Department of Computer Science
University of California, Los Angeles

Computer Science 144: Web Applications

Spring 2025
Prof. Ryan Rosario

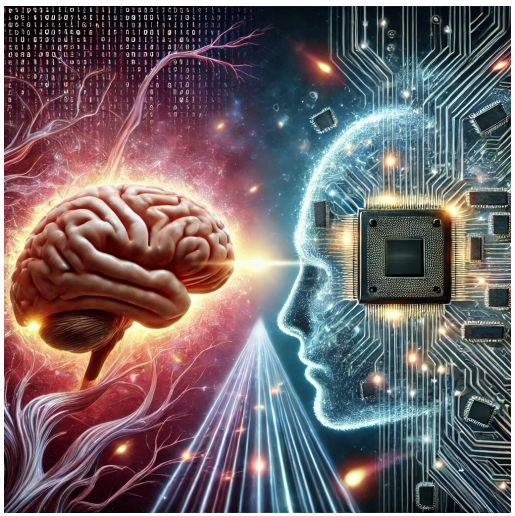
Lecture 6: April 16, 2025

Outline

1. JavaScript Memory Management
2. Asynchronous Programming
3. The Document Object Model (DOM)
4. Events

JavaScript Memory Management

We have talked about the **JavaScript** language, its **execution model** and its **compilation model**.



Another important aspect is the **memory management model** of JavaScript.

JavaScript Memory Management (contd.)

Historically, JavaScript has been slow and exhibited very poor performance.

In software engineering, we want code to be efficient and performance.

I also opine that some of us are traumatized by JavaScript's past performance that we want to be careful how we write our code.

So, a deep understanding of the lifecycle management mechanisms of stack and heap memory is crucial for writing efficient, memory-leak-free JavaScript code.

JavaScript Memory Management (contd.)

Memory is managed in the stack and the heap.

JavaScript Memory Management (contd.)

Simple data types like numbers. booleans, strings, undefined, and null are stored on the stack.

More complex types are stored on the heap with a variable pointing to them from the stack.

Function calls and execution contexts are also stored on the stack.

JavaScript Memory Management (contd.)

Things to know:

- Stack memory has an automatic cleaning mechanism. . . pop.
- Heap memory is highly dynamic and requires developer awareness of releasing references appropriately along with garbage collection.

The Stack

Primitive variables like numbers, booleans, strings, undefined, and null are stored in stack memory.

But, it is also used for temporary residents such as local variables and execution contexts during function calls.

```
let noise = "woof";
function speak() {
    let noise = "meow!";
}
greet();
```

Managing The Heap

Complex structures may be shared among multiple variables, or contain references within themselves, thus their lifecycle management becomes intricate.

1. Use local variables!
2. Release object references that are no longer in use to avoid unnecessary memory occupation.
3. Use tools like browser developer tools for memory analysis to regularly check and locate memory leaks.
4. Handle event listeners and timers carefully, ensuring they are cleaned up when no longer needed.
5. Understand and use weak data structures

Strategy: Release Object References

Once you know you no longer need to reference a particular object, set its value to `null`.

```
function longProcess() {
    let obj = new Object();
    // Do a bunch of stuff with obj
    // ...
    // Now we are done with obj
    obj = null; // Release reference to the object
}
```

This marks the object as "collectible" by the garbage collector. This is also important for closures and factories.

Strategy: Remove Event Listeners

If you add an event listener to an object, you should remove it when you no longer need it.

```
let button = document.createElement('button');
button.textContent = 'Click me';
document.body.appendChild(button);

button.addEventListener('click', handleClick);

// Properly remove elements and listeners
// when no longer needed
button.removeEventListener('click', handleClick);
document.body.removeChild(button);
```

Strategy: Weak Data Structures

Weak data structures are a special type of data structure that allows for automatic garbage collection of their contents.

- `WeakMap` — A collection of key-value pairs where the keys are objects and the values can be any value.
- `WeakSet` — A collection of unique objects, similar to a Set, but with weak references.

These structures do not prevent their keys or values from being garbage collected.

Strategy: Weak Data Structures (contd.)

In a standard `map`, if an object is used as a key, it **cannot** be garbage collected until the `map` is deleted, even if no other object references it. The `map` holds a **strong reference** to the object. This can cause memory leaks.

Instead, a `WeakMap` holds a **weak reference** to the key objects. It **will** be garbage collected if there are no other references to it. The value is also garbage collected. `WeakMaps` are also not interable.

Why would we ever use this?

Strategy: Weak Data Structures (contd.)

There are several good reasons to use `WeakMap`:

1. When storing metadata about DOM elements, such as event listeners or data attributes.
2. When creating caches for objects that may be removed from the DOM.

Strategy: Weak Data Structures (contd.)

For You to Think About: How would a `WeakSet` work and when would we want to use it?

Strategy: Weak Data Structures (contd.)

