Asynchronous JavaScript and Object-Oriented Programming

CIS 1962 (Winter 2026)
February 12th, 2026

Penn Engineering

Lesson Plan

Asynchronous Programming

Object-Oriented Programming

Weekly Logistics

Homework 3: Echo Chatbot DUE TONIGHT*

Homework 4: Project Scaffolding DELAYED (Released Monday, 2/16, due 2/26)

- As this homework relies on content from homework 3, this should allow people to finish both in a timely manner!

Review Activity

https://edstem.org/us/courses/91614/lessons/159449/slides/936745

Let's review some content from the previous lecture before we start!

Asynchronous Programming

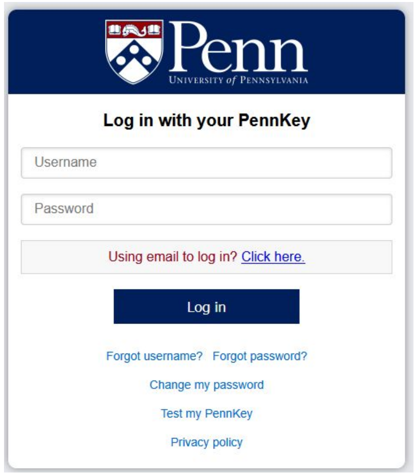
Why do we need asynchronous programming in JavaScript?

Why Asynchronous?

Think about the things you do on websites on a daily basis.

How long do those tasks take to process?

Database Queries


Penn
University of Pennsylvania

Log in with your PennKey

Username

Password

Using email to log in? Click here.

Log in

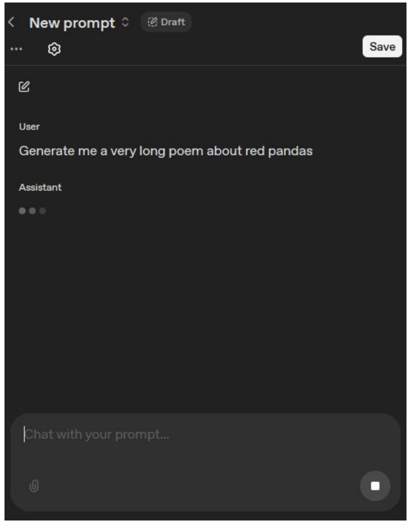
Forgot username? Forgot password?

Change my password

Test my PennKey

Privacy policy

Logging In



User

Generate me a very long poem about red pandas

Assistant

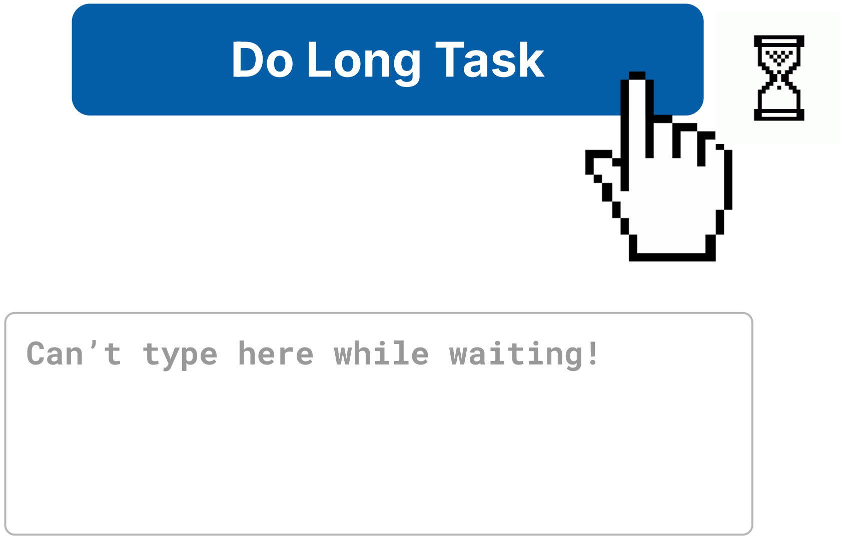
Chat with your prompt...

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!



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?

