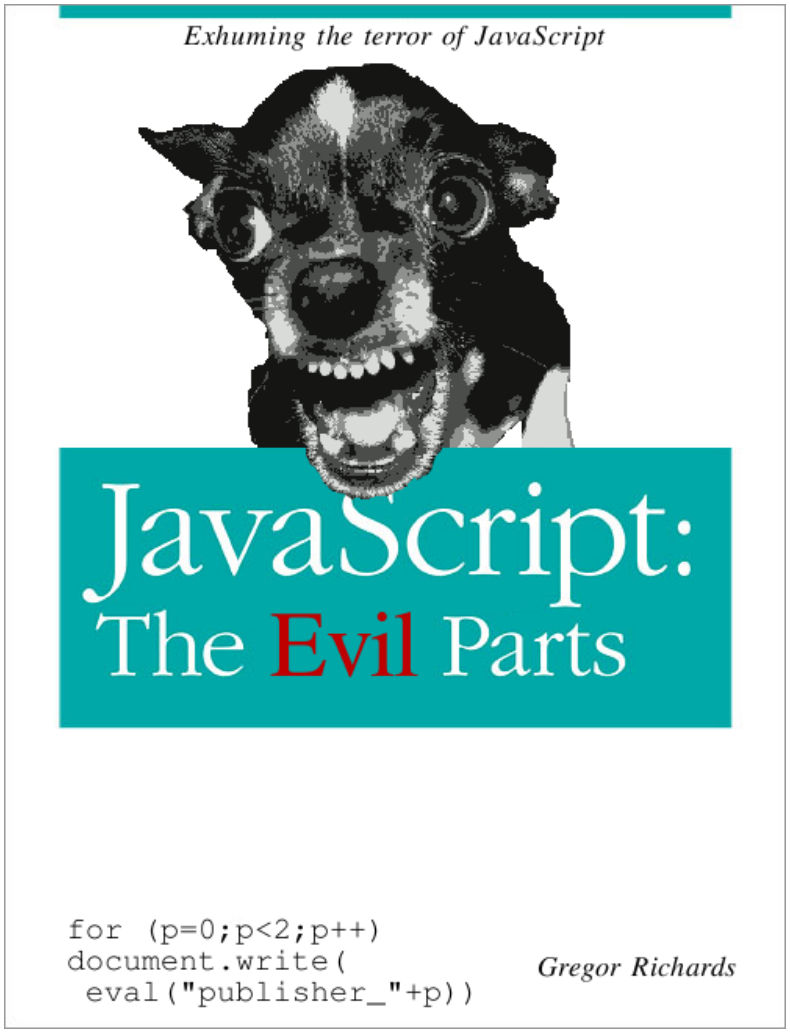
Client-side Dynamic Pages



Slides by Andrew DeOrio

Agenda

- Introduction to JavaScript
- Review
- Client-side dynamic pages
- JavaScript execution model: event-driven programming
- JavaScript data model
- Pitfalls (best and worst practices)

JavaScript is …

- "Python with C++ syntax"
- With more design flaws
- Only programming language that web browsers support
- Has nothing to do with Java
- Created in 1996 by Brendan Eich at Netscape … in 10 days

JavaScript history

- JavaScript AKA ECMA Script
- ES5 was the standard 2008 - 2015
  - Widely supported by web browsers
- ES6 was a major update
  - Classes, import, arrow functions, Promises and much more
  - Mostly supported by most web browsers
- As of fall 2020, we're up to ES10

JavaScript outside the browser

- JavaScript is interpreted
- node.js - CLI for Google Chrome's V8 JavaScript interpreter

```
$ node
> console.log('Hello world!');
Hello world!
undefined
>
```