Map

```
const map = new Map();
const objKey = { id: 1 };
map.set(objKey, "Object Value");
map.set("name", "Ryan");
console.log(map.get(objKey));
console.log(map.get("name"));
console.log(map.has(objKey));
```

Set

```
const set = new Set();
set.add("apple");
set.add("banana");
set.add("apple");
console.log(set.has("banana"));
console.log(set.size);

for (const item of set) {
  console.log(item);
}
```

WeakMap

```
const weakMap = new WeakMap();
let obj = { id: 2 };
weakMap.set(obj, "Hidden Data");
console.log(weakMap.get(obj));
console.log(weakMap.has(obj));
obj = null;
```

WeakSet

```
const weakSet = new WeakSet();
let obj1 = { name: "Obj1" };
let obj2 = { name: "Obj2" };
weakSet.add(obj1);
weakSet.add(obj2);
console.log(weakSet.has(obj1));
obj1 = null;
```

Garbage Collection

Similar to Java, Python, Go etc. JavaScript uses **garbage collection** to manage memory in the heap.

Garbage Collection (contd.)

If you do not manage memory youself using the methods earlier, the garbage collector will do it for you, it can lead to unexpected issues, performance problems.

Languages uses a few different garbage collection strategies:

1. Reference counting
2. Mark and sweep
3. Generational collection

Garbage Collection (contd.)

Reference Counting is a technique where each object has a reference count that tracks how many references point to it.

When the reference count drops to zero, the object is eligible for garbage collection.

```
function referenceCountingExample() {
    let o1 = new Object();
    let o2 = o1;
    o1 = null;
    o2 = null;
}
```

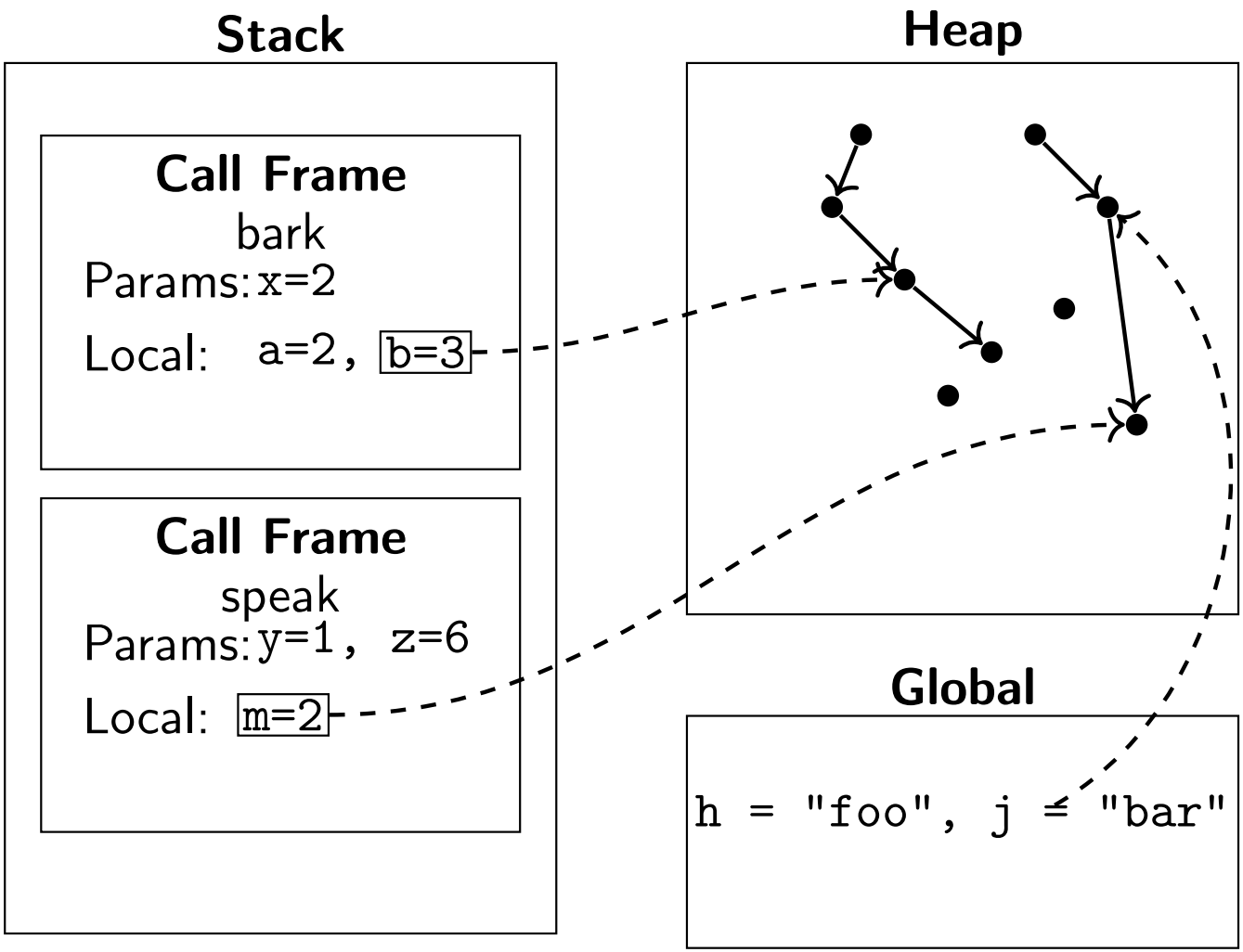
Garbage Collection (contd.)

Question: When would this not work well?

Garbage Collection (contd.)

Mark and Sweep is a more sophisticated approach. It requires understanding the object graph.

Root variables or objects are ones that are either **global** or are in a call frame on the stack.


