EECS 368
Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 2 due (today): 11:59 PM, Monday, September 19
- Assignment 3 due: 11:59 PM, Monday, October 3

Any Questions?

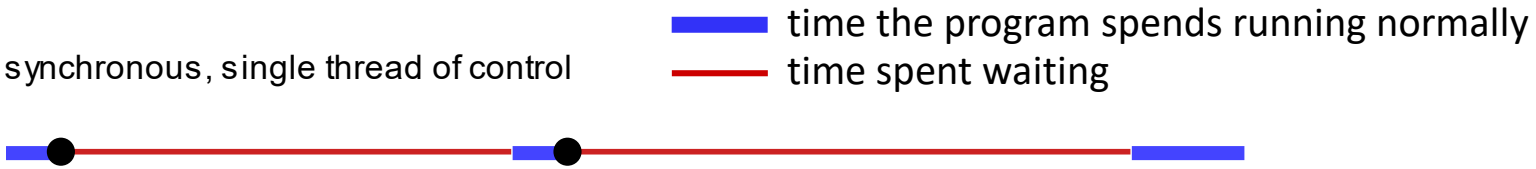
In-Class Problem Solution

- 10-(9-16) In-Class Problem Solution.pptx

Any Questions?

Synchronous Programming

- Synchronous programming is what you did in EECS 168 and 268, and in 368 so far.
- In a synchronous environment, where the request function returns only after it has done its work, the easiest way to perform this task is to make the requests one after the other.
- This has the drawback that the second request will be started only when the first has finished.
- The total time taken will be at least the sum of the two response times.

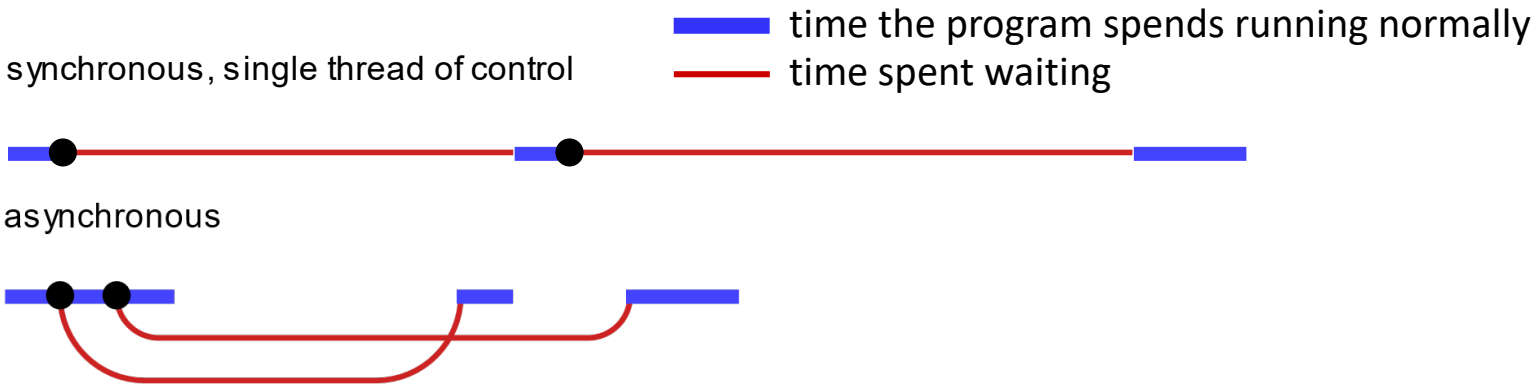


- This is fine, when you are running a program in one computer and everything you need is on that computer.
- But, it is not fine, if you have to go to another computer across a network to get something, like in a client-server application.

Asynchronous Programming

- In the asynchronous model, starting a network action conceptually causes a split in the timeline.
- The program that initiated the action continues running, and the action happens alongside it, notifying the program when it is finished.

- Another way to describe the difference is that waiting for actions to finish is:
  - implicit in the synchronous model
  - explicit, under our control, in the asynchronous one



- JavaScript environments typically implement this style of programming using callbacks, functions that are called when the actions complete.

