye

```

The Event Loop:

But what happens if there's a blocking element?

```
const sayHi2 = () => console.log("hi");

const saySomethingEngaging = () => console.log("I live in a giant shoe");

const sayGoodbye2 = () => console.log("goodbye");


const interact2 = () => {

```

```
  sayHi2();

  setTimeout(saySomethingEngaging, 500);

  sayGoodbye2();

};



interact2();

```

- Once the timer is finished, saySomethingEngaging is placed into the message queue (also where user inputs go) to be retrieved by the loop. Priority is given to the call stack (i.e. everything that's not waiting in the message queue).
- output:

```

> interact2()

hi

goodbye



> I live in a giant shoe

```

Asynchronous Programming, Promises, and Async/Await

Because javascript is single threaded, it's synchronous by default. So how do we get around that? With asynchronous programming!

Callbacks:

Callbacks are functions that are passed to other functions that act as listeners.

```
1
2  document.querySelector("button").addEventListener("click", () => {
3
4    console.log("Call me back!");
5
6  });
7
```

- To account for error handling, the first parameter of every callback is typically error, with the second parameter being the desired input (we'll see examples of this in a couple slides)
- Callbacks can be very helpful, however, they can very easily result in the pyramid of doom! (aka callback hell)

Promises

- Promises are one way to get around the pyramid of doom
- Ultimately, whena promise is called, it gets placed into its pending state (neither fulfilled or rejected), while the remaining, nonblocking code executes.
    - Promises resolve to either:
        - fulfilled: the operation was successful

- rejected: the operation failed

Promises

- When the promise is resolved, the respective handlers are called:
  - .then()
  - .catch()

```
1
2  const successfulPromise = new Promise((resolve, reject) => {
3
4    setTimeout(() => {
5
6      resolve("Huzzah!");
7
8    }, 500);
9
10 });
11
12
13
14 console.log("Successful Promise: ", successfulPromise);
15
16
17
18 successfulPromise.then(whateverWasResolved => {
19
20   console.log("Neat! we resolved " + whateverWasResolved);
21
22 });
23
```

Promises

While the previous code is fine, it is still somewhat clunky. We can further condense it by chaining our .methods (and getting ride of that middle log):

```
const successfulPromise2 = new Promise((resolve, reject) => {

  setTimeout(() => {

    resolve("Huzzah!");

  }, 500);

}).then(whateverWasResolved => {

  console.log("Neat! we resolved " + whateverWasResolved);

});

```

Promises

- The same goes for rejected promises:

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

  setTimeout(() => {

    reject("Not Huzzah!");

  }, 500);

```

```
}).catch(err => {

  console.log("Neat! we got rejected! " + err);

});

```

Promises

```

let whoKnowsWhatKindOfPromise6 = new Promise((resolve, reject) => {

  setTimeout(() => {

    Math.random() > 0.5 ? resolve("Huzzah!") : reject("Nuhzah");

  }, 500);

})

  .then(whateverWasResolved => {

    console.log(whateverWasResolved);

  })

  .catch(err => {

    console.error(err);

  });

```

Async / Await

Built on top of promises, async/awati makes the syntax a bit more readable: When calling an asynchronous function you need to follow the syntax of:

```
const functionName = async parameter => {

  await someAsynchronousFunction();

};

```

Async / Await

In real code, async await would look something akin to:

```
const someAsynchronousFunction = () => {

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

    setTimeout(() => {

      resolve("Taking our time!");

    }, 500);

  });

};


const useAsyncAwait = async () => {

```

```
  await someAsynchronousFunction();

};

```

Async / Await

- Now, instead of having a series of callbacks or promise.then's, we can use async await to wait on the responses from asynchronous calls and use them in a single code block:

```

const asyncFunc = async () => {

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

    setTimeout(() => {

      resolve("Taking our time!");

    }, 500);

  });

};



const asyncFunc2 = async () => {

  return 23;

};



const callingAsync = async () => {

  const response = await asyncFunc();
```

```
29
30    const response2 = await asyncFunc2();
31
32
33
34    console.log("response is", response);
35
36    console.log("response2 is", response2);
37
38  };
39
```

Functional Programming Paradigms:

There are a number of array manipulation methods to be familiar with as we move forward:

- map
- reduce
- filter
- find

Functional Programming Paradigms:

Some Data:

```
1
2  const officeFolk = [
3
4    { name: "Michael Scott", salary: 70000 },
5
```

```
  { name: "Dwight Schrute", salary: 40000 },

  { name: "Jim Halptert", salary: 40000 },

  { name: "Pam Beesly", salary: 30000 },

  { name: "Ryan Howard", salary: 0 },

  { name: "Andy Bernard", salary: 40000 },

  { name: "Robert California", salary: 1000000 },

  { name: "Stanley Hudson", salary: 50000 },

  { name: "Creed Branton", salary: 40000 },

  { name: "Meredith Palmer", salary: 40000 },

  { name: "Phyllis Lapin", salary: 50000 }

];

```

Functional Programming Paradigms: Map

Instead of writing a loop that can potentially manipulate data in a list, it is better to use a map to read data and return a new list of some given kind based off of the data that you already have. Suppose we would like to get a list only of the names of the characters above:

```
const characterNames = officeFolk.map(officePerson => {
  return officePerson.name;
});


console.log(characterNames);
```

Functional Programming Paradigms: Map

.map takes a function as a parameter, where the parameter for the function is each individual element of the list. We could easily write the function separately such as:

```
const retrieveOfficeCharacterName = officeCharacter => {
  return officeCharacter.name;
};


const characterNameList = officeFolk.map(retrieveOfficeCharacterName);


console.log(characterNameList);
```

Functional Programming Paradigms: Reduce

Reduce is a function that iterates over a list and has an accumulator. Suppose we wanted the total sum of all the salaries of the office characters:

```
1
2  const salarySum = officeFolk.reduce((accumulator, officeCharacter) => {
3
4    return accumulator + officeCharacter.salary;
5
6  }, (accumulator = 0));
7
8
9
10 console.log("Total salary is ", salarySum);
11
```

What we do is start with a given value (accumulator = 0), and then for each item within the list, we execute the supplied function, and update the accumulator.

Functional Programming Paradigms: Filter

When you wish to receive a list from another list that adheres to some specific set of criteria, filter is the best option. To get a list of all office characters who have salaries over 60,000, we'd write an anonymous function that returns a boolean value:

```
const higherSalaries = officeFolk.filter( officeCharacter =>
officeCharacter.salary >== 60000)



console.log(higherSalaries)
```

Functional Programming Paradigms: Find

Find will retrieve a value from the list by passing a function that returns a boolean value (just like the filter function):

```
const noSalary = officeFolk.find(character => character.salary === 0);


console.log(noSalary);
```

Functional Programming Paradigms: Find

As a note, however, find will not return ALL values that match, but only the first value that matches:

```
1
2  const fortyThousandSalary = officeFolk.find(
3
4    character => character.salary === 40000
5
6  );
7
8
9
10 console.log(fortyThousandSalary);
11
```