Event Loop & Promise

Rujuan Xing

Maharishi International University - Fairfield, Iowa

Introduction

- JavaScript executes code in a single thread, which brings a risk of blocking the thread if a single line takes long time to process. This means until that line finishes, the next line of code won't be processed.

- For example, handling of AJAX request should be done on a different thread, otherwise our main thread would be blocked until the network response is received.

Synchronous vs Asynchronous

- **Synchronous JavaScript:** As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.

```
<script>
    document.write("Hi"); // First
    document.write("<br>");
    document.write("Mayukh") ;// Second
    document.write("<br>");
    document.write("How are you"); // Third
</script>
```

- **Asynchronous JavaScript:** Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one.

```
<script>
    document.write("Hi");
    document.write("<br>");
    setTimeout(() => {
        document.write("Let us see what happens");
    }, 2000);
    document.write("<br>");
    document.write("End");
    document.write("<br>");
</script>
```
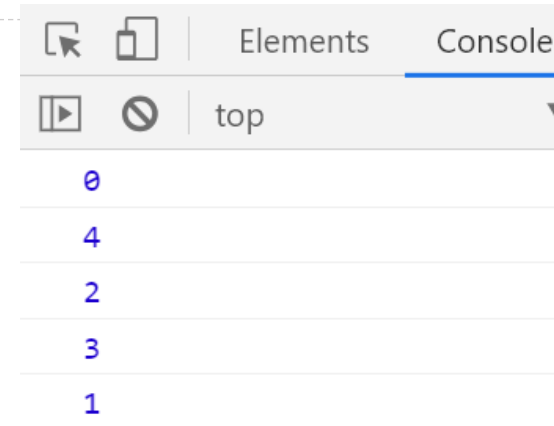
Web APIs

- Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily.

- **Web APIs**: APIs in client-side JavaScript. **Not part of the JavaScript language itself**, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code.

- Two Categories:

  - **Browser APIs** are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it.

    - **APIs for manipulating documents**

    - **APIs that fetch data from the server**

    - **APIs for drawing and manipulating graphics**

    - **Client-side storage APIs, etc…**

  - **Third-party APIs** are not built into the browser by default, and you generally have to retrieve their code and information from somewhere on the Web.

JavaScript Timers/Timing Events

- The `window` object allows execution of code at specified time intervals. These time intervals are called timing events.

- `setTimeout(function, milliseconds)`: Executes a function, after waiting a specified number of milliseconds.

- `setInterval(function, milliseconds)`: Same as setTimeout(), but repeats the execution of the function continuously.

- `clearTimeout(timeoutVariable)`: stops the execution of the function specified in setTimeout().

- `clearInterval(timeoutVariable)`: stops the executions of the function specified in the setInterval() method.

Example: `setTimeout(function, milliseconds)`

```javascript
console.log(0);

setTimeout(function() {
    console.log(1)
}, 2000);

setTimeout(function() {
    console.log(2)
}, 0); //Zero delay doesn't actually mean the call back will fire-
off after zero milliseconds.

setTimeout(function() {
    console.log(3)
}); //default delay time is 0

console.log(4);
```

Elements	Console
top	▼

```
0
4
2
3
1
```

Chrome – The Event Loop

Heap   **Stack**

Reduce()

Filter()

Add()

Main()

*Push only
When Stack
Is Empty*

Event
Loop

Web APIs

SetTimeOut   (callback2)

AJAX (callback1)