Callbacks

- In JavaScript, a callback is a function passed into another function as an argument to be executed later.
- Suppose that you have the following numbers array:

```
let numbers = [1, 2, 4, 7, 3, 5, 6];
```

- To find all the odd numbers in the array, you can use the filter() method of the Array object.
- Remember, the filter() method creates a new array with the elements that pass the test implemented by a function.
- The following test function returns true if a number is an odd number:

```
function isOddNumber(number) {
    return number % 2;
}
```

- Now, you can pass the isOddNumber() to the filter() method:

```
const oddNumbers = numbers.filter(isOddNumber);
console.log(oddNumbers); // [ 1, 7, 3, 5 ]
```

- In this example, the isOddNumber is a callback function.
- When you pass a callback function into another function, you just pass the reference of the function i.e., the function name without the parentheses ().

Callbacks

- To make it shorter, you can use an anonymous function as a callback:

```
let oddNumbers = numbers.filter(function(number) {
    return number % 2;
});
console.log(oddNumbers); // [ 1, 7, 3, 5 ]
```

- In ES6, you can use the arrow functions:

```
let oddNumbers = numbers.filter(number => number % 2);
```

- ES6 refers to version 6 of the ECMA Script programming language.
- It is a major enhancement to the JavaScript language, and adds many more features intended to make large-scale software development easier.
- ECMAScript, or ES6, was published in June 2015.
- It was subsequently renamed to ECMAScript 2015.

Callbacks & Web Browsers

- When you use the JavaScript on web browsers, you often listen to an event e.g., a button click and carry out some actions if the event occurs.
- Suppose that you have a button with the id btn:

```
<button id="btn">Save</button>
```

- To execute some code when the button is clicked, you use a callback and pass it to the addEventListener() method:

```
function btnClicked() {
  // do something here
}
let btn = document.querySelector('#btn');
btn.addEventListener('click',btnClicked);
```

- The btnClicked function in this example is a callback.
- When the button is clicked, the btnClicked() function is called to carry out some actions.

Any Questions?

Event Handlers

- Some programs work with direct user input, such as mouse and keyboard actions.
- That kind of input isn't available as a well-organized data structure.
- These type of input (e.g., mouse and keyboard actions) are called events.
- Events come in piece by piece, in real time, and the program is expected to respond to them as they happen.
- Event handlers make it possible for a JavaScript program to detect and react to events happening in a web page.

Event Handlers

- Browsers do this by allowing us to register functions as handlers for specific events.
- The window.addEventListener method is how a function is registered as a handler.

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

- The window binding refers to a built-in object provided by the browser.
- It represents the browser window that contains the document.
- Calling its addEventListener method:
  - registers the second argument to be called (callback function)
  - whenever the event described by its first argument occurs

Any Questions?

Events and DOM Nodes

- Each browser event handler is registered in a context.
- Event listeners are called only when the event happens in the context of the object they are registered on.

- In the previous example we called addEventListener on the window object to register a handler for the **whole** window.

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

Events and DOM Nodes

- We can use the querySelector method to narrow the scope down to a single element in the window.

Recall:

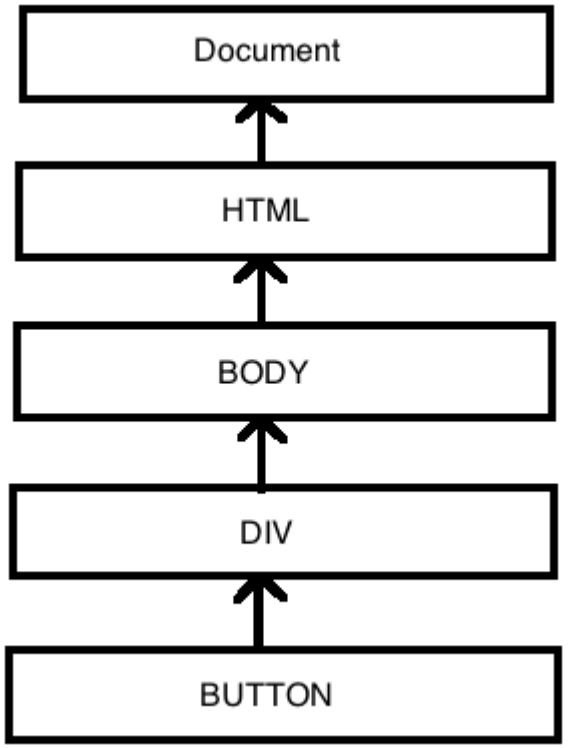
- The querySelector method is useful if you want a specific, single element.
- It takes a selector string and returns a node containing the element that it matches.
- For example, to handle the user clicking on a specific button in a window:

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

- This example attaches a handler (arrow function) to the button node.
- Clicks on the button cause that handler to run.
- But clicks on the rest of the document do not.

