

Callbacks and Asynchronous JavaScript

CIS 1962 (Fall 2025)
September 22th, 2025

Lesson Plan

Asynchronous Programming

4	Asynchronous Programming
11	Callback Functions
19	Promises and async/await
31	The Event Loop
40	Advanced Async Ideas

PollEverywhere!

We will use PollEv to take attendance and do polls during lecture. Scan this QR Code or use the link to join for attendance! (Please add your name for identification)

PollEv.com/voravichs673





Asynchronous Programming

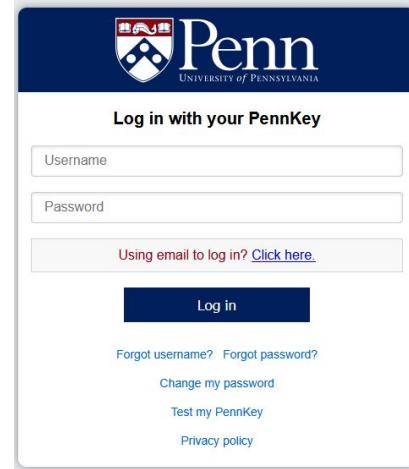
Why do we need asynchronous
programming in JavaScript?

Why Asynchronous?

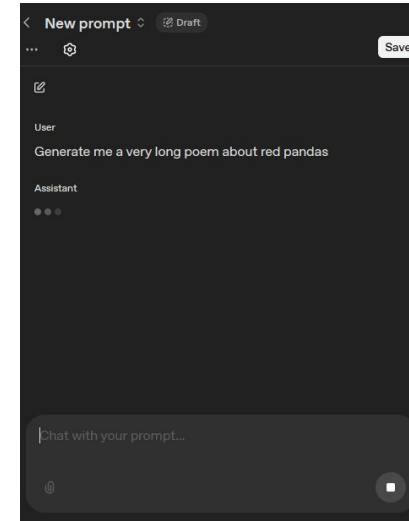
Think about the things you do on websites on a daily basis.
How long do those tasks take to process?



Database Queries



Logging In



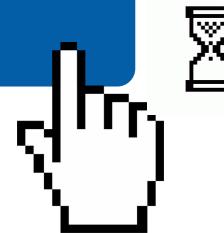
Waiting for LLMs

Why Asynchronous?

JavaScript is **single-threaded**.

If these long tasks were synchronous, the single thread would be reserved, causing the web page to become unresponsive!

Do Long Task



Can't type here while waiting!

Synchronous Code: Try It Yourself!

Paste the code below into your browser's console.

Watch your current web page become unresponsive for 5 seconds as it waits!

```
let end = Date.now() + 5000;
while(Date.now() < end) {}
alert('Done!');
```

Comparison: Sync vs. Async

Synchronous Programming

- Code runs line-by-line
- Every block/line of code waits for the previous to finish
- Web page may become unresponsive for long tasks

Asynchronous Programming

- Some operations run separately from the line-by-line execution
- Tasks can be reserved while code execution continues
- Web page remains responsive even despite long tasks

Comparison Example

```
console.log("Start");
// Synchronous function (blocks)
const data = fetchData();
console.log("End");
```

Start
(long wait)
End

Synchronous

```
console.log("Start");

function fetchDataAsync(callback) {
    fetch('https://jsonplaceholder.typicode.com/todos/1')
        .then(response => response.json())
        .then(data => callback(data))
}
fetchDataAsync(function(data) {
    console.log("Fetched data:", data);
});
console.log("End");
```

Start
End
Fetched data: { userId: 1, id: 1, title: 'delectus aut autem', completed: false }

Asynchronous

Asynchronous Web APIs

JavaScript uses asynchronous APIs to handle its event-driven programming. These APIs include:

- Network APIs: HTTP / `fetch()` / `axios`
- Timers (e.g `setTimeout()`)
- Event Handlers (e.g. waiting for a click)
- Web Workers



Callback Functions

How do we make functions that
can be called at a later time?