Stack
Call Frame
bark
Params: x=2
Local: a=2, b=3
Call Frame
speak
Params: y=1, z=6
Local: m=2
Heap
Global
h = "foo", j = "bar"

Garbage Collection

Mark and sweep works as follows:

1. **Mark Phase:**
   - Start at each root variable.
   - Follow all possible references to other variables.
2. **Sweep Phase**
   - Go through the heap and check if any objects are marked.
   - If not, remove them.

Garbage Collection (contd.)

Question: Does this solve the earlier problem we discussed with reference counting?

Garbage Collection (contd.)

Generational Collection is a more advanced approach. It is based on the observation that most objects are short-lived.

The heap is divided into two regions based on usage, heuristics and system limits: **New Space** and **Old Space**.

All newly created objects are added to New Space. Once it has survived a certain number of garbage collection cycles, it is promoted to the Old Space region using a copy collector.

New Space is collected more frequently (Minor GC, every few MBs). Old Space is not, but uses **incremental mark-and-sweep** (Major GC) to clean up. Old Space is only GCed when pressure is high or heap size is large.

Garbage Collection (contd.)

It is actually much more complicated than this. There are many other regions of the heap:

1. New and Old Generations
2. Code Space (compiled machine code)
3. Map Space — stores layout and order of properties in objects and tracks changes
4. Large Object Space
5. Read-Only Space

Garbage Collection (contd.)

Some aspects of garbage collection are handled in separate threads, but the final object cleanup and pointer updates are done in the **main thread**.

If the main thread is blocked, this leads to performance problems.

And, the garbage collector itself can block the main thread!

For this class, let's assume that most coordination is done in the main thread and things like marking and sweeping are done in the background.

Garbage Collection (contd.)

Marking and sweeping is done in a few different ways:

1. **Parallel** — all threads work together to mark and sweep.
2. **Concurrent** — the main thread is not blocked, but the GC runs in the background.
3. **Incremental** — the GC runs in small chunks, allowing the main thread to run in between.

V8 references state that it uses a combination of these techniques.

Garbage Collection and You

In Node.js you can manually garbage collect by running it in a special mode. This does not work in production.

```
node --expose-gc script.js
global.gc()
```

Very few things are smarter than an UCLA Engineering student. . . but garbage collection is one of them.

Garbage Collection and You (contd.)

In this class we are focusing on the V8 engine. These are the strategies used by V8:

1. JavaScript **does not** use reference counting.
2. New Space uses **copying collection** and Old Space uses **mark-and-sweep** often optimized to be parallel, concurrent or incremental.

♪ Call Me (Back) Maybe? ♫

With traditional programming, all actions are **synchronous**. We must wait for each line to fully execute before moving on. For example:

```
function sendPicture(user_id) {
    user = db.findOne({userid: user_id});
    picture = fs.readFile(user.avatar);
    socket.write(picture);
}
```

Too much blocking!

♪ Call Me (Back) Maybe? ♫ (contd.)

JavaScript is single-threaded and we do not have the patience to wait for all of these actions to take place.

Instead, we fire off a function call that **returns immediately** and does something in the *background*.

Later on, when the long-running function is done, the main thread can do something with the result.

This something is called a **callback** and it is executed on the main thread when there is no other work to do.

♪ Call Me (Back) Maybe? ♫ (contd.)

A callback is a function that is executed at a later time, often in response to an event or condition.

This condition can be as simple as some task completing, or a timeout.

♪ Call Me (Back) Maybe? ♫ (contd.)

Let's go back to Hell's Kitchen and check up on Chef Gordon Ramsay.

Chef is prepping dinner.

- He needs to bake a cake that takes 30 minutes in the oven.
- While the cake is baking, he wants to chop a steak and plate the dish.

Now, Gordon is only one chef (like JavaScript is single-threaded). He can't just stand there waiting for the cake.

♪ Call Me (Back) Maybe? ♫ (contd.)

Without a Callback...

Chef stands there, like a donkey, watching the cake bake.

♪ Call Me (Back) Maybe? ♫ (contd.)

With a Callback: Chef tells his sous chef Christina:

"I just put a cake in the oven. ***Set a timer for 30 minutes, when the timer goes off, come get me so I can frost it."***



He then continues chopping the steak etc.

♪ Call Me (Back) Maybe? ♫ (contd.)

When the timer goes off:

- Christina alerts Chef that the cake is done.
- Chef frosts the cake.
- He then moves on to his next tasks.

This is an example of an asynchronous task for concurrency which uses a callback.

♪ Call Me (Back) Maybe? ♫ (contd.)

Chef's actions are the **main thread**.

Chef putting the cake in the oven is **synchronous but quick**.
Him giving the command to Christina to set the timer and return (**registering a callback**) is also synchronous and quick. He tells her he needs to frost the cake when it is done (callback).

The baking of the cake and Christina watching it is **asynchronous**. It happens in the background with respect to Chef.

The timer goes off, Christina notifies Chef that the cake is done so he can frost it.

When is available he then frosts the cake (executes the callback).

♪ Call Me (Back) Maybe? ♫ (contd.)