Penn Engineering

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 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);
          });
     });
});
```

Penn Engineering

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?

Penn Engineering

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!

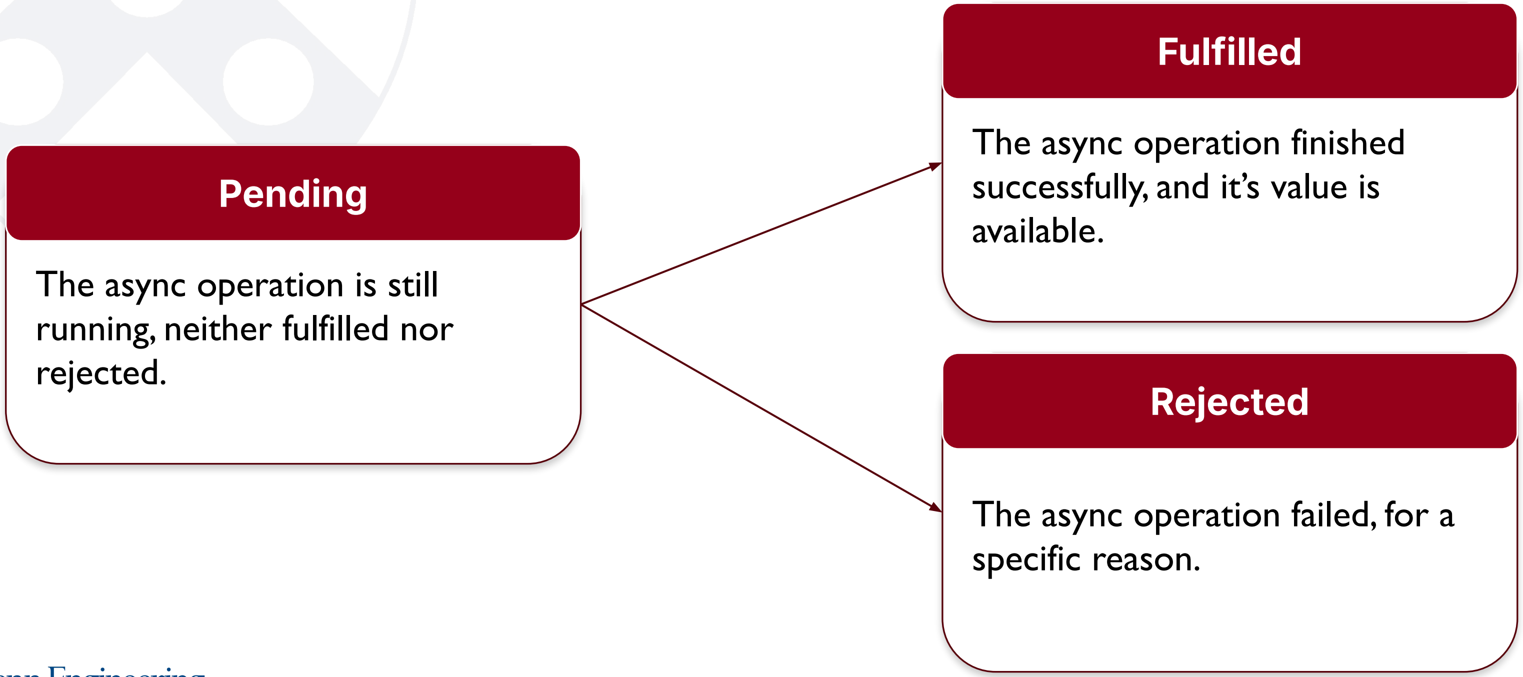
Leave your name to order

VS

Taking a ticket to order at the deli

Promise States

A Promise can be in one of 3 states:



Pending

The async operation is still running, neither fulfilled nor rejected.

Fulfilled

The async operation finished successfully, and it’s value is available.

Rejected

The async operation failed, for a specific reason.

Creating a Promise

You can use the **Promise constructor** to make promises, and many APIs use and return promises to be consumed (TS: generics can be used with <>)

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.

```
// Typescript allows use of generics:
const alwaysOk: Promise<string> = new Promise<string>((resolve, reject) => {
    resolve("I always succeed! :)");
});
const alwaysFail: Promise<string> = new Promise<string>((resolve, reject) => {
    reject("I always fail! :(");
});
alwaysOk.then(value => console.log(value));
alwaysFail.catch(err => console.error(err));
```

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`: marks functions to return a promise
- `await`: pauses a function until a Promise settles, can only be used in an async function*

*ES2022 allows top-level await now, but that's beyond the scope of this course

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.log.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()
```

Penn Engineering

CLASS ACTIVITY

https://edstem.org/us/courses/91614/lessons/159449/slides/936738

The Event Loop

How does JavaScript manage multiple asynchronous tasks?