Types of Callback Functions

There are two types of callback functions:

Synchronous Callbacks

- Invoked immediately during execution of higher-order functions

```
let words = ["hi", "hello",  
"hey"];  
let longWord = words.find(w =>  
w.length > 3); // "hello"
```

Asynchronous Callbacks

- Invoked later after an asynchronous task completes, scheduled via the event loop

```
console.log("start");  
setTimeout(() => {  
    console.log("timeout  
callback");  
}, 1000);  
console.log("end");
```

Delegation

When we want asynchronous actions to occur, such as waiting for something to load or waiting for user input, we don't want JavaScript's main thread to stop or wait.

We must **delegate** or **hand over** the task to the browser or Web API.

Delegation Example

“I want to wait for the user to press this button, then do something”

Press Me!

delegates to:



```
document.getElementById("button").addEventListener("click", handleButtonClick)
```

Event handler

callback

Callback Functions

When the delegated task is “done”, such as:

- A timer finishing
- A click on a button is detected
- A resource is finished fetching from an API

Then the **callback function** specified by the delegation is scheduled to run.

Callback functions are always required. You want something to happen after certain tasks finish after all!

Callback Functions Example

```
document.getElementById("button").addEventListener("click",
  handleButtonClick);
  On click, do this
function handleButtonClick() {
  const jokeDiv = document.getElementById("joke");
  jokeDiv.textContent = "Loading joke...";

  // Fetch a random joke from an API
  fetch("https://icanhazdadjoke.com/", {
    headers: { Accept: "application/json" }
  })
    .then(response => response.json())
    .then(data => {
      jokeDiv.textContent = data.joke;
    })
    .catch(err => {
      jokeDiv.textContent = "Oops! Couldn't fetch a joke.";
    });
}
```

**Also async!
(promise)**

**Promise
chaining**

"Callback Hell"

Also called: "The Pyramid of Doom"

```
login(user, pass, function(err, token) {  
    if (err) return showError(err);  
    loadProfile(token, function(err, profile) {  
        if (err) return showError(err);  
        fetchSettings(profile, function(err, settings) {  
            if (err) return showError(err);  
            renderUI(profile, settings);  
        });  
    });  
});
```



Best Practices for Callback Use

Modularize and document your functions: Name and label your functions that use callbacks so that they are more readable and reusable

```
// Handles authentication and generates a login token
function login(user, pass, callback){ ... }

// Gives feedback to webpage about login results
function handleLogin(err, token){ ... }

// Submit handler, user submitted a login form
document.getElementById('login-form').addEventListener('submit',
function(event) {
    ... // omitted: get the username/password from text inputs
    login(user, pass, handleLogin);
})
```



Modern Solutions for Callbacks

How do we get out of callback hell
with Promises and async/await?

Modern Solutions for Callbacks

There are a few modern solutions that directly aim to solve the “pyramid of doom” for callbacks.

Promises are objects that represent the different states, like failure and completion, of an asynchronous operation.

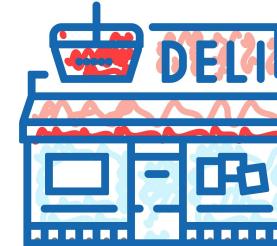
async/await is syntax that builds upon promises to allow asynchronous code to be written like synchronous code.

Promises

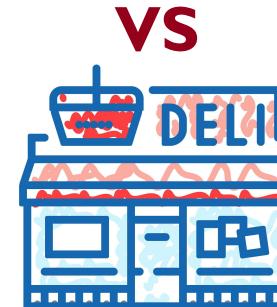
Promises flip the script:

Instead of immediately **passing callbacks** into functions...

You can **attach callbacks** to a returned object, a promise!