Callback-Task Queue/ Macrotask Queue

callback1

Callback 2

onClick callback
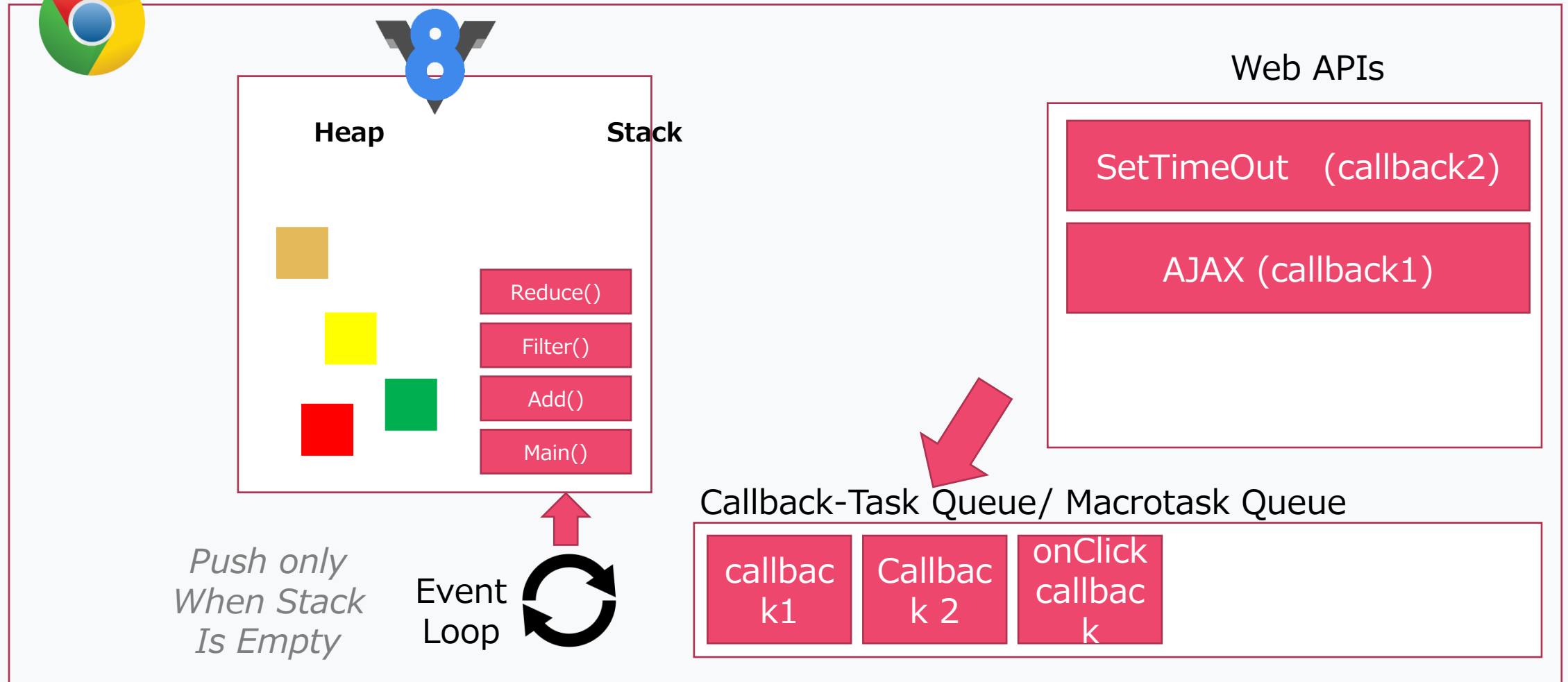
If you block the stack, browser can't run the render queue

Concurrency model and the event loop

- JavaScript has a concurrency model based on an **event loop**, which is responsible for executing the code, collecting and processing events, and executing queued sub-tasks.

- **Stack**: Function calls form a stack of *frames*.

- **Heap**: Objects are allocated in a heap which is just a name to denote a large (mostly unstructured) region of memory.

- **Queue**: A JavaScript runtime uses a message/function queue, which is a list of messages/ functions to be processed.

  - When the stack is once empty, the event loop will process the function in the queue (if there is one), starting with the oldest one.

- Event loop

  - The **event loop** got its name because of how it's usually implemented, which usually resembles:

    ```
    while (queue.waitForMessage()) {
        queue.processNextMessage()
    }
    ```
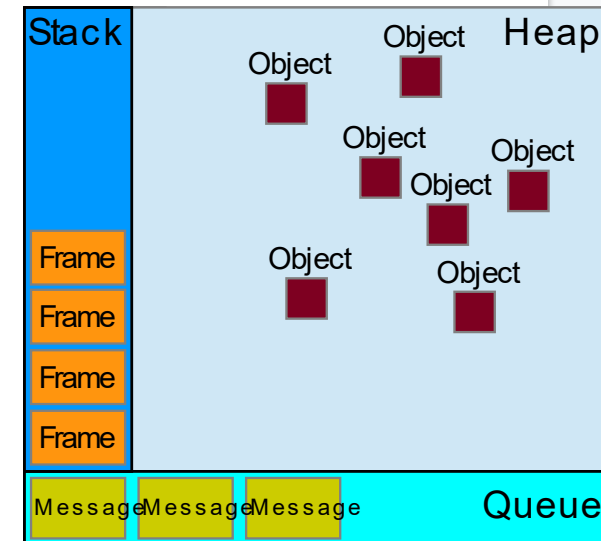
"Run-to-completion"

- Each function is processed completely before any other function is processed. It means each function cannot be pre-empted and will run entirely before any other code runs.

    - This differs from Java, C, for instance, where if a function runs in a thread, it may be stopped at any point by the runtime system to run some other code in another thread.

- Pros of this model:

    - **Never blocking**

    - Handling I/O is typically performed via events and callbacks, so when the application is waiting for an IndexedDB query to return or an XHR request to return, it can still process other things like user input.