Event Propagation (or Bubbling)

- For most event types, handlers registered on nodes with children will also receive events that happen in the children.
- If a button inside a paragraph is clicked, event handlers on the paragraph will also see the click event.
- But if both the paragraph and the button have a handler, the more specific handler—the one on the button—gets to go first.
- The event is said to propagate outward, from the node where it happened to that node's parent node and on to the root of the document.
- Finally, after all handlers registered on a specific node have had their turn, handlers registered on the whole window get a chance to respond to the event.

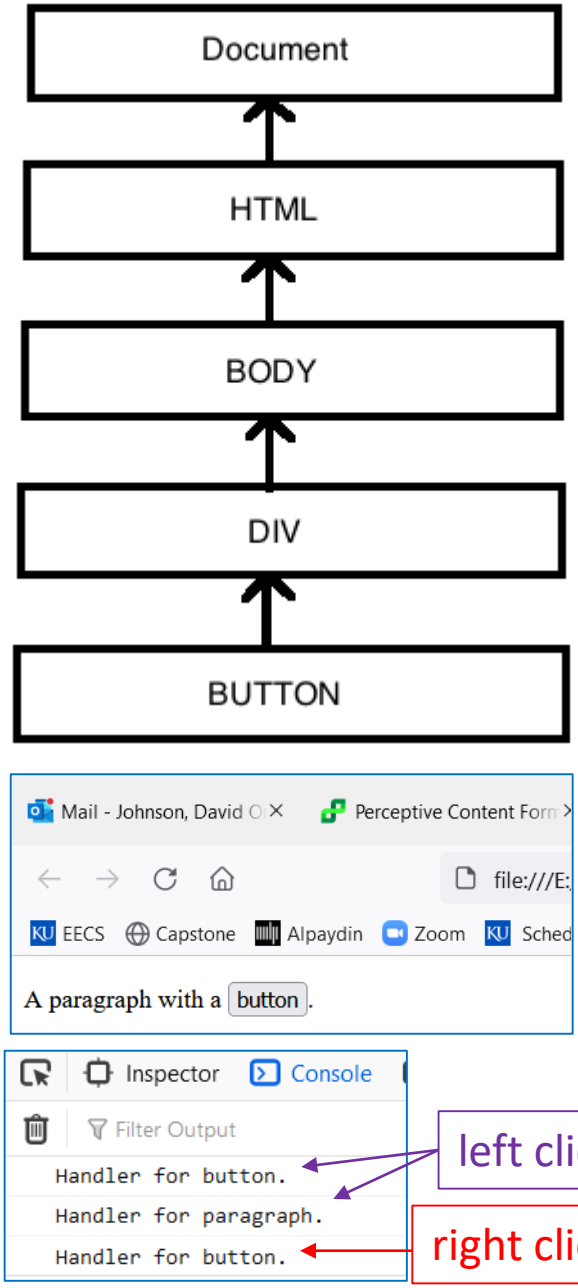


Event Propagation (or Bubbling)

- At any point, an event handler can call the stopPropagation method on the event object to prevent handlers further up from receiving the event.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

- This example registers "mousedown" handlers on both a button and the paragraph around it.
- When clicked with the right mouse button, the handler for the button calls stopPropagation, which will prevent the handler on the paragraph from running.
- When the button is clicked with another mouse button, both handlers will run.



Any Questions?