Leave your name to order

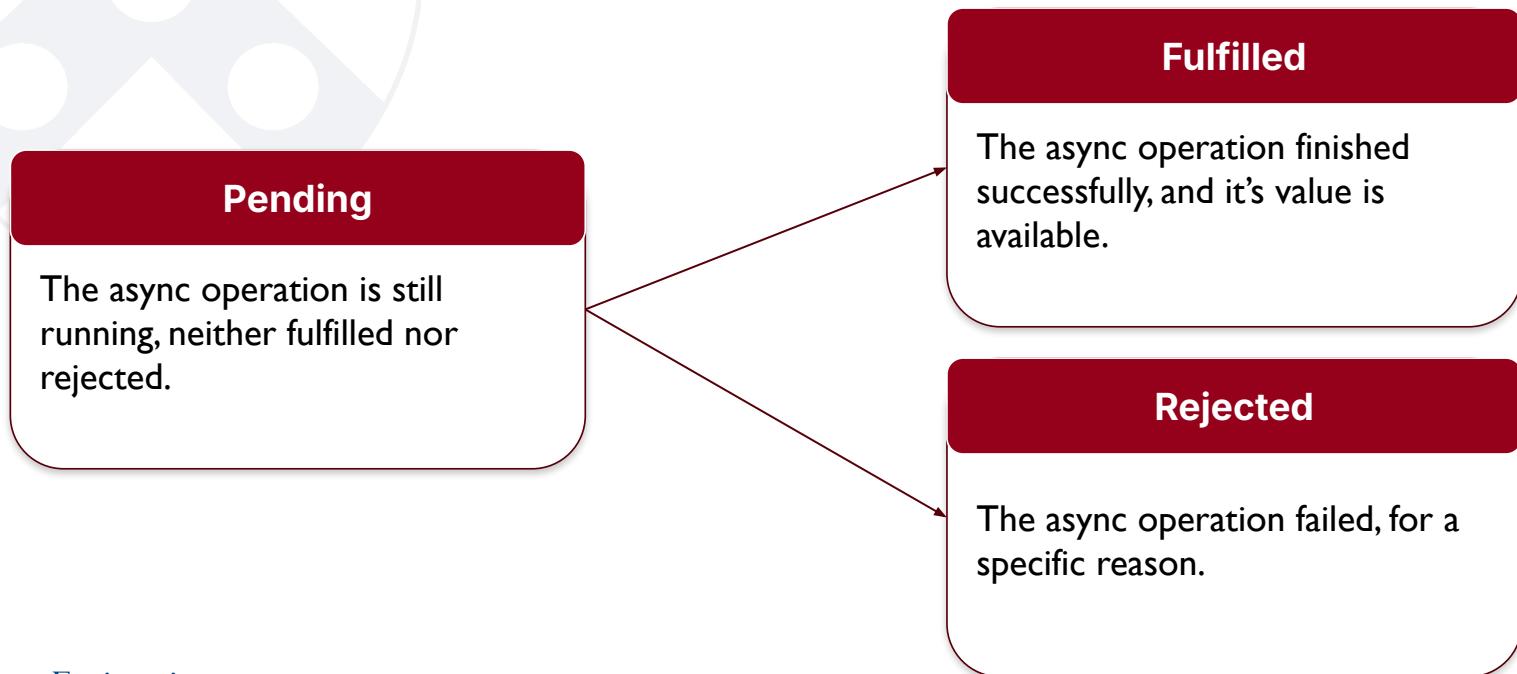


VS

Taking a ticket to order at the deli

Promise States

A Promise can be in one of 3 states:



Creating a Promise

You can use the **Promise constructor** to make promises.

As an argument, it takes an executor function that **runs immediately**.

You must specify the **resolve()** and/or the **reject()** methods and the conditions they occur, so that the promise may change state properly.

When a promise settles in a resolve or reject, the state can no longer change.

```
const alwaysOk = new Promise((resolve, reject) => {
  resolve("I always succeed! :)");
});
const alwaysFail = new Promise((resolve, reject) => {
  reject("I always fail! :(");
};

alwaysOk.then(value => console.log(value));
// I always succeed!
alwaysFail.catch(err => console.error(err));
// I always fail!
```

Consuming a Promise

The act of “**consuming**” a promise means to react to the completion of promises, through the **.then** and **.catch** methods.

Because each **.then** method returns another promise, you can chain these together, letting the results of the previous **.then** methods flow into the next!

Promise Chaining with Fetch

The **Fetch API** is often used with promise chaining.

Fetch allows you to make HTTP requests. It returns a Promise, which is often consumed to process the data into JSON for use.

```
fetch('https://dog.ceo/api/breeds/image/random')
  .then(response => response.json())
  .then(data => {
    console.log("Here is a dog image:", data.message);
  })
  .catch(error => {
    console.error("Fetch error:", error);
});
```

async/await

Asynchronous functions, using **async/await**, is syntactic sugar that specifies another way of consuming a promise.

It abstracts away asynchronous code that would've used promises, to allow code to be written without promise chaining and callbacks.