- Cons of this model:

    - If a function takes too long to complete, the web application is unable to process user interactions like click or scroll. The browser mitigates this with the "a script is taking too long to run" dialog.

    - Good practice: Make function processing short and if possible cut down one function into several functions.

Blocking the Event-Loop/ Rendering UI

```html
<body>
    <input type="text"><br>
    <script>
        document.querySelector('input').addEventListener('keyup',
            e => {
                let n = 0
                for (let i = 0; i < 9e7; i++) {
                    n = n + e.which;
                }
                console.log(n);
                console.log(e.key);
            });
    </script>
</body>
```

Callbacks vs Asynchronous?

- What is a callback?
  - A callback is a function that is to be executed after another function has finished executing — hence the name 'call back'.
  - In JavaScript, functions can take functions as arguments, and can be returned by other functions. Functions that do this are called higher-order functions. **Any function that is passed as an argument is called a callback function.**

- Are Callbacks Always Asynchronous?

```
console.log(`Start`);
[1, 2, 3].forEach(i => console.log(i));
console.log(`Finish`);

Start
1
2
3
Finish
```

```
console.log(`Start`);
[1, 2, 3].forEach(i => setTimeout(() => console.log(i)),
0);
console.log(`Finish`);

Start
Finish
1
2
3
```

The Boomerang Effect (Callback Hell)

What is "callback hell"?

- Asynchronous JavaScript, or JavaScript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

- See the pyramid shape and all the }) at the end? Eek! This is affectionately known as callback hell.

- The cause of callback hell is when people try to write JavaScript in a way where execution happens visually from top to bottom.

```javascript
fs.readdir(source, function (err, files) {
    if (err) {
        console.log('Error finding files: ' + err)
    } else {
        files.forEach(function (filename, fileIndex) {
            console.log(filename)
            gm(source + filename).size(function (err, values) {
                if (err) {
                    console.log('Error identifying file size: ' + err)
                } else {
                    console.log(filename + ' : ' + values)
                    aspect = (values.width / values.height)
                    widths.forEach(function (width, widthIndex) {
                        height = Math.round(width / aspect)
                        console.log('resizing ' + filename + 'to ' + height + 'x' + height)
                        this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
                            if (err) console.log('Error writing file: ' + err)
                        })
                    }.bind(this))
                }
            })
        })
    }
})
```

Solutions to callback hell

- There are four solutions to callback hell:

  1. Write comments

  2. Split functions into smaller functions

  3. Using **Promises**

  4. Using **Async/await**

What is a Promise?

- A promise is an object that represents something that will be available in the future. In programming, this "something" is values.

- Promises propose that instead of waiting for the value we want (e.g. the image download), we receive something that represents the value in that instant so that we can "get on with our lives" and then at some point go back and use the value generated by this promise.

- Promises are based on time events and have some states that classify these events:

  - Pending: still working, the result is undefined;

  - Fulfilled: when the promise returns the correct result, the result is a value.

  - Rejected: when the promise does not return the correct result, the result is an error object.

Create a Promise Object

```
let promise = new Promise(function(resolve, reject) {
    // executor
});
```

new Promise(executor)

state: "pending"
result: undefined

resolve(value) →
state: "fulfilled"
result: value

reject(error) →
state: "rejected"
result: error

- The function passed to new Promise is called the **executor**. When new Promise is created, **the executor runs automatically**. Only the parts of resolve and reject are going to be asynchronous.

- Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself.

- When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

  - `resolve(value)` — if the job is finished successfully, with result `value`.

  - `reject(error)` — if an error has occurred, `error` is the error object.

- The promise object returned by the new Promise constructor has these internal properties:

  - `state` — initially "`pending`", then changes to either "`fulfilled`" when `resolve` is called or "`rejected`" when `reject` is called.

  - `result` — initially `undefined`, then changes to `value` when `resolve(value)` called or `error` when `reject(error)` is called.

Create a Promise Object (cont.)

```javascript
let promise = new Promise(function(resolve, reject) {
    // the function is executed automatically when the promise is constructed
    // after 1 second signal that the job is done with the result "done"
    setTimeout(() => resolve("done"), 1000);
});
```

new Promise(executor)

| state: "pending" | resolve("done") | state: "fulfilled" |
| result: undefined | → | result: "done" |

```javascript
let promise = new Promise(function (resolve, reject) {
    // after 1 second signal that the job is finished with an error
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

new Promise(executor)

| state: "pending" | reject(error) | state: "rejected" |
| result: undefined | → | result: error |

Consumers: `then, catch, finally`

- A Promise object serves as a link between the executor and the consuming functions, which will receive the result or error. Consuming functions can be registered (subscribed) using methods `.then`, `.catch` and `.finally`.