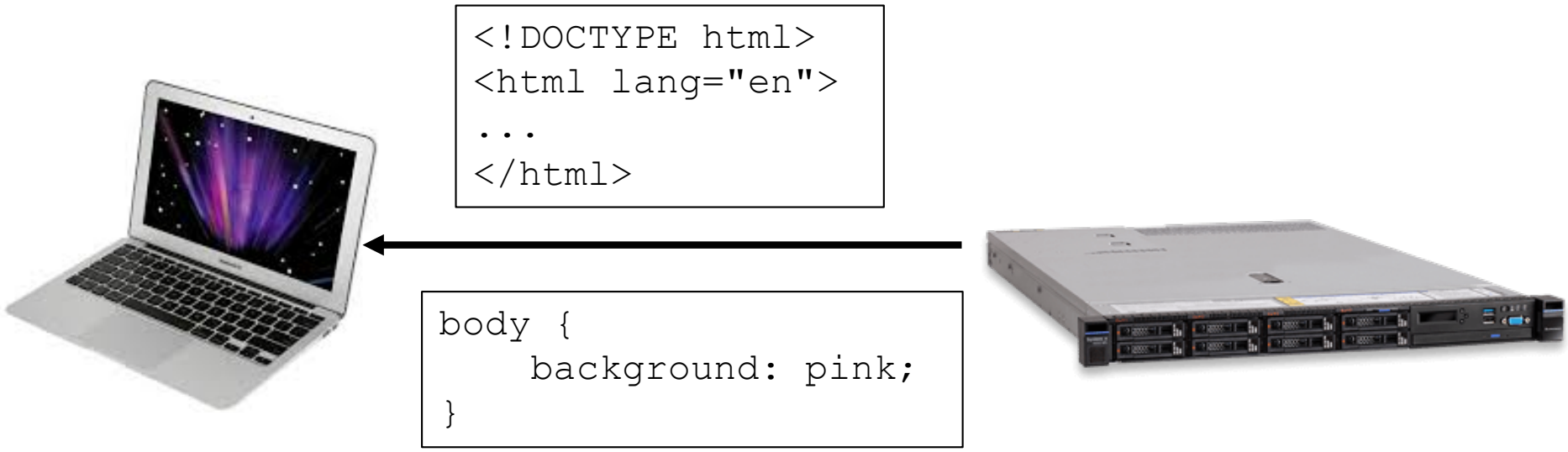
- Note: the output of `console.log()` is undefined because all functions return something

Agenda

- Introduction to JavaScript
- **Review**
- Client-side dynamic pages
- JavaScript execution model: event-driven programming
- JavaScript data model
- Pitfalls (best and worst practices)

Review: static pages

- A *static page* is only HTML/CSS
  - No programming language on the server
  - Same content every time the page is loaded



Review: static pages

- On the server side: HTTP servers are fileservers
- On the client side: browsers are HTML renderers

- Example
  - `python3 -m http.server`
  - Copies files

Review: server-side dynamic pages

- Server-side dynamic pages: Response is the output of a function.

1. Client makes a request
2. Server executes a function
   - Output is usually HTML
3. Server response is the output of the function

Review: project 2 server-side dynamic pages

Client specifies a URL

- This *looks* like a file path on the server
- But server *really* runs a function, serves returned output

- How does function generate content?
  - State is stored in a database (SQLite)
  - Function issues SQL queries to get relevant state
  - Populates Python object
  - Renders template using object
  - Returns resulting HTML

- Generation of content specific to each request

Limitations of server-side dynamic pages

- Server-side dynamic pages
  - Are created at time of request
  - Don't change after the HTML has been generated and transferred to client's browser
- What would not work if all we had was server-side dynamic pages?

Limitations of server-side dynamic pages

- What would not work if all we had was server-side dynamic pages?
- Examples from the web
  - No in-browser chat
  - No browser-based field validation
  - No grabbable maps
  - No deferred data loading in Gmail
- Examples from Project 2:
  - Add comment without a page reload
  - Add like without a page reload
  - Delete without a page reload
  - Infinite scroll
  - Double click to like

Agenda

- Introduction to JavaScript
- Review
- **Client-side dynamic pages**
- JavaScript execution model: event-driven programming
- JavaScript data model
- Pitfalls (best and worst practices)

Client-side dynamic pages

- Client-side dynamic pages: JavaScript running in the client's web browser modifies the DOM. The rendered page changes.

1. Client executes JavaScript
2. JavaScript code modifies the DOM
3. Rendered page changes

Client-side dynamic pages more detail

1. Client requests static HTML page from server
   - Static HTML has a `<script src="script.js">` tag
2. Client requests static JavaScript file from server
3. Client executes JavaScript
4. JavaScript code modifies the DOM
   - Finds a node in the DOM
   - Changes the node's text, add or remove a node, etc.
5. Rendered page changes

Example HTML

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<body>
  <!-- the code for hello() is in script.js -->
  <button onclick="hello()">
    Click me!
  </button>
  <div id="JSEntry"></div>

  <!-- script tags go in body, not in head -->
  <script src="script.js"></script>
</body>
</html>
```

Example JavaScript