Instead of .then and .catch, we use try/catch blocks for **async/await**.

async/await Example

Promises

```
fetch('https://dog.ceo/api/breeds/image/random')
  .then(response => response.json())
  .then(data => {
    console.log("Here is a dog image:", data.message);
  })
  .catch(error => {
    console.error("Fetch error:", error);
})
});
```

async/await

```
async function getDogImage() {
  try {
    const response = await
      fetch('https://dog.ceo/api/breeds/image/random');
    const data = await response.json();
    console.log("Here is a dog image:", data.message);
  } catch (err) {
    console.error("Error:", err);
  }
}
getDogImage()
```

POLL QUESTION

```
function doWork(flag) {  
    return new Promise((resolve, reject) => {  
        if (flag) {  
            setTimeout(() => resolve('A'), 100);  
            setTimeout(() => reject('B'), 50);  
        } else {  
            setTimeout(() => reject('C'), 50);  
            setTimeout(() => resolve('D'), 100);  
        }  
    });  
}  
doWork(true)  
.then(result => console.log(result))  
.catch(error => console.log(error));
```



What will be printed to the console after this runs?

B

POLL QUESTION

```
async function getUserName(userId) {  
  let response =  
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`);  
  let data = response.json();  
  return data.name;  
}  
getUserName(2).then((data) => console.log(data))
```

```
async function getUserName(userId) {  
  let response = await  
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`);  
  let data = await response.json();  
  return data.name;  
}  
getUserName(2).then((data) => console.log(data))
```



**What lines should include
the ‘await’ keyword?**

2, 3



5-Minute Break!



The Event Loop

How does JavaScript manage
multiple asynchronous tasks?

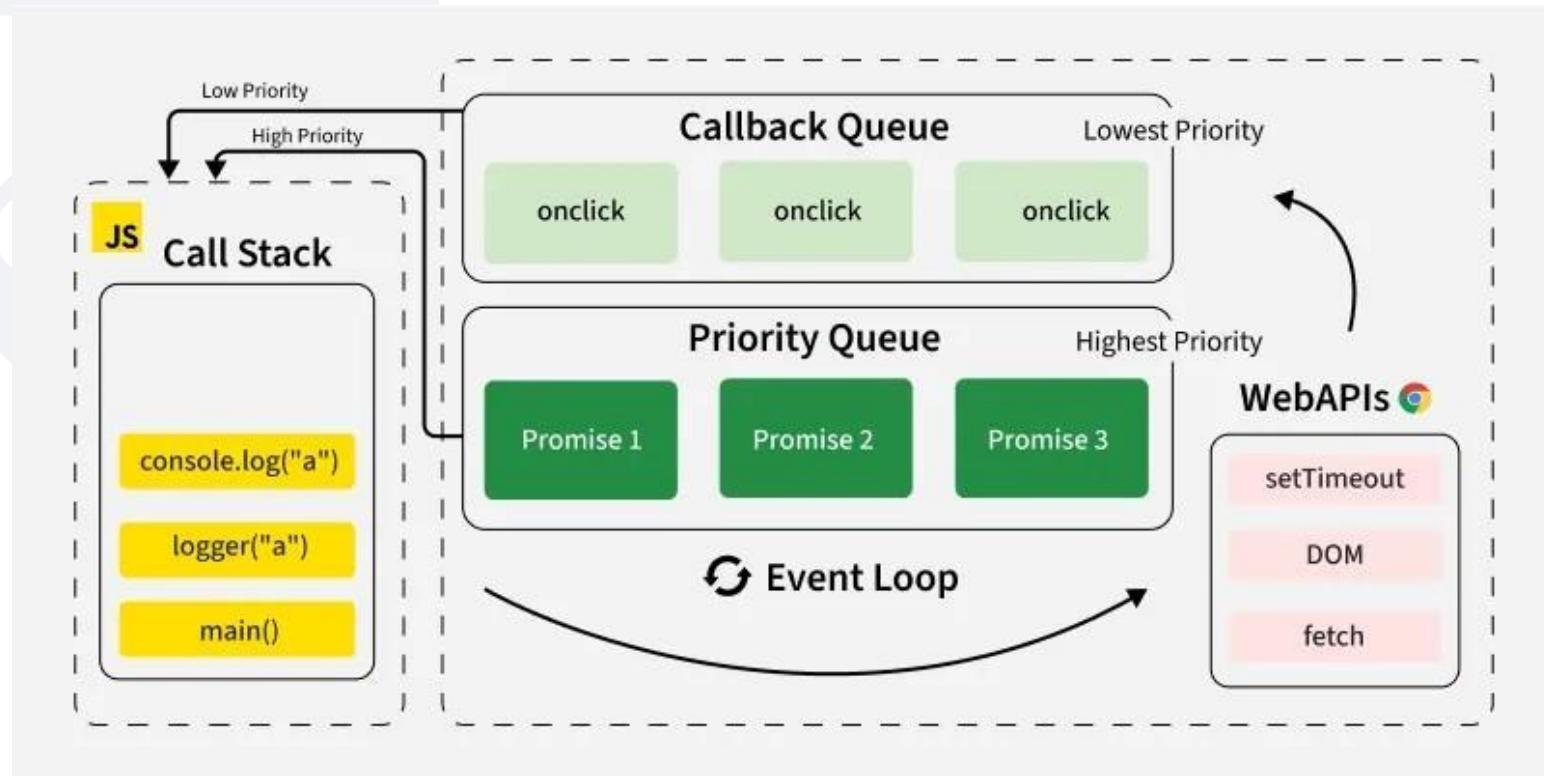
What is the Event Loop?

