Learnathon 2.0 - JS Class 4

•••

Asynchronous JavaScript

Kazi Tajnur Islam | Software Engineer (L-2) | Vivasoft Limited

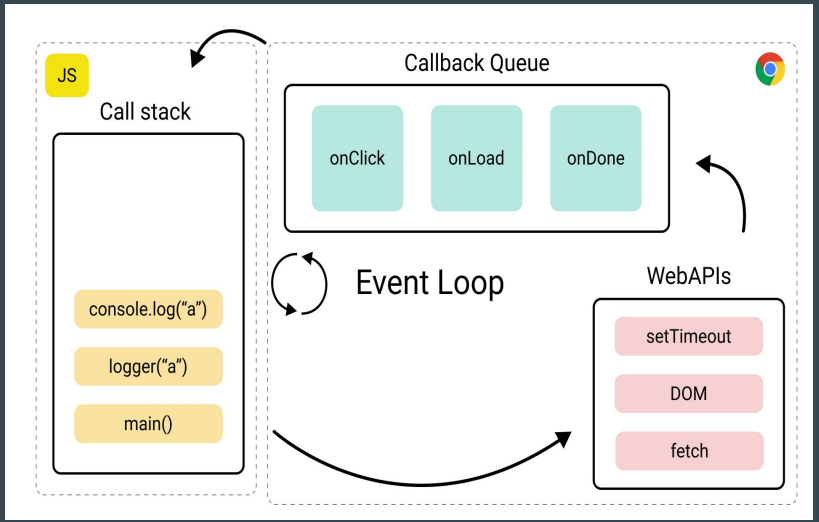
What does this asynchronous term mean?

In JavaScript, the term "asynchronous" refers to a programming paradigm or style of executing code where tasks are not completed in a sequential, step-by-step manner. Instead, asynchronous operations allow certain tasks to be executed independently, without blocking the main program's execution.

Event Loop

Event loop allows JavaScript to handle asynchronous operations efficiently without blocking the execution of other code. It plays a crucial role in managing the execution of events, callbacks, and promises in a non-blocking manner.

The event loop is a continuously running process that listens for events and executes the associated callbacks when they occur. It maintains a queue of tasks (also known as the "message queue" or "task queue") that need to be executed. These tasks can include events like user input, network requests, timers, and other asynchronous operations.



Promise and Promise States

A **'Promise'** is a built-in object that represents a value that might not be available yet but will be at some point in the future. Promises are used to manage asynchronous operations in a more organized and predictable way. Promises provide a way to work with asynchronous code that is more readable and maintainable than using callbacks alone.

Promises have three possible states:

- **Pending**: Initial state, neither fulfilled nor rejected.
- **Fulfilled**: The operation was successful, and the promise now has a resulting value.
- **Rejected**: The operation failed, and the promise now has a reason for the failure

Creating and Consuming Promises

Creating

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  if (/* operation successful */) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

Consuming

```
myPromise
  .then((result) => {
    // Handle the fulfilled promise
  })
  .catch((error) => {
    // Handle the rejected promise
  });
```

Chaining Promises

```
myPromise
  .then((result) => {
    // First operation succeeded, return another Promise
    return anotherAsyncOperation(result);
  })
  .then((result2) => {
    // Handle the result of the second operation
  })
  .catch((error) => {
    // Handle errors from any part of the chain
  });
```

How to Handle Multiple Promises?

The Promise class offers four static methods to facilitate async task concurrency:

- **Promise.all():** Fulfills when all of the promises fulfill; rejects when any of the promises rejects.
- **Promise.any():** Fulfills when any of the promises fulfills; rejects when all of the promises reject.
- **Promise.allSettled():** Fulfills when all promises settle.
- **Promise.race():** Settles when any of the promises settles. In other words, fulfills when any of the promises fulfills; rejects when any of the promises rejects.

Async/await keyword

The '**async/await**' keywords in JavaScript are a pair of language features introduced in ECMAScript 2017 (ES8) to simplify and improve the readability of asynchronous code. These keywords work together to make asynchronous operations appear more like synchronous code, making it easier to understand and maintain.

```
async function process() {
  const result1 = await doSomethingAsync();
  const result2 = await doAnotherAsync();
  return result1 + result2;
}
```

Callback

A **callback** is a function that is passed as an argument to another function and is executed after the completion of that function or at a specified event. Callbacks are widely used to handle asynchronous operations, events, and various other scenarios.

```
function myCallbackFunc() {
  console.log('This is my callback function.');
}

function execute(callback) {
  callback();
}

execute(myCallbackFunc);
```

Callback Hell

Callback hell, also known as the "Pyramid of Doom" or "Callback Pyramid," is a situation in JavaScript programming where multiple nested callbacks make the code difficult to read, understand, and maintain. It often occurs when dealing with asynchronous operations, such as handling callbacks from asynchronous functions or managing sequences of asynchronous tasks. Callback hell can lead to code that is hard to debug and prone to errors.

Callback Hell Example

```
asyncFunction1(function (result1) {
  asyncFunction2(result1, function (result2) {
    asyncFunction3(result2, function (result3) {
      asyncFunction4(result3, function (result4) {
        asyncFunction5(result4, function (result5) {
          // More nested callbacks
        });
      });
    });
  });
});
```

Improved Code Readability and Maintainability

Using Promises

```
asyncFunction1()
  .then((result1) => asyncFunction2(result1))
  .then((result2) => asyncFunction3(result2))
  .then((result3) => asyncFunction4(result3))
  .then((result4) => asyncFunction5(result4))
  .then((result5) => {
    // Handle the final result
  })
  .catch((error) => {
    // Handle errors
  });
```

Using 'async/await'

```
async function processData() {
  try {
    const result1 = await asyncFunction1();
    const result2 = await asyncFunction2(result1);
    const result3 = await asyncFunction3(result2);
    const result4 = await asyncFunction4(result3);
    const result5 = await asyncFunction5(result4);
    // Handle the final result
  } catch (error) {
    // Handle errors
  }
}

processData();
```


Q&A
time

Thank You! Goodbye Everyone!