```javascript
let promise = new Promise(function(resolve, reject) {
    const random = Math.random();
    console.log('random: ', random);
    if (random > 0.5) {
        setTimeout(() => resolve("done!"), 1000);
    } else {
        setTimeout(() => reject(new Error("Whoops!")), 1000);
    }

});

promise.then(result => console.log(result))
    .catch(error => console.log(error))
    .finally(() => console.log("Promise ready!"));
```

Queue Example

```
setTimeout(() => console.log('setTimeout results'), 0);

const promise = new Promise((resolve) => resolve(`Promise results`));

console.log('Code starts');

promise.then(console.log);

console.log('I love JS');
```

```
Code starts
I love JS
Promise results
setTimeout results
```

Why?

- `then`, `catch` and `finally` methods of a promise register the callback functions passed to them and these callbacks are provided to the event loop when the promise is resolved or rejected. These callbacks are added to the microtask queue which has higher priority than macrotask queue

Promises chaining

```javascript
new Promise(function(resolve, reject) {

    setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

    alert(result); // 1

    return result * 2;

}).then(function(result) { // (***)

    alert(result); // 2

    return result * 2;

}).then(function(result) {

    alert(result); // 4

    return result * 2;

});
```
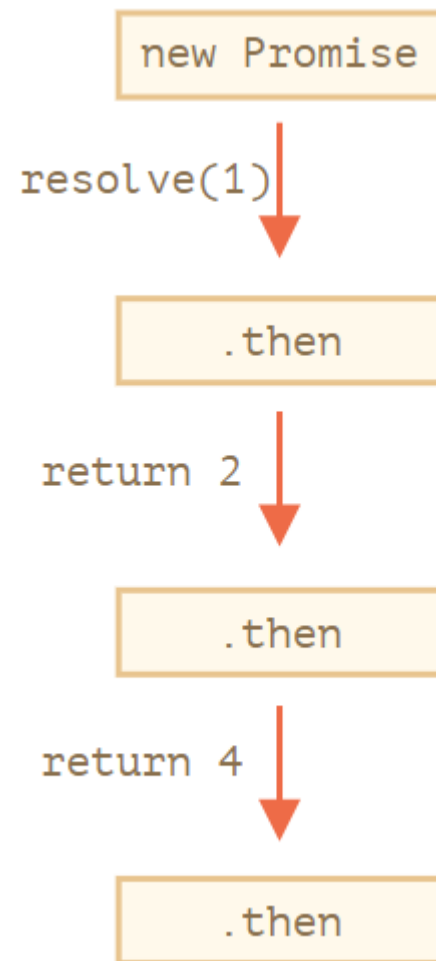
How does Promise solve callback hell problem?

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

```javascript
const verifyUser = function (username, password, callback) {
    dataBase.verifyUser(username, password, (error, userInfo) => {
        if (error) {
            callback(error)
        } else {
            dataBase.getRoles(username, (error, roles) => {
                if (error) {
                    callback(error)
                } else {
                    dataBase.logAccess(username, (error) => {
                        if (error) {
                            callback(error);
                        } else {
                            callback(null, userInfo, roles);
                        }
                    })
                }
            })
        }
    })
};
```

```javascript
const verifyUser = function(username, password) {
    database.verifyUser(username, password)
        .then(userInfo => dataBase.getRoles(userInfo))
        .then(rolesInfo => dataBase.logAccess(rolesInfo))
        .then(finalResult => {
            //do whatever the 'callback' would do
        })
        .catch((err) => {
            //do whatever the error handler needs
        });
};
```

Promise.all

- `Promise.all` takes an array of promises (it technically can be any iterable, but is usually an array) and returns a new promise.

- The new promise resolves when all listed promises are settled, and the array of their results becomes its result.

```
let promise = Promise.all([...promises...]);
```

```
Promise.all([
    new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
    new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
    new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(console.log); // 1,2,3 when promises are ready: each promise contributes an array member
```

- The order of the resulting array members is the same as in its source promises. Even though the first promise takes the longest time to resolve, it's still first in the array of results.

Promise.all (cont.)

- If any of the promises is rejected, the promise returned by `Promise.all` immediately rejects with that error.

```
Promise.all([
    new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
    new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
    new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(console.log); // Error: Whoops!
```

- `Promise.all(iterable)` allows non-promise "regular" values in `iterable`

```
Promise.all([
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(1), 1000)
    }),
    2,
    3
]).then(console.log);
```

Promise.race

- Similar to `Promise.all`, but waits only for the first settled promise and gets its result (or error).

```
Promise.race([
    new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
    new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")),
2000)),
    new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(console.log); // 1
```

- The first promise here was fastest, so it became the result. After the first settled promise "wins the race", all further results/errors are ignored.