When performing async tasks, what order will tasks execute in, relative to normal synchronous code?

JavaScript uses an **event loop** to manage the order of execution for asynchronous events.

The event loop manages a **call stack** for running function calls and a **micro/macrotask queues** for scheduling events.

Event Loop Illustration



In the (Event) Loop

The event loop continually checks:

- Is the call stack empty?
- Are there callbacks waiting in the micro/macrotask queues?

If both of these are true, the event loop moves tasks from the **queues** onto the **call stack** for execution.

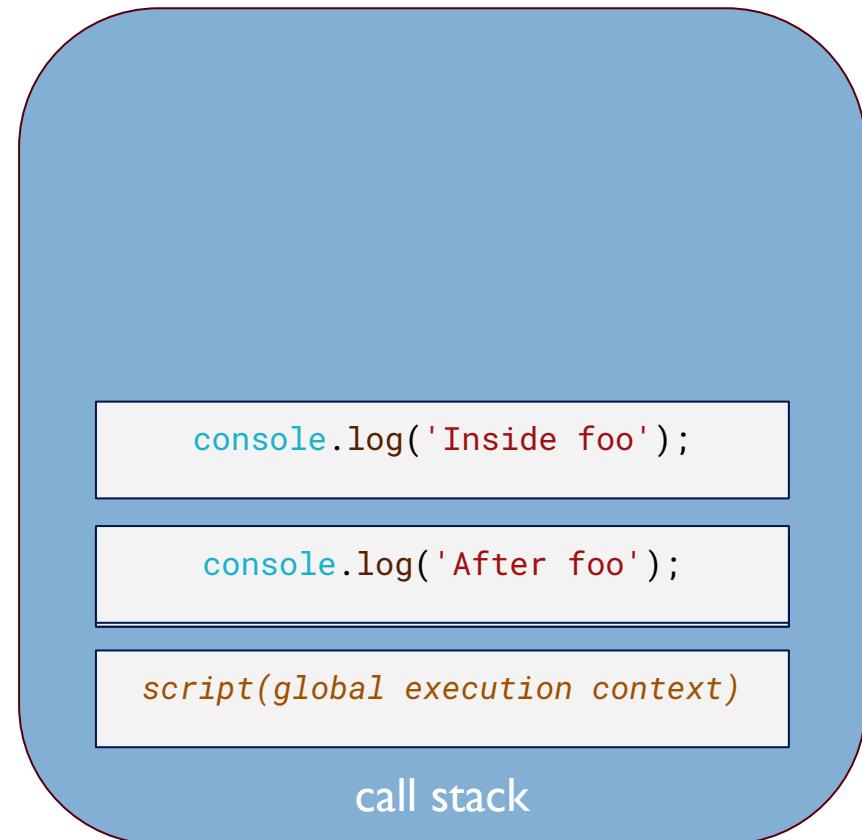
The looping nature allows asynchrony because there are constant checks for asynchronous tasks to execute.

Call Stack: Where JS Runs Code

The call stack is where JavaScript runs function calls.