In terms of the runtimes:

1. Chef is again the main thread in the JavaScript runtime.
2. He pushes a function (insert the cake) to the stack and pops it off immediately (when he is done).
3. He tells Christina to physically set a timer on the oven. He pushes a function to the stack which is immediately popped. (`setTimeout()`)
   - The oven is the **Web API** (browser runtime). It handles the timing and baking. It does the background work.
   - Christina herself is the **callback queue**. She waits for Chef to be free of tasks and delivers the notification (**callback**), if any.
4. The timer goes off and Christina pushes the registered callback ("frost the cake") to her queue of messages to chef.
5. When Chef has a down moment (empty stack), she delivers the callback to him.
6. Chef pushes the callback "frost the cake" to the stack, executes it, and pops it off when he is done.

♪ Call Me (Back) Maybe? ♫ (contd.)

It is important to note that while all of this is going on, only Chef completes important work. Everyone/thing else is waiting or doing background tasks.

♪ Call Me (Back) Maybe? ♫ (contd.)

The cast:

Main thread — JS runtime: call stack, heap

Browser runtime: event loop, task queue

Browser runtime: Web APIs (e.g., timers, fetch)

The callback (executed later)

♪ Call Me (Back) Maybe? ♫ (contd.)

A function with a callback looks like the following. Note that all of this is with respect to Chef (main thread) and nobody else:

```
function frostCake() {
    console.log("Frosting the cake!");
}

// Register the callback, send Christina on her way.
setTimeout(frostCake, 1800000);
// Chef, Christina and the oven are doing their thing.
console.log("Chopping and baking...");
// Chef chops
// 30 minute later, if the call stack is free:
// Christina tells Chef the cake is done.
frostCake(); // is executed
```

♪ Call Me (Back) Maybe? ♫ (contd.)

This is syntatically equivalent to:

```
setTimeout(() => {  console.log("Frosting the cake!"); }, 1800000);
```

♪ Call Me (Back) Maybe? ♫ (contd.)

This can really be boiled (hehe) down to:

1. Go off and do something, this is what I want to do with the result.
2. When you're done, let me know.
3. When I am available, I will do that something with the result.

♪ Call Me (Back) Maybe? ♫ (contd.)

Another example: File download and processing

```
function downloadFile(filename, callback) {
  console.log(`Downloading ${filename}...`);

  // Let's pretend the file is already available (no delay)
  const fileData = `Contents of ${filename}`;

  // Immediately invoke the callback with the data
  callback(fileData);
}

function processFile(data) {
  console.log("Processing file:");
  console.log(data);
}

// Call it
downloadFile("lecture6.pdf", processFile);
```

♪ Call Me (Back) Maybe? ♫ (contd.)

Please do not forget the point of this.

We do not want to wait (block) while the file is downloaded and processed, we want to do other work.

We will process the file and do something with **the result** some time later when the main thread has a down moment.

♪ Call Me (Back) Maybe? ♫ (contd.)

So what happens if we do not use a callback?

```
function noop() {}
downloadFile("lecture6.pdf", noop);
```

We do not get any result back. There is no callback.

♪ Call Me (Back) Maybe? ♫ (contd.)

"Hey, won't you call me back?
'Cause I've been waiting for a text back."

– *She Plays Bass*, Beabadoobee

Processing Multiple Callbacks

Let's return to the first example where we:

1. Look up a user by their user ID
2. Read the path to the user's avatar from the `user` object
3. Load the avatar image from the file system
4. Send the image over the network over a socket

Each one of these takes time, and each can be executed asynchronously using callbacks.

But it is very difficult to read. This is often called **callback hell**.

```
function sendPicture(user_id, finalCallback) {
  lookupUser(user_id, (err, user) => {
    readFromFileSystem(user, (err, picture) => {
      sendOverSocket(picture, finalCallback);
    });
  });
}

function lookupUser(user_id, callback) {
  db.findOne({ userid: user_id }, callback); // (err, user)
}

function readFromFileSystem(user, callback) {
  fs.readFile(user.avatar, callback); // (err, picture)
}

function sendOverSocket(picture, callback) {
  socket.write(picture, callback); // (err)
}

// Usage
sendPicture("user123", () => {
  console.log("Process complete");
});
```

Processing Multiple Callbacks

Instead we could nest them. This is sometimes referred to as the **Pyramid of Doom** or **Callback Hell**.

```
function sendPicture(user_id) {
  db.findOne({userid: user_id}, (err, user) => {
    fs.readFile(user.avatar, (err, picture) => {
      socket.write(picture, () => {
        console.log("process complete");
      })
    })
  })
}
```

It looks better, but there is no error handling!

Processing Multiple Callbacks (contd.)

Just looking at this code gives me anxiety.

There must be a better way...

♪ Promises, Promises ♫

To solve this mess, ES5 gave us the concept of **promises**, and ES2017 gives us `async/await` which we will get to in a bit.

♪ Promises, Promises ♫ (contd.)

A **promise** is often described as a placeholder for a value that is not yet available, but **will** be resolved in the future.

You can think of it as a “receipt" or “IOU" for a result that hasn’t arrived yet.

♪ Promises, Promises ♫ (contd.)

For example, when you make a purchase on Amazon:

- You don't get the product immediately
- But you get a tracking number for the order (the Promise)
- When the product arrives, the tracking number and order becomes "fulfilled"
- If it's lost or fails to ship, the promise becomes "rejected"

♪ Promises, Promises ♫ (contd.)