Penn Engineering

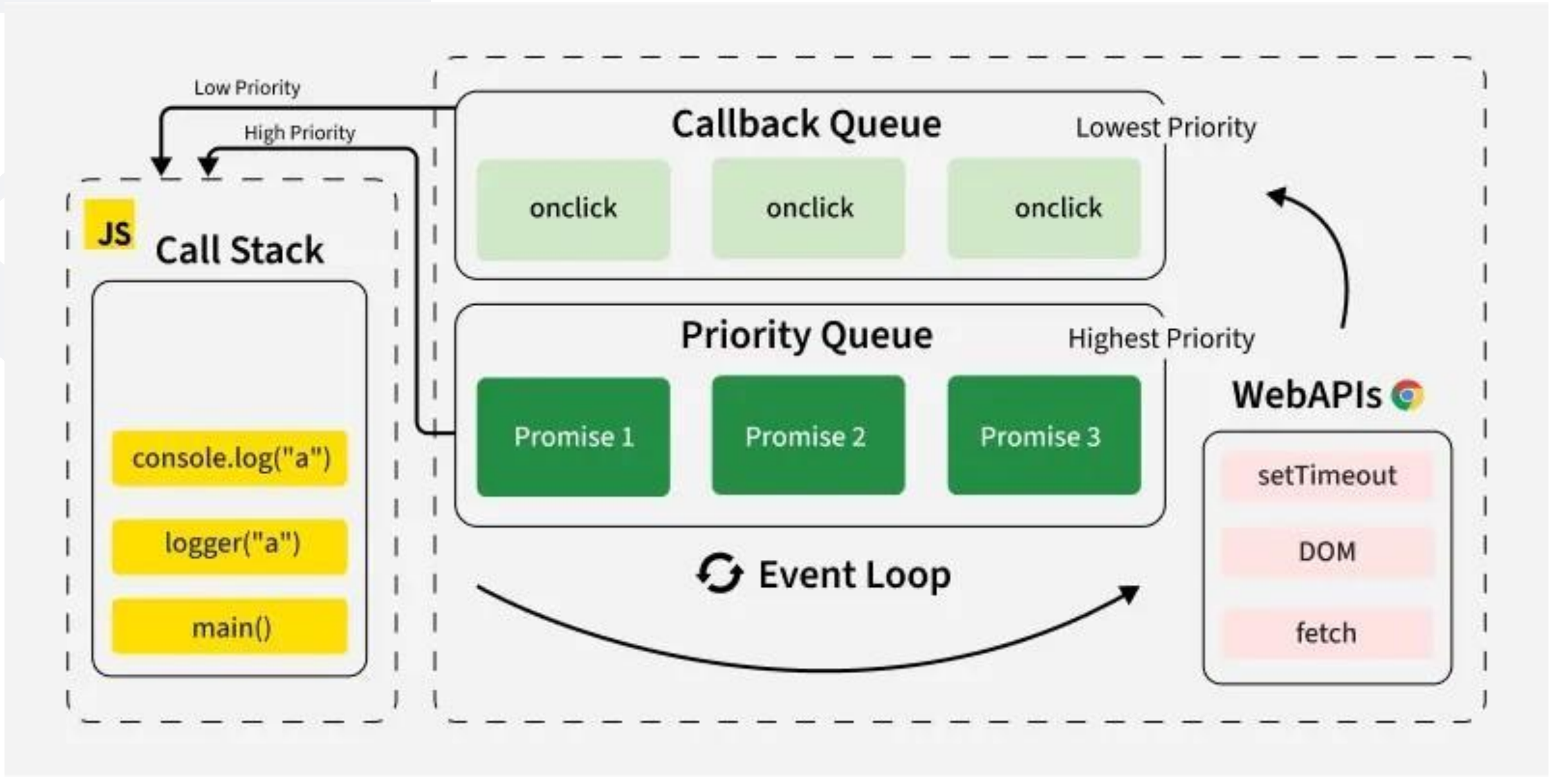
What is the Event Loop?

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



Source: https://media.geeksforgeeks.org/wp-content/uploads/20250208123836185275/Event-Loop-in-JavaScript.jpg

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.