```
//script.js
function hello() {
  console.log("Hello World!");
  n = document.getElementById("JSEntry");
  n.innerHTML = "Hello world!";
}
```

Example in browser

- Before button click

- After button click

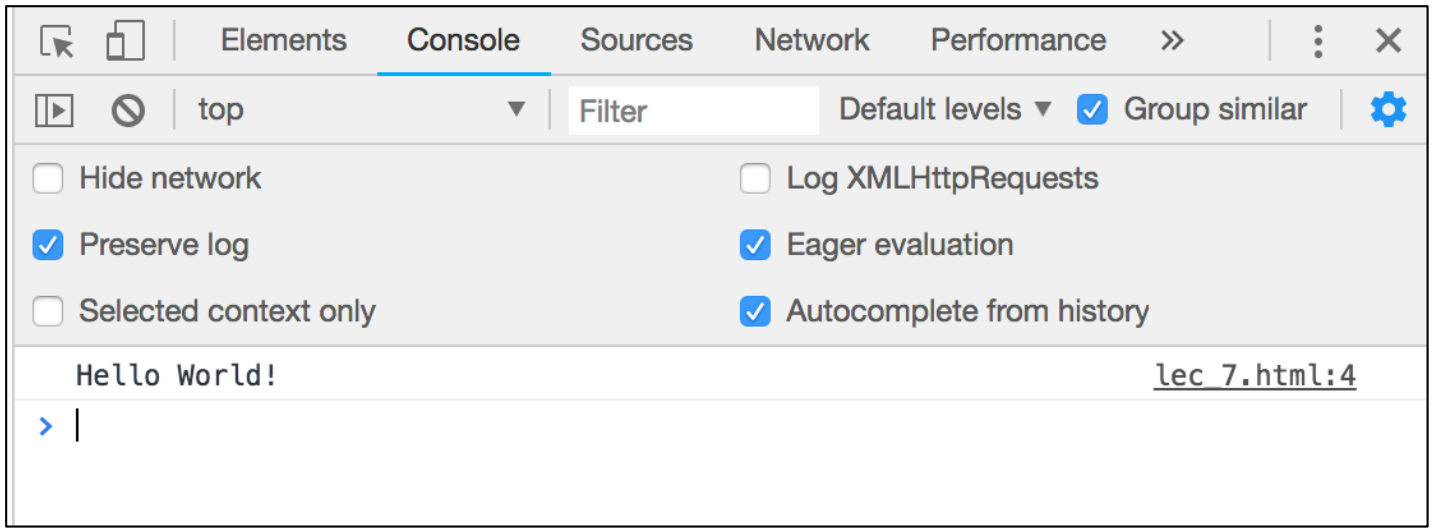
DOM

```
<html>
<body>
  <button onclick="hello()">
    Click me!
  </button>
  <div id="JSEntry"></div>
  <script src="script.js"></script>
</body>
</html>
```

`console` API

```
function hello() {
  console.log("Hello World!");
  n = document.getElementById("JSEntry");
  n.innerHTML = "Hello world!";
}
```

- Code running in the browser's interpreter has access to APIs
- Accessible via global objects
- `console` API writes text to the developer console



document API

```
function hello() {
  console.log("Hello World!");
  n = document.getElementById("JSEntry");
  n.innerHTML = "Hello world!";
}
```

- The `document` API represents the web page loaded in the browser
  - Web page represented as a Document Object Model (DOM)

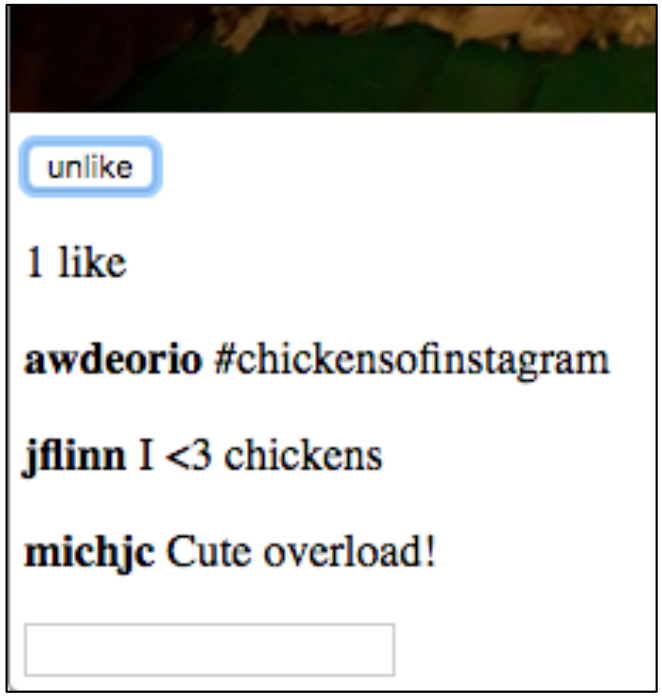
Agenda

- Introduction to JavaScript
- Review
- Client-side dynamic pages
- **JavaScript execution model: event-driven programming**
- JavaScript data model
- Pitfalls (best and worst practices)

User interaction

- How does a button click lead to JS code execution?

- Attach a function to a button
- Button causes an *event*
- *Event* runs a function
- Function modifies the DOM



Events