Below, `myProm` is a promise returned.

```
// MongoDB returns a promise of a value, not a value itself
let myProm = db.findOne({userid: user_id});
```

We can then specify the next computation using `then`

```
myProm.then(fullfillCallback, rejectCallback);
```

If the call to `findOne` does not result in an error, we then execute the `fulfillCallback`.If an error is raised, we call `rejectCallback` with an error object.

The call to `then()` returns a new promise.

♪ Promises, Promises ♫ (contd.)

To make it more readable and understandable, we can chain promises instead.

```
function sendPicture(user_id) {
  db.findOne({userid: user_id})
    .then(user => fs.promises.readFile(user.avatar)) // must return a promise
    .then(avatar => new Promise((resolve, reject) => {
      socket.write(avatar, (err) => {
        if (err) reject(err);
        else resolve();
      });
    }))
    .then(() => console.log("process complete"))
    .catch(err => {
      console.error("Something went wrong:", err);
    });
}
```

♪ Promises, Promises ♫ (contd.)

Note that `sendPicture` returns immediately as it's processing is asynchronous. Each action in `then` waits until the previous promise is resolved or rejected before executing.

♪ Promises, Promises ♫ (contd.)

Question: Wait a minute. So if we have to wait for a promise to be resolved, this is synchronous, right?

♪ Promises, Promises ♫ (contd.)

A promise is *settled* when it's either *fulfilled* (is resolved) or *rejected* (error).

♪ Promises, Promises ♫ (contd.)

A promise may be rejected, but we do not need to set an error handler for each `then()`.

```
function sendPicture(user_id, errorHandler) {
        db.findOne({userid: user_id});
          .then(user => fs.readFile(user.avatar))
          .then(avatar => socket.write(avatar))
          .then(() => console.log("process complete"))
          .catch(errorHandler);
}
```

If a rejection is not handled, it execution immediately jumps to the nearest `catch` statement.

♪ Promises, Promises ♫ (contd.)

We can make our own promises:

```
new Promise((error, reject) => {
        ...
        if (successCond) {
                resolve(val);
        } else {
                reject(err);
        }
});
```

But, never make a promise you cannot keep.

`async` and `await`

`async` and `await` are syntactic sugar for promises.

- `async` can be placed before a function declaration. It tells JavaScript that this function will **return a promise**.
- `await` can be placed before something that returns a promise. It tells JavaScript to wait for the promise to resolve before continuing *outside* of a chain.

async and await (contd.)

Here is a version of the previous example using `async` and `await`:

```
async function sendPicture(user_id) {
  try {
    const user = await db.findOne({ userid: user_id });
    const avatar = await fs.promises.readFile(user.avatar);
    await socket.write(avatar);
    console.log("done");
  } catch (err) {
    console.error(err);
  }
}
```

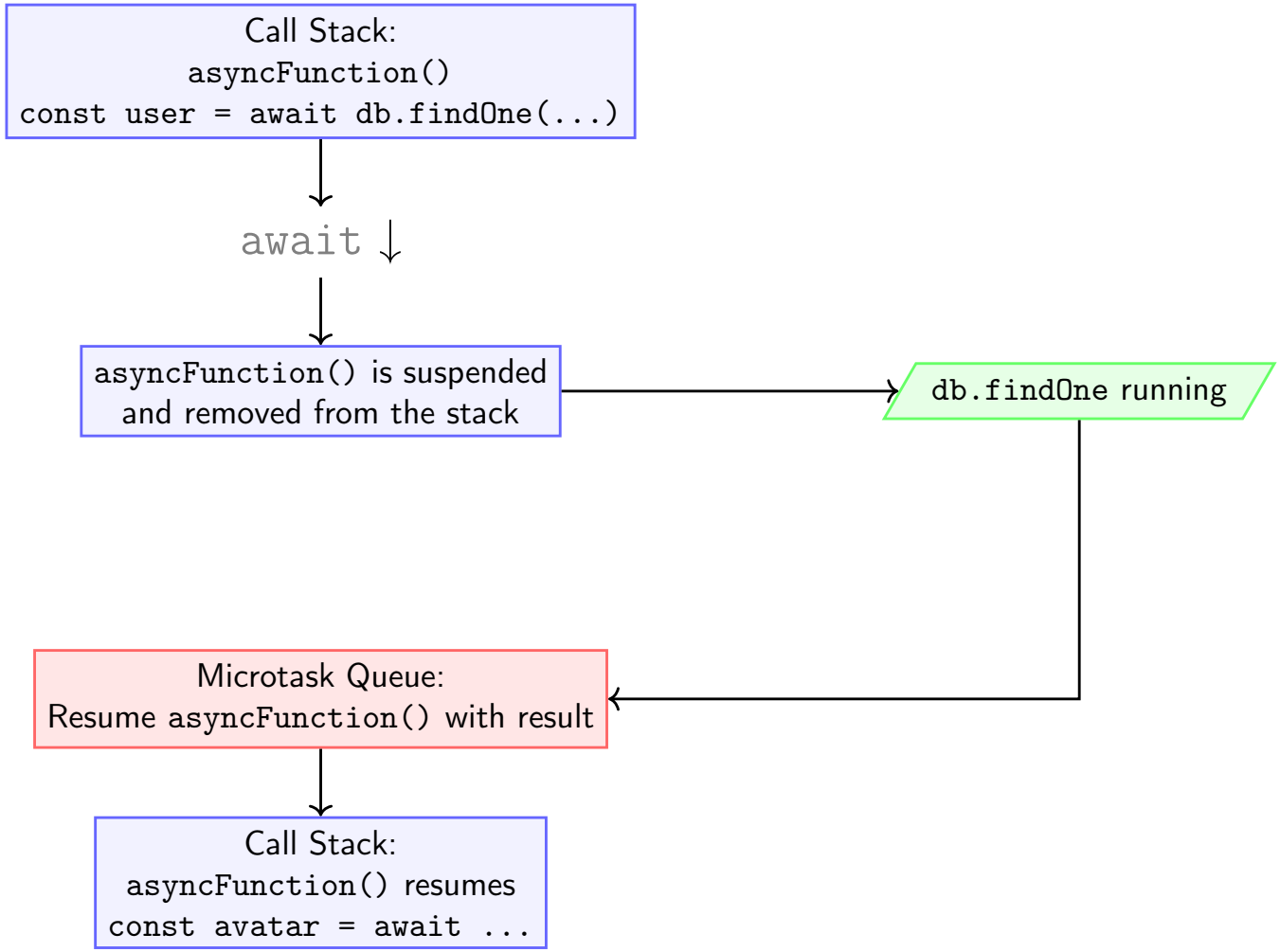
`await` can only be used inside an `async` function. Know that `sendPicture` now returns a promise.