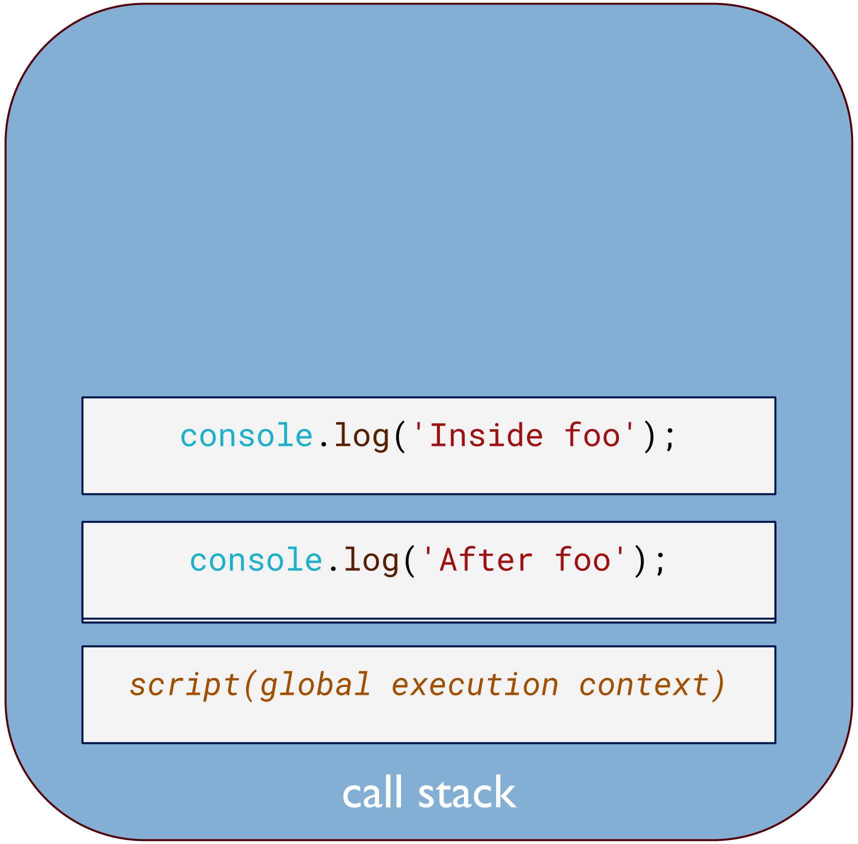
The looping nature allows asynchrony because there are constant checks for asynchronous tasks to execute.

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');
```

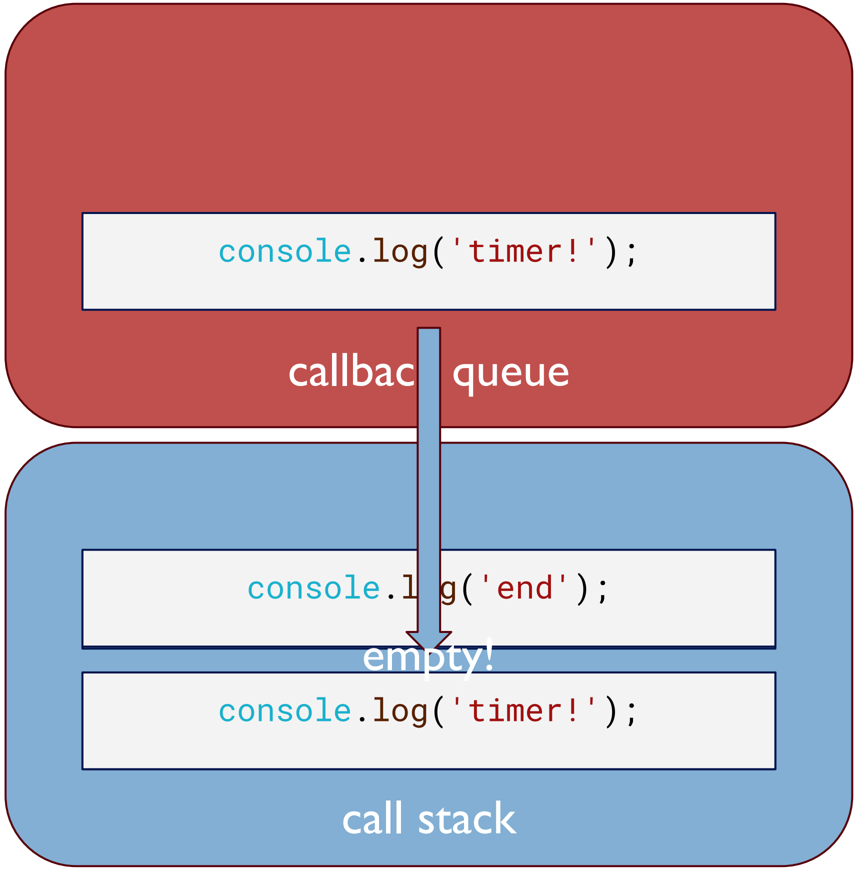

console.log('Inside foo');
console.log('After foo');
script(global execution context)
call stack

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)

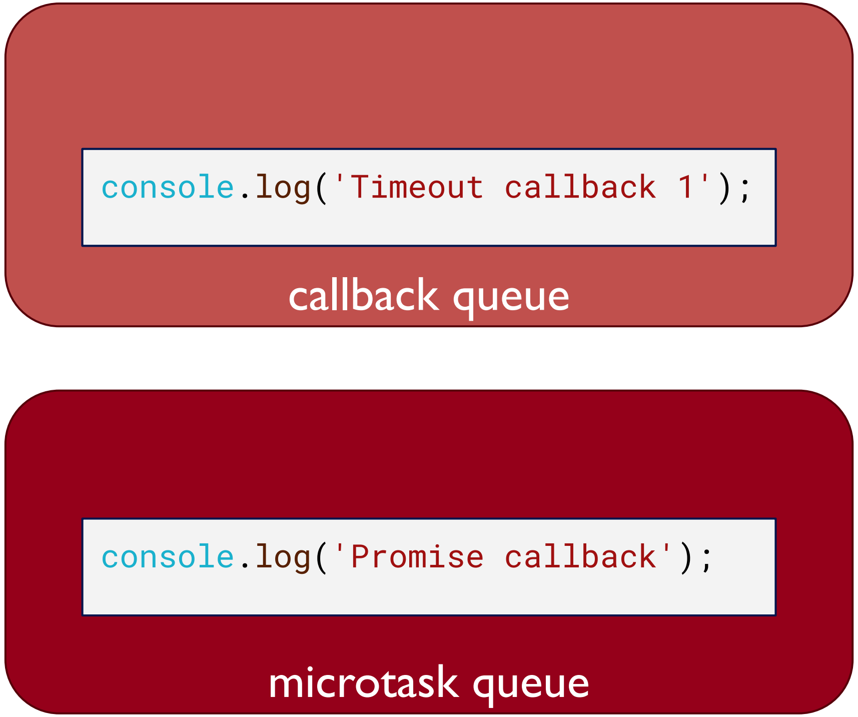
When the asynchronous task **completed or detected**, such as a timer completing or triggering a click handler, the callback function is placed in one of two task queues:

- **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
```