```
<html><body>
  <button onClick="hello()" type="button">
    Click Me!
  </button>
  <div id="JSEntry"></div>
  <script>
  function hello() {
    n = document.getElementById("JSEntry");
    n.innerHTML = "Hello World!";
  }
  </script>
</body></html>
```

Before click

After click

Event-driven programming

- In event-driven programming, the flow of the program is determined by *events*
- A few examples of events built into the browser:
  - `onclick`: user clicks a button
  - `onmouseover`: The user moves the mouse over an HTML element
  - `onkeydown`: The user pushes a keyboard key
  - `onload`: The browser has finished loading the page
- Event-driven programming is useful for GUIs like web applications

Callback functions

- A main loop listens for events and triggers a *callback function*
- A callback function is just a normal function, waiting to be executed
  - Current example: `hello()` is a callback

```
function hello() {
  n = document.getElementById("JSEntry");
  n.innerHTML = "Hello World!";
}
```

Event handlers

- In the HTML, we registered our function as an *event handler*
- That means telling the browser "please run this function when X event occurs"

```
<button onClick="hello()" type="button">
  Click Me!
</button>
```

- The JavaScript interpreter maintains a table of events that map to functions

Event	Function
onClick	hello

Execution model

- In C/C++, Python, etc., function calls live on the stack, and dynamic objects live on the heap
- The function on the top of the stack executes

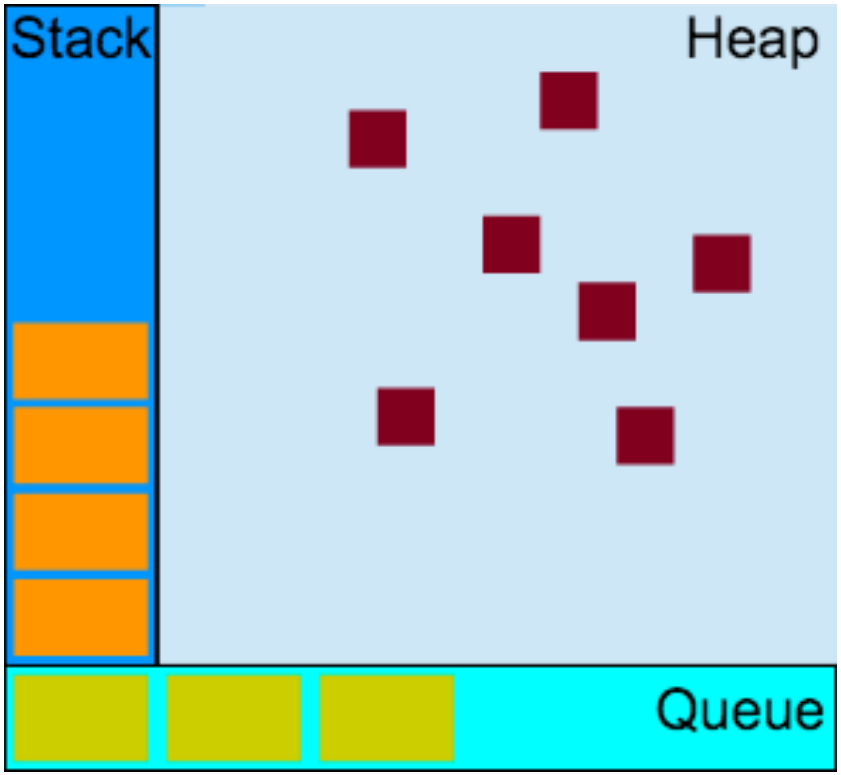
The event queue

- In JavaScript, function calls live on the stack, objects live on the heap, and *messages live on the queue*
- The function on the top of the stack executes.
- *When the stack is empty, a message is taken out of the queue and processed.*
- Each message is a function

- An event adds a message to the queue

The event queue

- Conceptual model



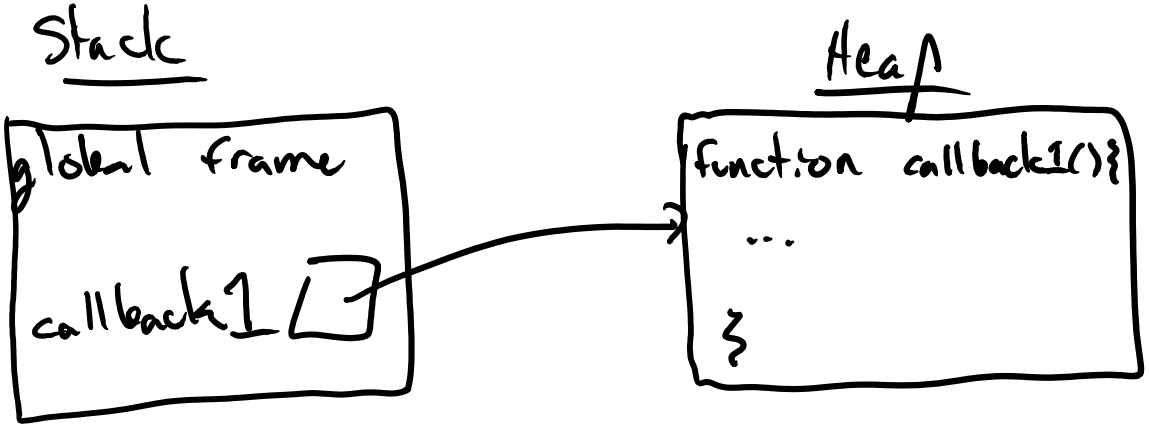
Adding events to the queue

- Example: You can schedule an event on the queue for a later time
- This function will run approximately 1s in the future
- `callback1` is added to the *event table*, which maps events to callbacks

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```

Adding events to the queue

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```

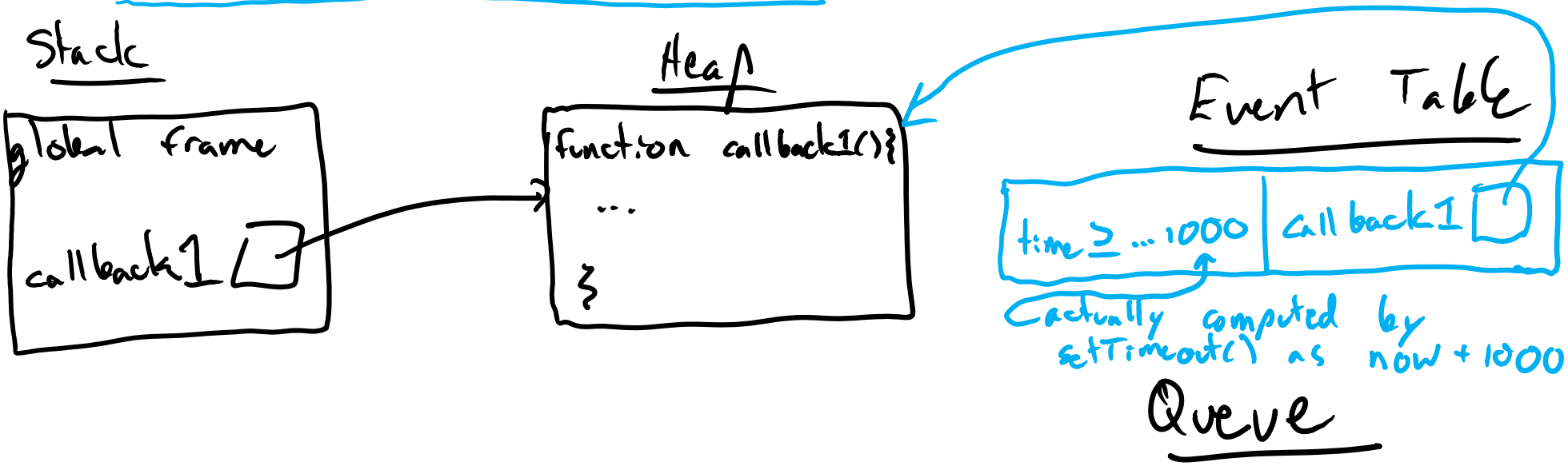


Event Table

Queue

Adding events to the queue

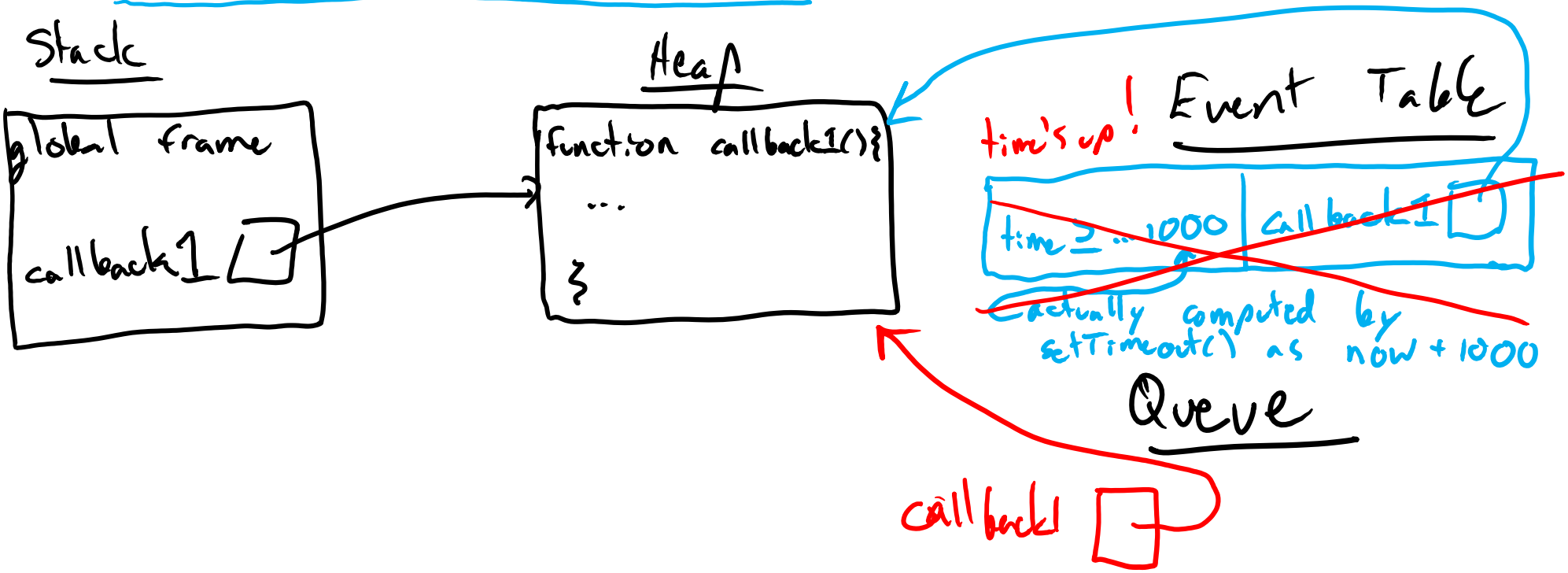
```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```