async and await (contd.)

We no longer have that `catch` statement at the end of the chain, so we must insert a `try/catch` block.

```
async function sendPicture(user_id) {
  try {
    const user = await db.findOne({ userid: user_id });
    const avatar = await fs.promises.readFile(user.avatar);
    await socket.write(avatar);
    console.log("process complete");
  } catch (err) {
    console.error("Something went wrong:", err);
  }
}
```

async and await (contd.)



Javascript to DOM

We have discussed Javascript quite a bit now, and still have a lot to discuss about it.

Now we will focus on how to connect HTML and Javascript as an important building block to build a web application.

Document Object Model (DOM)

HTML is exposed to Javascript through the **D**ocument **O**bject **M**odel (DOM) which is a tree containing all of the structure and content on the web page.

We can use Javascript to access/read or modify this tree structure and the nodes in it.

The global object representing the browser window is the `window`. `window.document` is the root of the DOM. We usually only use a root object `document` though.

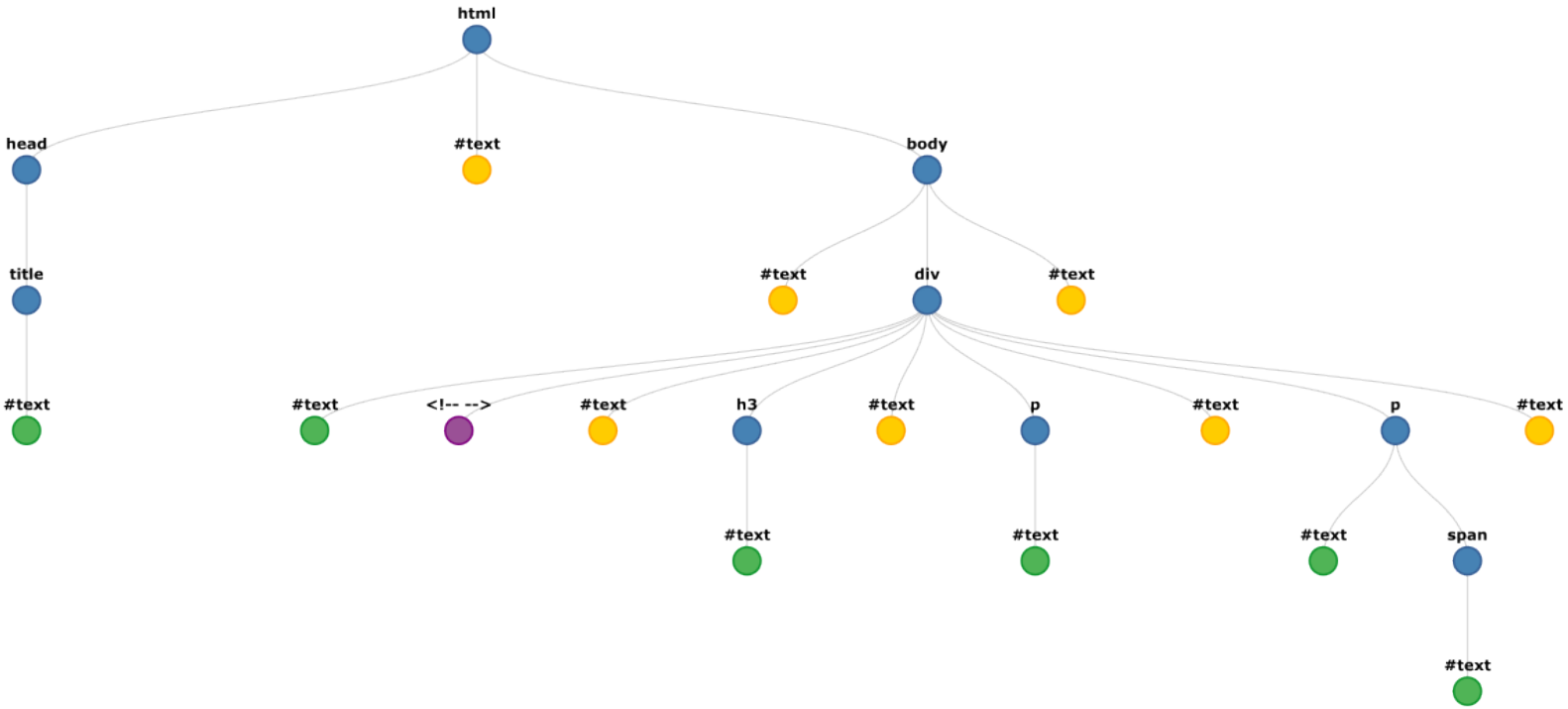
Document Object Model (DOM) (contd.)