Event Objects

- Event handler functions are passed an argument: the event object.
- This object holds additional information about the event.
- For example, if we want to know which mouse button was pressed, we can look at the event object's button property.
- The information stored in an event object differs per type of event.
- The object's type property always holds a string identifying the event (such as "click" or "mousedown").

```
<button>Click me any way you want</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Left button");
    } else if (event.button == 1) {
      console.log("Middle button");
    } else if (event.button == 2) {
      console.log("Right button");
    }
  });
</script>
```

Event Types

- JavaScript supports a number of events:
  - Pressing a key fires "keydown" and "keyup" events.
  - Pressing a mouse button fires "mousedown", "mouseup", and "click" events.
  - Moving the mouse fires "mousemove" events.
  - Touchscreen interaction will result in "touchstart", "touchmove", and "touchend" events.
  - Scrolling can be detected with the "scroll" event.
  - Focus changes can be detected with the "focus" and "blur" events.
  - When the document finishes loading, a "load" event fires on the window.
- For more information on these events, see Chapter 15 of Eloquent JavaScript, or google it.

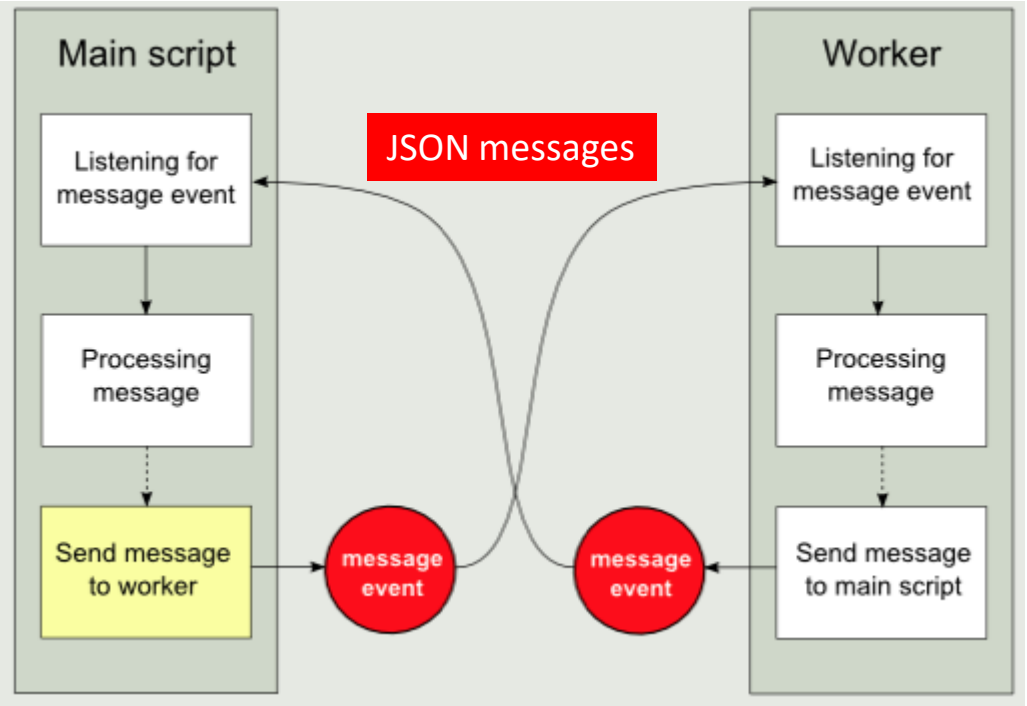
Any Questions?

Events and the Event Loop

- In the context of the event loop browser, event handlers behave like other asynchronous notifications.
- They are scheduled when the event occurs but must wait for other scripts that are running to finish before they get a chance to run.
- If the event loop is tied up with other work, any interaction with the page (which happens through events) will be delayed until there's time to process it.
- So if you schedule too much work, either with long-running event handlers or with lots of short-running ones, the page will become slow and cumbersome to use.