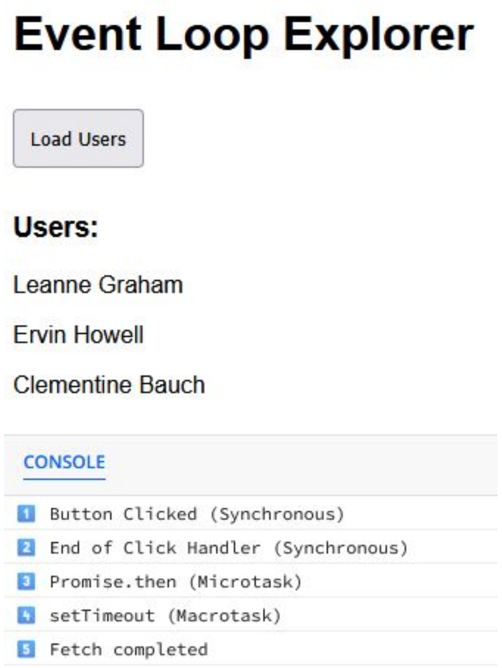


```
console.log('Timeout callback 1');
```

```
console.log('Promise callback');
```

Class Activity: Fetch & Event Loop

https://edstem.org/us/courses/91614/lessons/159449/slides/936781


Event Loop Explorer

Users:

Leanne Graham

Ervin Howell

Clementine Bauch

```
1 Button Clicked (Synchronous)
2 End of Click Handler (Synchronous)
3 Promise.then (Microtask)
4 setTimeout (Macrotask)
5 Fetch completed
```

Extra Async Ideas

https://edstem.org/us/courses/91614/lessons/159488/edit/slides/936964

- Race Conditions
- Starvation
- Deadlock

5-Minute Break!

Object-Oriented Programming

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming style that uses **classes** and **objects** to model data and behavior.

JavaScript makes extensive use of objects, but only recently introduced classes with ES6.

OOP promotes code reuse and scalability through its various features:

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Abstraction/Encapsulation

Abstraction focuses on exposing only essential features and hiding unnecessary details from the user.

Encapsulation is the practice of bundling data and methods together and restricting direct access to some of the object's components.