We will use both a real as well as a small toy page that looks like the following:

```
<!DOCTYPE html>
<html lang="en">
  <head><title>My Site</title></head>
  <body>
    <div>
      <!-- useless comment -->
      <h3>A Heading</h3>
        <p>This is a paragraph.</p>
        <p>This is another <span style="display: none">paragraph.
          </span></p>
    </div>
  </body>
</html>
```

Document Object Model (DOM) (contd.)

The DOM for this simple page is below:



Document Object Model (DOM) (contd.)

You can see at least two different node types here. There are several node types in the DOM tree. The three most common:

1. **Element**: An HTML element. Every tag creates an element node.
2. **Text**: The text enclosed in an element (between tags). It becomes a child of the element.
3. **Attribute**: Attribute of an element. It is associated with the Element node but not a child of it.

Navigating/Traversing the DOM

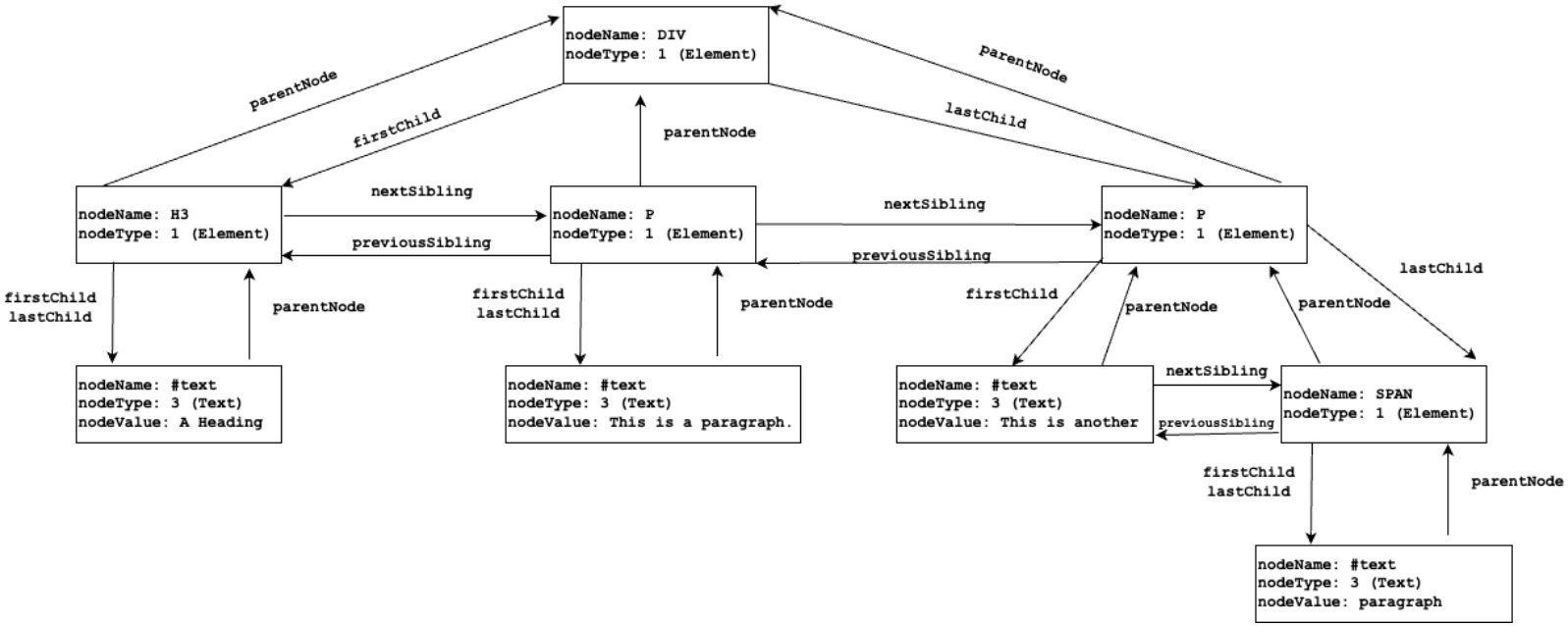
In order to access and modify a web page via the DOM, we need to find elements.

We can start at the root, called on `document`: `firstChild`, `lastChild`, `childNodes`, `previousSibling`, `nextSibling`, `parentNode`.