Web Workers

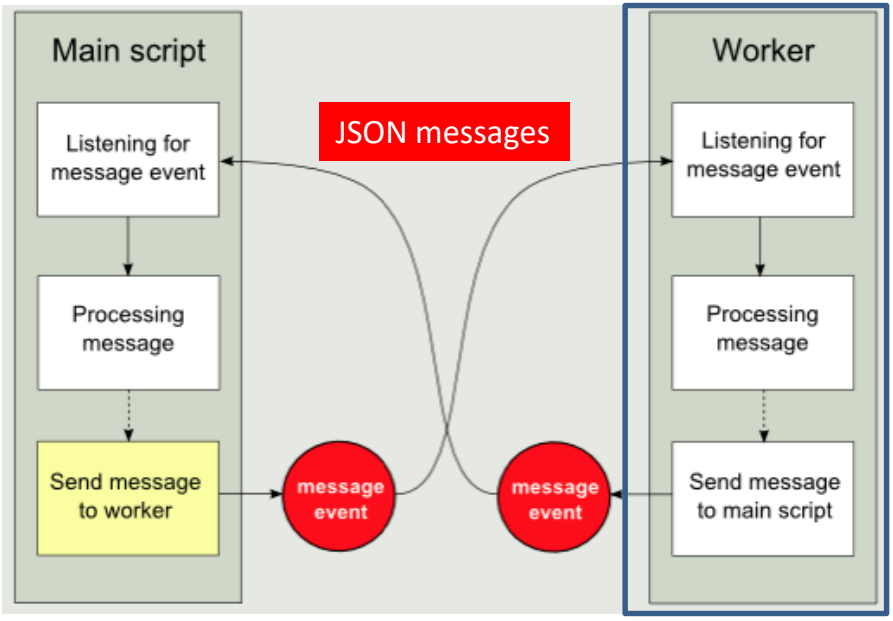
- For cases where you really do want to do some time-consuming thing in the background without freezing the page, browsers provide something called web workers.
- A worker is a JavaScript process that runs alongside the main script, on its own timeline.
- To avoid the problems of having multiple threads touching the same data, workers do not share their global scope or any other data with the main script's environment.
- Instead, you have to communicate with them by sending JSON messages back and forth.



Web Workers

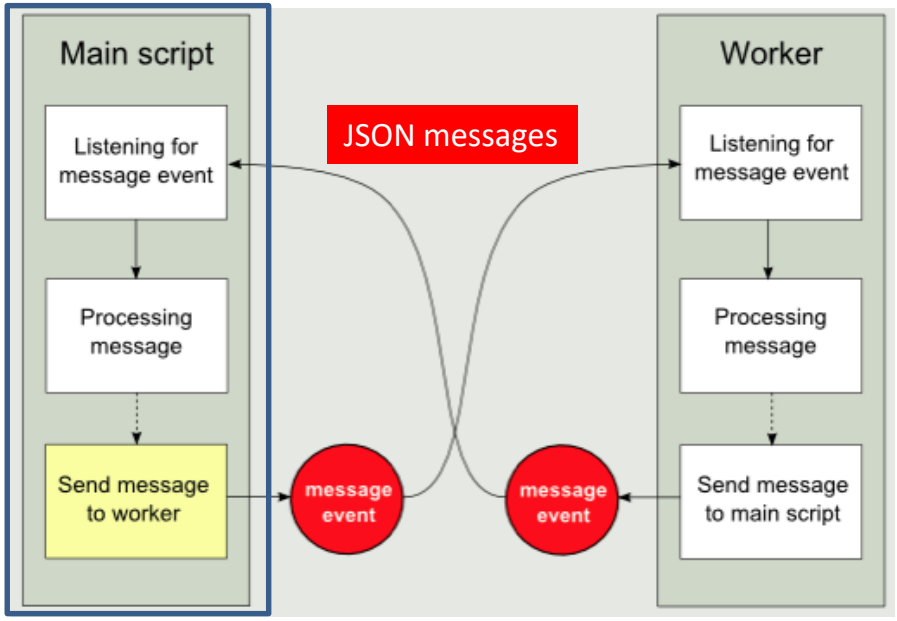
- Imagine that squaring a number is a heavy, long-running computation that we want to perform in a separate thread.
- We could write a file called code/squareworker.js that responds to messages by computing a square and sending a message back.
- Listening for JavaScript event = "message"
- "event" is a key word representing the event object for "message"; it is passed to anonymous callback arrow function
- Event.data is a property in the event object containing the message in JSON, in this example a number to be squared
  event.data * event.data squares number
- postMessage sends squared number back to web page in JSON

```
//This worker script is in code/squareworker.js
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```



Web Workers

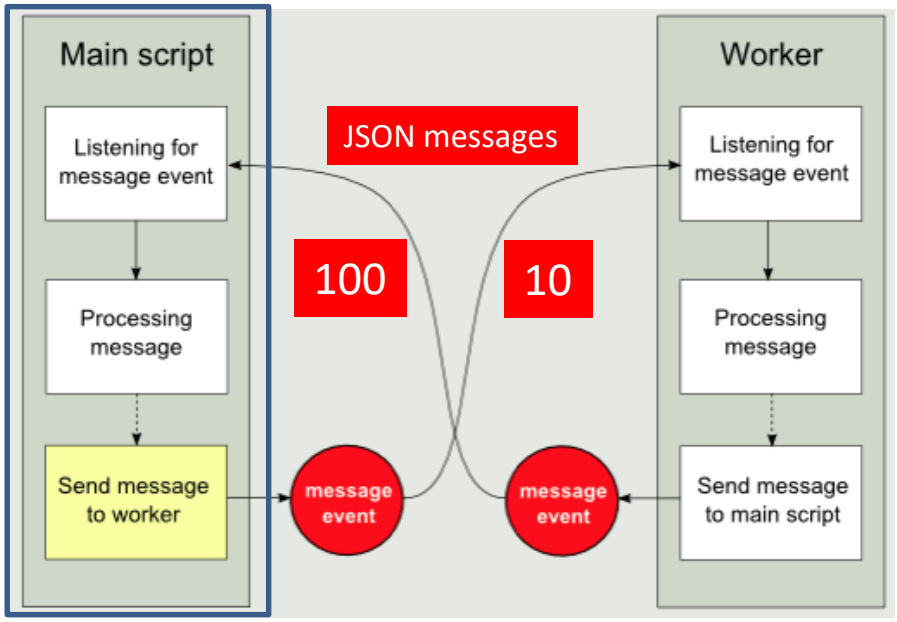
- This code in main script spawns worker and waits for message back from Worker as an event.
- "event" is a key word representing the event object for "message".
- It is passed to an anonymous callback arrow function.
- Event.data is property in event object containing message in JSON, in this example a number squared.