JavaScript, while it doesn't have dedicated abstract classes or interfaces, achieves these ideas through:

- Functions
- Objects/Methods
- Scope/Closures
- Classes

Abstraction Examples

```
function calculateArea(radius) {
      return
            Math.PI * radius * radius;
}

console.log(calculateArea(5));
```

Functions

```
const bankAccount = {
      balance: 0,
      deposit(amount) {
            this.balance += amount;
      },
      getBalance() {
            return this.balance;
      }
};

bankAccount.deposit(100);
console.log(bankAccount.getBalance());
```

Objects/Methods

Encapsulation Examples

```
function makeCounter() {
  let count = 0; // not directly accessible
  return { // Creates closures
    increment() {
      count++;
    },
    getCount() { // access count variable
      return count;
    }
  };
}

const counter = makeCounter();
counter.increment();
console.log(counter.getCount()); // 1
console.log(counter.count); // undefined
```

Scope/Closures

```
class Rectangle {
  #width; // private, hidden to user
  #height; // private, hidden to user
  constructor(width, height) {
    this.#width = width;
    this.#height = height;
  }
  getArea() {
    return this.#width * this.#height;
  }
}

const rect = new Rectangle(10, 5);
console.log(rect.getArea());
```

Classes

Inheritance

Inheritance lets one class (a subclass) acquire the properties and methods of another class (a superclass).

Before classes in ES6, JavaScript used to use prototypal inheritance, but now it has proper class syntax and the `extends` keyword for inheritance.

```
class Animal { // Superclass
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a
sound.`);
  }
}

class RedPanda extends Animal { // Subclass
  speak() {
    console.log(`${this.name} says wah!`);
  }
}

const klein = new RedPanda("Klein");
klein.speak(); // Klein says wah!
```

Polymorphism

Polymorphism is the practice of allowing objects to share and override behaviors. It allows the same method to function differently based on what object they act on.

JavaScript handles polymorphism in 2 ways:

- Method Overriding
- Method Overloading (not natively)

Method Overriding

Overriding lets a subclass provide a different implementation of a method in a superclass.

You need no extra syntax, just use the same name for the method within a class that extends a superclass.

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(`${this.name} makes a
sound.`);
  }
}

class RedPanda extends Animal {
  speak() { // Overrides Animal’s speak()
    console.log(`${this.name} says wah!`);
  }
}

const klein = new RedPanda("Klein");
klein.speak(); // Klein says wah!
```

Method Overloading

```
class Calculator {
     // Check arguments, then do something
different depending on existing arguments
     add(a, b) {
          if (b) {
               return a + b;
          }
          return a + a;
     }
}

const calc = new Calculator();
console.log(calc.add(1)) // 2
console.log(calc.add(1,2)) // 3
```

Method overloading allows you to have multiple methods with the same name but different arguments.

JavaScript **does not natively support method overloading**, however we can simulate it through checking arguments.

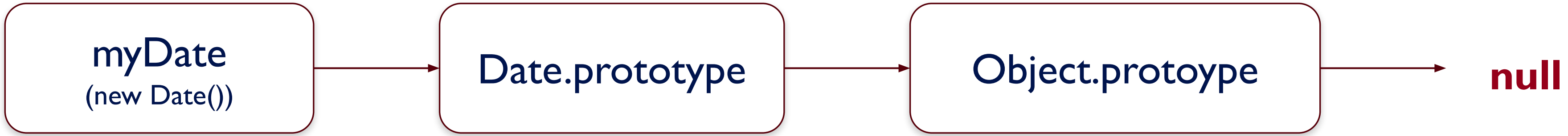
Pre-ES6: Prototypes

How did JavaScript handle OOP without using classes like other programming languages do?

JavaScript and Prototypes

All objects in JS have an internal link to another object called a **prototype**, which define shared properties and methods that other objects will inherit.

If a property or method is not found in an object itself, JavaScript will look for it in the prototypes, up the **prototype chain**, until the end of the chain (null)



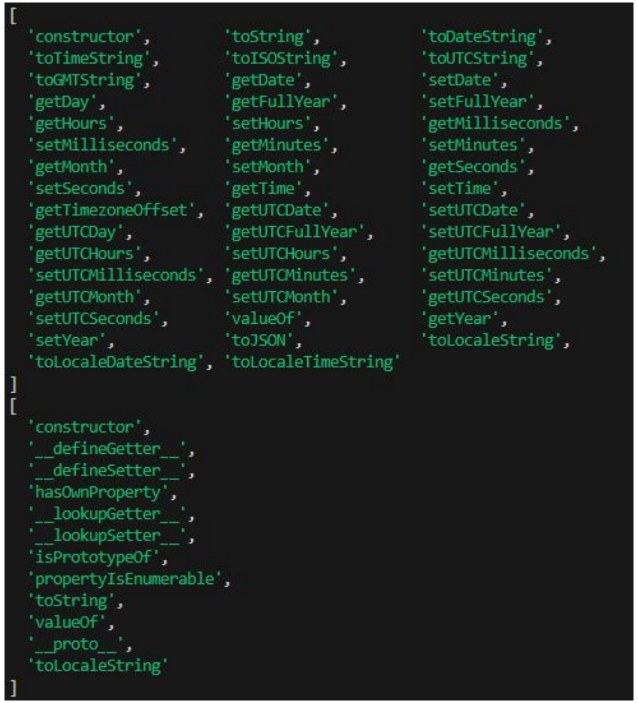
Native Prototypes: Date

Let’s walk up the prototype chain of Date and see the properties within:

```javascript
const myDate = new Date();
do {
  myDate = Object.getPrototypeOf(myDate);
  console.log(
     Object.getOwnPropertyNames(myDate));
} while (myDate);
```

Date

Object



```
[
  'constructor',        'toString',           'toDateString',
  'toTimeString',       'toISOString',        'toUTCString',
  'toGMTString',        'getDate',            'setDate',
  'getDay',             'getFullYear',        'setFullYear',
  'getHours',           'setHours',           'getMilliseconds',
  'setMilliseconds',    'getMinutes',         'setMinutes',
  'getMonth',           'setMonth',           'getSeconds',
  'setSeconds',         'getTime',            'setTime',
  'getTimezoneOffset',  'getUTCDate',         'setUTCDate',
  'getUTCDay',          'getUTCFullYear',     'setUTCFullYear',
  'getUTCHours',        'setUTCHours',        'getUTCMilliseconds',
  'setUTCMilliseconds', 'getUTCMinutes',      'setUTCMinutes',
  'getUTCMonth',        'setUTCMonth',        'getUTCSeconds',
  'setUTCSeconds',      'valueOf',            'getYear',
  'setYear',            'toJSON',             'toLocaleString',
  'toLocaleDateString', 'toLocaleTimeString'
]
[
  'constructor',
  '__defineGetter__',
  '__defineSetter__',
  'hasOwnProperty',
  '__lookupGetter__',
  '__lookupSetter__',
  'isPrototypeOf',
  'propertyIsEnumerable',
  'toString',
  'valueOf',
  '__proto__',
  'toLocaleString'
]
```

What happens when we call myDate.propertyIsEnumerable?

Properties, Fields, and this

We use the term **property** to define the key-value pairs of an object.

We use the term **field** to describe the properties of classes of JavaScript.

To refer to the current instance of the class from within the object itself, you can use the keyword `this`.

```
// username and password are properties
function Account(username, password) {
  this.username = username;
  this.password = password;
}

// username and password are fields
class Account {
  constructor(username, password) {
    this.username = username;
    this.password = password;
  }
}
```

Function Constructors

```
function Account(username, password) {
  this.username = username;
  this.password = password;
}

// Creation
const goose = new Account('goose123',
'secret123');
const randy = new Account('randy456',
'mypassword');

// Access properties
console.log(goose.username); // "goose123"
console.log(randy.password);   // "mypassword"
```

Constructors are ways to create an initialize objects, often with initial properties/fields.

Before JS had classes, it could use function constructors to build objects with properties.

Prototypal Inheritance

Because JavaScript has prototype and the prototype chain, we can use these properties to perform **prototypal inheritance**.

Prototypal inheritance makes use of the fact that we can look to other objects down the prototype chain for inherited properties and methods.

Prototypal Inheritance Examples

```
function Account(username, password) {
  this.username = username;
  this.password = password;
}
```

```
Account.prototype.describe = function() {
  console.log(`Username: ${this.username}`);
};

const goose = new Account('goose123', 'secret123');
goose.describe();
```

Adding prototype methods

Prototypal Inheritance Examples

```
function VIPAccount(username, password, level) {
  Account.call(this, username, password); // Inherit properties
  this.level = level; // add new properties
}

VIPAccount.prototype = Object.create(Account.prototype);
VIPAccount.prototype.constructor = VIPAccount;