Adding events to the queue

1000 ms later...

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```

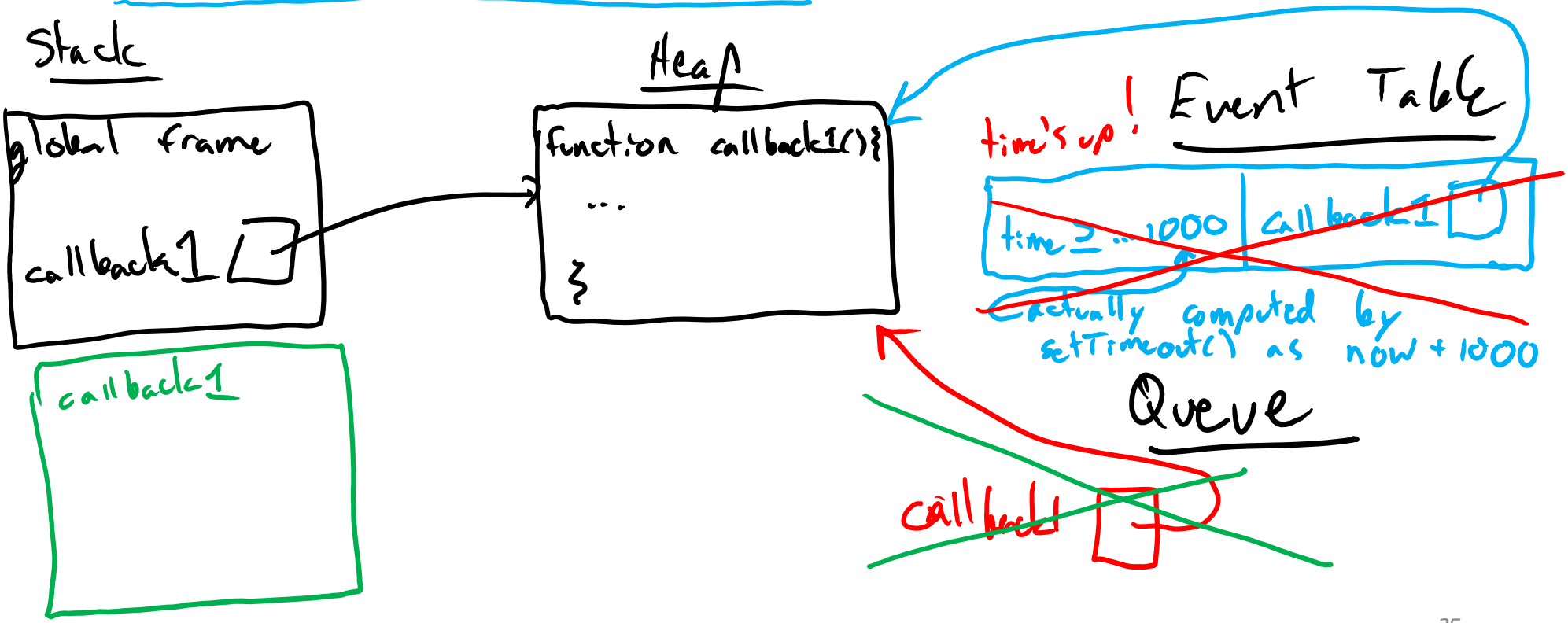


output: 'this is a msg from callback1'