In a last-in-first-out (LIFO) manner, new functions can be **pushed** onto the stack, later to be **popped** off the stack.

```
function foo() {  
    console.log('Inside foo');  
}  
console.log('Before foo');  
foo();  
console.log('After foo');
```

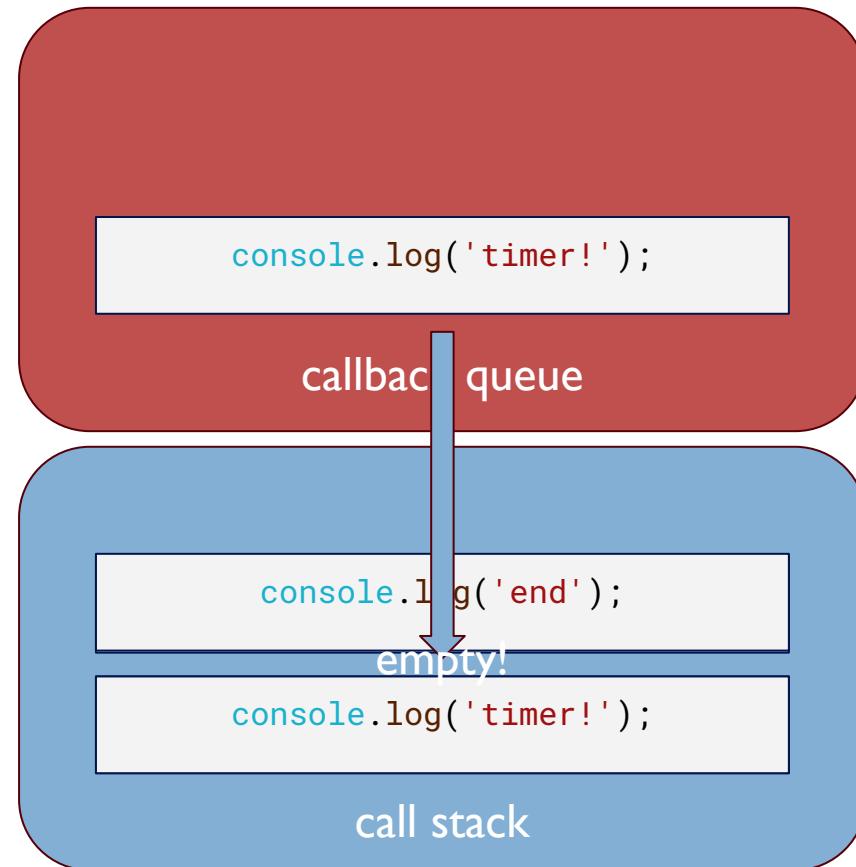


Call Stack: In the Loop

The event loop waits for the call stack to be empty.

This means as long as there are still tasks to execute in the call stack, other asynchronous tasks cannot be scheduled.

```
console.log('start');
setTimeout(() =>
  console.log('timer!'), 0);
console.log('end');
```



Web APIs and the Task Queues

In order for asynchronous execution to occur, JS's engine hands asynchronous tasks off to **web APIs** (such as timer and fetch)

When the asynchronous task **completed or detected**, such as a timer completing or triggering a click handler, the callback function is placed in one of two task queues:

- **Microtask/Priority queue:** used for promises, higher priority
- **Macrotask/Callback queue:** all other tasks, lower priority

Callback/Microtask Queue: Example

```
console.log('Script start');

setTimeout(() => {
  console.log('Timeout callback 1');
}, 2000);

Promise.resolve().then(() => {
  console.log('Promise callback');
});

setTimeout(() => {
  console.log('Timeout callback 2');
}, 0);

console.log('Script end');
```

Script start
Script end
Promise callback
Timeout callback 2
Timeout callback 1

console.log('Timeout callback 1');

callback queue

console.log('Promise callback');

microtask queue

POLL QUESTION

```
console.log('1');
setTimeout(() => {
  console.log('2');
  Promise.resolve().then(() => {
    console.log('3');
  });
}, 0);
Promise.resolve().then(() => {
  console.log('4');
  setTimeout(() => {
    console.log('5');
  }, 0);
});
console.log('6');
```



**What is the order of the
console.log() output?**

1, 6, 4, 2, 3, 5



Advanced Async Ideas

What are some complications that can arise from the order or timing and task scheduling in the event loop?