Or, we can start our search at an arbtirary element in the DOM:

```
const targetElement = document.getElementById('some_id');
const targetElements = document.getElementsByClassName('some_class');
const targetTags = document.getElementsByTagName('some_tag');
const element = document.querySelector('query');
const elements = document.querySelectorAll('query');
```

Node Relations in our Simple Page



Note that this diagram focuses only on the `<div>` element and also ignores comment and text nodes inserted by the browser.

Live Collections

The concept of a *live* collection is interesting.

`getElementsByTagName(...)` and `getElementsByClassName()` returns an `HTMLCollection` which is live.

This means the collection is updated automatically when the DOM changes.

```
let bodychildren = document.body.childNodes;
bodychildren.length;          // returns?

// lets us add a child to the body (embed it))
document.body.appendChild(document.createElement("P"));
bodychildren.length;     // returns?
```

What Do We Do With Nodes?

Once we find the elements we are looking for, there are several things we can do with them. The most common are:

- `textContent`: the text content of a node and its descendants as they appear in the HTML file.
- `innerText`: the *rendered* text content of a node and its descendants.
- `innerHTML`: HTML describing all descendants
- `outerHTML`: similar to `innerHTML` but includes HTML for the current node
- `getAttribute() / setAttribute()`

What Do We Do With Nodes?: Modify Them

We can use these properties to *modify* nodes in the DOM as well:

```
element.innerHTML = "<p>This is <em>new</em> HTML that replaces other HTML.";
```

Changing `innerHTML` can also be used to change the structure of DOM, by automatically creating nodes from HTML elements etc.

Why? We want to change content based on some event.

What Do We Do With Nodes?: Modify Them (contd.)

We can also modify attributes.

```
// change a class to affect styling
element.className = "error";

// Should never update styles directly. Use clases.
element.style.color = "#00ff00";          // No!
```

Why? We may need to add or remove CSS classes or IDs, or disable a form submission button etc. based on an event.

What Do We Do With Nodes?: Create/Delete Them

We can use methods on elements to **create new nodes** in the DOM or **delete unneeded nodes**. To add an element, we first create it, then add it to the DOM:

```
// create the node
const newElement = document.createElement("P");
const newText = document.createTextNode("Go Bruins!");

// add it to the DOM
parent.appendChild(newElement);  // or newText
// or, insert it in a specific place.
parent.insertBefore(newElement, existingSibling);
```

You can also use cloneNode() or replaceChild() for convenience.

What Do We Do With Nodes?: Create/Delete Them (contd.)

And to delete a node...

```
// Remove via the parent node.
theParentNode.removeChild(oldNode);

// or directly
oldNode.remove();
```

Event Driven Programming

Programming with the DOM and Javascript uses an **event-driven** model.

- We define the event we want to track and what elements trigger it.
- We attach handlers to specific DOM nodes that **listen and wait** for events.
- The browser takes appropriate action when the event arrives.

Event Driven Programming (contd.)

Every DOM element can be associated with one or more **events**. There are several classes of events. Here are some of them:

- mouse and keyboard
- HTML frames, forms and DOM mutation
- user interface
- progress
- touch
- animation
- clipboard and drag n' drop
- gestures
- payments
- printing
- text selection

Event Driven Programming (contd.)

Some of the more comment *events* are below. You are probably familiar with them:

- click
- mouseover
- mousemove
- mouseout
- dblclick
- keydown
- keyup
- submit

Event Driven Programming (contd.)

We can set an **event handler** on an object/element for each **event** that it needs to react to. (At least) *one event handler, per event, per element.* The event handler is a **function** that is **invoked** when the event is **triggered**.

When an event is triggered on an object (*the event target*) the associated **callback function** (*event handler*) is called (*invoked*).

Event Driven Programming (contd.)

When we want to handle an event on an element, we add create an event listener and handler.

1. Write an event handler that does something. It could change the DOM, write to a database etc. This is just a Javascript function that takes one parameter: `event`.
2. Add an event listener that binds the element to an event and event handler.

Event Driven Programming (contd.)

Below is an example.

```
<!DOCTYPE html>
<html lang="en">
<head><title>Event Example</title>
        <link rel="stylesheet"type="text/css" href="colors.css">
<script>
function getRandomInt(max) {
  return Math.floor(Math.random() * max);
}
function changeColor(event) {
        let colors = [ "red", "green", "blue", "purple", "orange", "pink" ];
        document.body.className = colors[getRandomInt(6)];
}
document.body.addEventListener("click", changeColor);
</script>
</head>
<body>Click me!</body></html>
```

But, there is something wrong with the code. What is it? How do we fix it?

Event Driven Programming (contd.)

We saw that the callback function takes one parameter, this is the object `event`. For debugging we have access to two properties.

1. `event.target`: the target element including its HTML
2. `event.type`: the event type (e.g. `click`)

Event Bubbling

Events "bubble up" through the DOM all the way up to the `document`, or even `window`.

An event, *even if it is handled by an event handler*, will be bubbled up the DOM **so the ancestors can act on it as well**.

If we want to stop events from bubbling up past a certain element, we use the following in the event handler:

```
event.stopPropagation();
```

References for Lecture 6

- Memory Management in JavaScript
- W3C DOM Technical Reports
- W3C DOM Level 3 Events
- JavaScript and DOM Events
- Common CSS Property Names in JavaScript