Adding events to the queue

1000 ms later...

```
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000);
```



Exercise

- What is the output of this code?

```
function f() {
  console.log(‘beginning');
  function callback1() {
    console.log('callback1');
  }
  setTimeout(callback1, 1000); //1s
  console.log('middle');
  function callback2() {
    console.log('callback2');
  }
  setTimeout(callback2, 2000); //2s
  console.log('end');
}
f();
```

Solution and diagram

```
function f() {
  console.log(beginning');
  function callback1() {
    console.log('callback1');
  }
  setTimeout(callback1, 1000);
  console.log('middle');
  function callback2() {
    console.log('callback2');
  }
  setTimeout(callback2, 2000);
  console.log('end');
}
f();
```

```
beginning
middle
end
callback1
callback2
```

Event Table

Agenda

- Introduction to JavaScript
- Review
- Client-side dynamic pages
- JavaScript execution model: event-driven programming
- **JavaScript data model**
- Pitfalls (best and worst practices)

`null` vs. `undefined`

- `null`: a value that indicates a deliberate non-value
- `undefined`: a value of type undefined that indicates an uninitialized value

```
> let x;
undefined
> x === undefined;
true
> x = null;
null
> x === undefined;
false
> x === null;
true
```

Primitives

- *Primitives and objects* are JavaScript's abstraction for data
- *Primitives* are not objects and have no methods
  - `string, number, boolean, null, undefined, symbol`
  - Note the lowercase
  - Low-level representation in interpreter
  - Immutable literals
- Examples:
  - `let s = 'hello';`
  - `let pi = 3.141;`
  - `let b = true;`

Objects

- *Objects* have *properties* and a *prototype*
- *Properties* are values associated with an object
  - Named
  - Unordered
- Example

```
> let course = { name: 'Web Systems', num: 485 };
undefined
> course.name
'Web Systems'
 > course.num
485
```

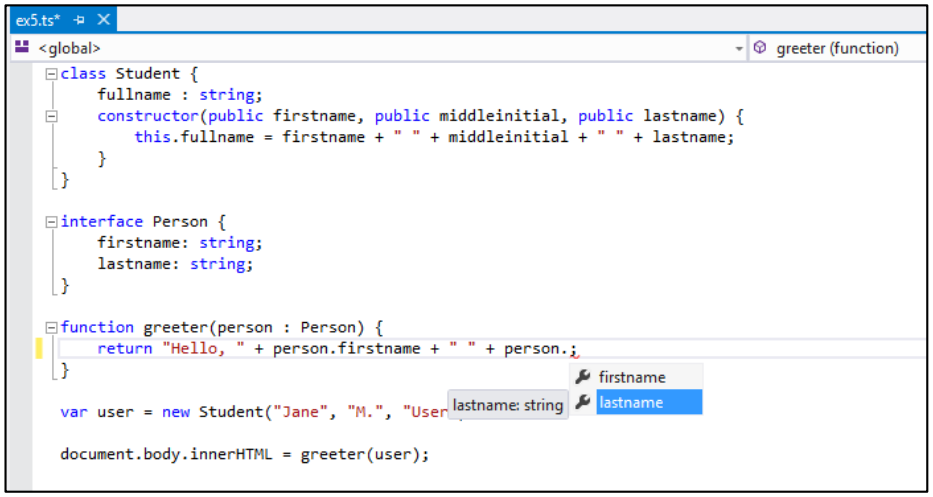
Data model: built in objects

- Primitive values have object equivalents that wrap around the primitive values
  - `String`, `Number`, `Boolean`, `Symbol`
  - Except for `null` and `undefined`
  - Provides useful member functions

- Additional objects built in
  - `Array, Map, Set`
  - Many others: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

Type system

- JavaScript is dynamically typed
  - No static type checking
- Some language extensions add static type checking with a compiler:
  - Flow (Facebook)
  - TypeScript (Microsoft)




```
1  // @flow
2  function square(n: number): number {
3    return n * n;
4  }
5
6  square("2"); // Error!
```

Prototypes

- *Objects* have *properties* and a *prototype*
- *Properties* are values associated with an object
- *Prototypes* are the mechanism by which JavaScript objects inherit features from one another
- In JavaScript, there is no distinction between instances and classes/types
  - Everything is an object
- For Java/C/Python programmers, prototypes feel very strange

Prototypes

- Every JS object has a *prototype attribute*
  - Akin to the object's "parent"
  - All objects inherit the properties and methods from their prototype
    - When resolving a reference, JS climbs prototype tree until name is found (or not)
  - The prototype is *another object*, not a superclass
  - Examine it via the `__proto__` attribute
  - **CAREFUL (also: WEIRD)**: `__proto__` and `prototype` are not the same thing!