Race Conditions

Asynchronous actions may have an unpredictable runtime, for instance the time it takes for a network resource to be fetched.

In these cases, the outcome of a program may differ depending on the order in which asynchronous actions occur. This is a **race condition**.

Race Conditions: Order Matters

```
let counter = 0;

setTimeout(() => {
  let tmp = counter;
  setTimeout(() => {
    counter = tmp + 1;
  }, Math.random() * 10);
}, 0);

setTimeout(() => {
  let tmp = counter;
  setTimeout(() => {
    counter = tmp + 1;
    console.log(counter); // logs 1 or 2
  }, Math.random() * 10);
}, 0);
```

Race Conditions: Promise.all()

```
let counter = 0;

function safeIncrement() {
    return new Promise(resolve => {
        setTimeout(() => {
            counter += 1;
            resolve();
        }, Math.random() * 10);
    });
}

Promise.all([safeIncrement(), safeIncrement()])
    .then(() => {
        console.log(counter);
    });

```

Promise.all() can synchronize the asynchronous calls, so that ‘Both done!’ is only printed when both timers complete, regardless of the order.

Promise.all() only fulfills when the all input promises are fulfilled as well.

Think of it as the **&&** operator for multiple promises.

Race Conditions: Promise.race()

```
function fetchWithTimeout(url, ms) {
  const fetchPromise = fetch(url);

  const timeoutPromise = new Promise((_, reject) =>
    setTimeout(() => reject(new Error("Timeout")), ms)
  );

  return Promise.race([fetchPromise, timeoutPromise]);
}

fetchWithTimeout("https://dog.ceo/api/breeds/image/random", 1000)
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

Let's say you want to reject a fetch call if it takes too long.

You can do so with **Promise.race()**, which fulfills only the first promise that settles.

Think of it as the || operator of multiple promises.

Starvation

As you know, an async operation cannot run if the call stack is still running methods, or a microtask (promise) is still running.

What if a certain async operation is never given a chance to run?

```
function floodMicrotasks() {
    Promise.resolve().then(floodMicrotasks);
}
setTimeout(() => {
    // never runs (starved)
    console.log("Timer fired");
}, 0);

floodMicrotasks();
```

Infinitely runs,
timer is never
fired!

Avoiding Starvation

Starvation occurs when some async task is blocked by other tasks that are executing.

Pay attention what tasks may take a while (or go infinitely).

Understand where the starved task is within the event loop!

```
function floodMacrotasks() {  
    setTimeout(floodMacrotasks, 0);  
}  
setTimeout(() => {  
    console.log("Timer fired");  
}, 0);  
floodMacrotasks();
```

Infinitely runs,
but the timer
does fire!

Deadlocks

Sometimes, two async tasks may wait for each other to complete, but are never able to complete due to either one not starting without the other. This is called a **deadlock**.

While JS is single-threaded, there are still ways to write code that logically deadlocks, and freezes execution.

Deadlock Example

```
function doTaskA(callback) {
  if (taskBisDone) {
    callback();
  } else {
    setTimeout(() => doTaskA(callback), 100);
  }
}

function doTaskB(callback) {
  if (taskAisDone) {
    callback();
  } else {
    setTimeout(() => doTaskB(callback), 100);
  }
}
```

```
let taskAisDone = false;
let taskBisDone = false;

doTaskA(() => {
  taskAisDone = true;
  console.log("Task A done!");
});

doTaskB(() => {
  taskBisDone = true;
  console.log("Task B done!");
});

// A waits for B to be done
// B waits for A to be done
// Neither can complete
```

Avoiding Deadlocks

Avoid Circular Dependency: To prevent deadlock conditions, avoid cases where one async task depends on another, and vice versa.

Sequence Tasks: Using defined sequences, like through promise chaining, will prevent deadlocks from occurring

POLL QUESTION

```
function delay(msg, ms, shouldReject = false) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      shouldReject ? reject(msg) : resolve(msg);  
    }, ms);  
  });  
  
Promise.race([  
  delay('A', 100),  
  delay('B', 50, true),  
  delay('C', 75),  
])  
.then(result => console.log('Resolved with:', result))  
.catch(error => console.log('Rejected with:', error));
```



What should be logged to the console?

Rejected with: B



Live Coding: Fetch!