VIPAccount.prototype.vipHello = function() {
  console.log("VIP access for " + this.username);
};


const randy = new VIPAccount('randy456', 'mypassword', 'gold');
randy.vipHello() // from VIPAccount
randy.describe() // from Account
```

Adding Inheritance steps

Limitations of Prototypes

Complex Syntax: Prototypal inheritance is verbose to setup and uses specific syntax for prototype chaining (`Object.create`, `.constructor` property)

No access modifiers: All properties and methods are public

Debugging and Readability: Developers used to proper OOP syntax from other languages (like Java) may find prototypal inheritance confusing and harder to debug.

Post-ES6: Classes

How did ES6’s classes simplify OOP for JavaScript developers?

Classes

ES6 finally introduced classes, allowing more readable and familiar OOP code.

Classes are blueprints for creating objects, complete with proper syntax for fields, access modifiers, constructors, and inheritance.

Class Syntax

```
class User {
  static species = 'Homo sapiens';        // Public field
  #password;                              // Private field

  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }

  greet() {                               // Instance method
    return `Hello, ${this.name}`;
  }

  get maskedPassword() {                  // Getter
    return '*'.repeat(this.#password.length);
  }

  set password(newPwd) {                  // Setter
    this.#password = newPwd;
  }

  static describe() {                     // Static method
    return 'All users are humans.';
  }
}
```

Classes feature:

- Class constructors
- Inheritance with `extends` and `super`
- Getters/Setters
- Public and private fields (ES2022+)
- Instance, Static, and Private (ES2022+) methods

Class Constructor

Classes provide a direct way to interface with the **constructor** method in order to perform initialization tasks.

Only 1 constructor can be defined.

Often, you will use super in subclasses to call the superclass's constructor.

```
class Account {
  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }
}

class VIPAccount extends Account {
  constructor(name, pwd, level) {
    super(name, pwd);
    this.level = level;
  }
}
```

Inheritance: extends and super

The keyword `extends` allows you to define a subclass that inherits from a superclass.

Additionally, you can use the `super` keyword to call the constructor of the superclass to say, set initial fields.

This greatly simplifies the syntax of inheritance over prototypal inheritance.

```
class Account {
  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }
}

class VIPAccount extends Account {
  constructor(name, pwd, level) {
    super(name, pwd);
    this.level = level;
  }
}

const vip = new VIPAccount(“randy”,
“mypassword”, “gold”)
```

Polymorphism with Classes

```
class Animal {
  speak() {
    console.log(`${this.name} makes a
sound.`);
  }
}

class RedPanda extends Animal {
  speak() { // Overrides Animal’s speak()
    console.log(`${this.name} says wah!`);
  }
}

const klein = new RedPanda("Klein");
klein.speak(); // Klein says wah!
```

Classes provide easier overriding of methods by just allowing you to define a method with the same name in the subclass, no need for prototype assignment.

Getters & Setters

```
class User {
  #password;

  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd;
  }

  get maskedPassword() {
    return '*'.repeat(this.#password.length);
  }

  set password(newPwd) {
    this.#password = newPwd;
  }
}

const user = new User("randy", "mypassword");
console.log(user.maskedPassword) // **********
user.password = "newpassword"
console.log(user.maskedPassword) // ***********
```

Getters and **Setters** improve upon encapsulation by restricting access and updating to certain fields through specific `get` and `set` methods.

These is classic OOP syntax that easier maintenance of class and object properties.

Public & Private fields (ES2022)

All properties we've used before were public- which is problematic for encapsulation and data integrity

Access modifiers for fields were introduced in ES2022 to fix this issue

```
class User {
  static species = 'Homo sapiens';
  #password; // Private

  constructor(name, pwd) {
    this.name = name;
    this.#password = pwd; // Private
  }

  get maskedPassword() {
    return '*'.repeat(this.#password.length);
  }
}

const user = new User("randy", "mypassword");
console.log(user.password) // undefined
console.log(user.maskedPassword) // **********
```

Instance vs. Static

Instance Fields/Methods

Instance fields/methods belong to the objects create from classes themselves (instances). To access fields and methods, you call them on the instances themselves.

```
class User {
  species = 'Homo sapiens';

  describe() {
     console.log("This is a user")
  }
}

const user = new User();
console.log(user.species)
console.log(user.describe())
```

Static Fields/Methods

Static fields/methods belong to the class itself, not the instances. They can be accessed by calling upon the class directly.

```
class User {
  static species = 'Homo sapiens';

  static describe() {
     console.log("This is a user")
  }
}

console.log(User.species)
console.log(User.describe())
```

CLASS ACTIVITY

https://edstem.org/us/courses/91614/lessons/159449/slides/936805

https://edstem.org/us/courses/91614/lessons/159449/slides/936806