```
//This worker script is in code/squareworker.js
addEventListener("message", event => {
  postMessage(event.data * event.data);
});

//This code spawns a worker running that script
let squareWorker = new
Worker("code/squareworker.js");
squareWorker.addEventListener("message",
event => {
  console.log("The worker responded:",
event.data);
});
```

Web Workers




```
//This worker script is in code/squareworker.js
addEventListener("message", event => {
  postMessage(event.data * event.data);
});

//This code spawns a worker running that script
let squareWorker = new
Worker("code/squareworker.js");
squareWorker.addEventListener("message",
event => {
  console.log("The worker responded:",
event.data);
});

//This code tests the whole process
//Console output: The worker responded: 100
squareWorker.postMessage(10);
```

Any Questions?

Summary

- Asynchronous programming makes it possible to express waiting for long-running actions without freezing the program during these actions.
- JavaScript environments typically implement this style of programming using callbacks, functions that are called when the actions complete.
- A callback is a function passed into another function as an argument to be executed later.
- Event handlers make it possible to detect and react to events happening in our web page.
- Event handlers are callback functions.
- The document.querySelector and addEventListener methods are used to register such a handler.
- Most events are called on a specific DOM element and then propagate to that element's ancestors, allowing handlers associated with those elements to handle them.
- The stopPropagation method can be used to stop further propagation.
- When an event handler is called, it is passed an event object with additional information about the event.
- Each event has a type ("keydown", "focus", and so on) that identifies it.
- Event handlers are scheduled when the event occurs but must wait for other scripts that are running to finish before they get a chance to run.
- A worker is a JavaScript process that runs alongside the main script, on its own timeline, to handle time consuming tasks.
- The main script and worker communicate by sending JSON messages back and forth.

Any Questions?

In-Class Problem

Given:

- document.body.style.background changes the background color of a web page.
- document.body.style.background = "green" changes it to green
- document.body.style.background = "" changes it to transparent
- When a key on the keyboard is pressed, your browser fires a "keydown" event.
- When it is released, you get a "keyup" event.
- The key property of the event object holds a string that corresponds to the symbol that pressing that key would type.

```
<script>
//Add a comment here that describes what this
//JavaScript program does

//Add a comment describing what this block of
//code is doing
window.addEventListener("keydown", event => {
   if (event.key == "g") {
     document.body.style.background = "green";
   }
 });
//Add a comment describing what this block of
//code is doing
window.addEventListener("keyup", event => {
   if (event.key == "g") {
     document.body.style.background = "";
   }
 });
</script>
```