Agenda

- Introduction to JavaScript
- Review
- Client-side dynamic pages
- JavaScript execution model: event-driven programming
- JavaScript data model
- **Pitfalls (best and worst practices)**

Common mistake: equality operators

- "JavaScript has two sets of equality operators: `===` and !==, and their evil twins == and !=." -- Douglas Crockford
- `==` performs a type conversion when comparing two things
- `===` no type conversion
  - Return false if the types differ

- A few interesting cases:

```
'' == '0' // false
0 == '' // true
0 == '0' // true
```

ALWAYS use === and !==

Common mistake: scope

- "Simply" assigning values always creates a global variable

```
> function f() {
    x = 5;
  }
> f();
> x
5
```

NEVER simply assign values

Common mistake: scope

- `var` creates a local or global scoped variable
- Functions create scope

```
> var x = 0;
> function f() {
    var x = 5;
  }
> f();
> x
0
```

- Other blocks do not

```
> var x = 0;
> if (x === 0) {
    var x = 5;
  }
> x
5
```

Common mistake: scope

- `let` and `const` create **block**-scoped (not global- or function-scoped) variables
- A lot like C/C++

```
> let x = 0;
> if (x === 0) {
    let x = 5;
  }
> x
0
```

Common mistake: hoisting

- Variables declared with `var` are *hoisted* to the top of the function

```
> function f() {
    console.log(x === undefined);
    var x = 5;
  }
> f();
true
```

- Variables declared with `let` or `const` are not

```
> function f() {
    console.log(x === undefined);
    let x = 5;
  }
> f();
ReferenceError: x is not defined
```

Common mistake: hoisting

- Variables declared with `var` are *hoisted* to the top of the function

```
> function f() {
    console.log(x === u
    var x = 5;
  }
> f();
true
```

NEVER use `var`

- Variables declared with `let` or `const` are not

```
> function f() {
    console.log(x === u
    let x = 5;
  }
> f();
ReferenceError: x is not defined
```

ALWAYS use `let` or `const`

Common misunderstanding: `const`

- `const` means you can't reassign the reference

```
> const eecs485 = { name: 'Web Systems', num: 485 };
> eecs485 = { name: 'Chicken Stories', num: 101 };
TypeError: Assignment to constant variable.
```

- Changing the object is OK

```
> const eecs485 = { name: 'Web Systems', num: 485 };
> eecs485.name = 'Chicken Stories';
'Chicken Stories'
> eecs485.num = 101;
101
> eecs485
{ name: 'Chicken Stories', num: 101 }
```

- `const x` in JavaScript is like `int *const p` in C.

Common mistake: for-in loops

- `for-in` loops often yield unexpected results
  - They iterate "up the prototype chain"

```
> const chickens = ['Magda', 'Marilyn', 'Myrtle II'];
> for (let chicken in chickens) {
>   console.log(chicken);
> }
1
2
3
```

- ES6's for-of loops are nice, but are hard to analyze statically, so some style guides do not allow them

```
> for (let chicken of chickens) {
>   console.log(chicken);
> }
Magda
Marilyn
Myrtle II
```

Common mistake: for-in loops

- `for-in` loops often yield unexpected results
  - They iterate "up the prototype chain"

```
> const chickens = ['Magda', 'Marilyn', 'Myrtle II'];
> for (let chicken in chickens) {
>   console.log(chicken);
> }
1
2
3
```

NEVER use `for-in`

- ES6's for-of loops are nice, but are hard to analyze statically, so some style guides do not allow them

```
> for (let chicken of chickens) {
>   console.log(chicken);
> }
Magda
Marilyn
Myrtle II
```

SOMETIMES use `for-of`

(if style guide allows)

Iteration with `forEach` and `map`

- `forEach` loops "do the right thing"
  - Behave like other programming languages (C, C++, Perl, Python ...)
  - We'll learn about the => syntax soon (it's an anonymous function)

```
const chickens = ['Magda', 'Marilyn', 'Myrtle II'];
chickens.forEach((chicken) => {
  console.log(chicken);
});
```

- `map` is another nice option
  - Use it to transform an array into another array

```
const chickens_say = chickens.map(chicken => (
  `${chicken} says cluck`
));
console.log(chickens_say);
//[ 'Magda says cluck', 'Marilyn says cluck',
//  'Myrtle II says cluck' ]
```

Iterating over an object's keys and values

- Iterator over objects using a `forEach` loop

```
> const chickenAges = {
  magda: 1,
  marilyn: 2,
  myrtleii: 1.5,
};
> Object.entries(chickenAges).forEach(([key, value]) => {
  console.log(key, value);
});
magda 1
marilyn 2
myrtleii 1.5
```

References

- Resource for those who already know how to program (you!)
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

- Full reference with all the details
  https://developer.mozilla.org/en-US/docs/Web/JavaScript