Review: Closures

```
var counter = function() {
  var count = 0;
  return {
    increment: function() {
      count += 1;
    },
    getValue: function() {
      return count;
    }
  }
}
```

- Can make `count` a private member with a closure
- Encapsulation using closures

Review: Closures

```
var i = counter();
console.log(i.getValue()); //0
i.increment();
console.log(i.getValue()); //1
i.increment();
console.log(i.getValue()); //2
var j = counter();
console.log(j.getValue()); //0
j.increment();
console.log(j.getValue()); //1
console.log(i.getValue()); //2
i.increment();
console.log(i.getValue()); //3
```

- It works!

Review: Closures

```
var counter = function() { /*...*/ };
var i = counter();
console.log(i.getValue()); //0
i.increment();
```

- Notice that the inner function has a longer lifetime than its outer function

Review: Closures

- The `counter` function is designed to be used without the `new` prefix, so the name is not capitalized
- When we call counter, it returns a new object containing `getValue()` and `increment()` methods

```
var counter = function() {
  var count = 0;
  return {
    increment: function() { /*...*/ },
    getValue: function() { /*...*/ }
  }
}
var i = counter();
```

Review: Closures

- `getValue()` and `increment()` have privileged access to `counter`'s `count` property, *even though counter has already returned*!

```
var counter = function() {
  var count = 0;
  return {
    increment: function() { /*...*/ },
    getValue: function() { /*...*/ }
  }
}
var i = counter();
console.log(i.getValue()); //0
```

Review: Closures

- The `count` property is not a copy, it is the original parameter
- This is possible because the function has access to the context in which it was created
  - *This is a closure*

```
var i = counter();
console.log(i.getValue()); //0
i.increment();
console.log(i.getValue()); //1
i.increment();
console.log(i.getValue()); //2
```

Review: Pitfalls around new

```
function Professor(first, last) {
    this.first = first;
    this.last = last;
};
var deorio = Professor("Drew", "DeOrio");
console.log(deorio.first); //Error!
console.log(first);//Drew Why??
```

- Without `new`, `deorio` is undefined
- `first` and `last` are global variables!!!
- `"use strict";` can help avoid this error

Prototypes

- JavaScript has no classes, so how can we implement inheritance?
- Use *prototypes*
  - Inspired by Self, Smalltalk
  - Class-free inheritance
- 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

- Let's start with an example
- We'll use a constructor function
  - Capitalize, since it's meant to be used with `new`

```
function Professor(first, last) {
    this.first = first;
    this.last = last;
};

var deorio = new Professor("Drew", "DeOrio");
```

Prototypes

- Let's start with an example
- We'll use a constructor function

```
function Professor(first, last) {
    this.first = first;
    this.last = last;
};
var deorio = new Professor("Drew", "DeOrio");
console.log(deorio);

  //Professor {first: "Drew", last: "DeOrio"}
var mjc = new Professor(”Mike", ”Cafarella");
console.log(mjc);

  //Professor {first: ”Mike", last: ”Cafarella"}
```

Prototypes

- Add a property to an existing object

```
var deorio = new Professor("Drew", "DeOrio");
var mjc = new Professor(”Mike", ”Cafarella");
deorio.num_chickens = 4;
console.log(deorio.num_chickens); //4
console.log(mjc.num_chickens); //undefined
```

Prototypes

- Add a method to an existing object

```
deorio.name = function() {
    return this.first + " " + this.last;
}

console.log(deorio.name()); //Drew DeOrio
console.log(mjc.name()); //Uncaught TypeError:
mjc.name is not a function
```

- This is OK, but it's piecemeal

Adding a method: Option 1

Modify the constructor function

```
function Professor(first, last) {
    this.first = first;
    this.last = last;
    this.name = function() {
      return this.first + " " + this.last;
    };
};
var deorio = new Professor("Drew", "DeOrio");
var mjc = new Professor(”Mike", ”Cafarella");
console.log(deorio.name()); //Drew DeOrio
console.log(mjc.name()); //Mike Cafarella
```

Adding a method: Option 2

- 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
  - Note that `__proto__` and `prototype` are not the same thing!

- When `new` creates an object, the obj's `__proto__` attr is set to `Constructor.prototype`

Adding a method: Option 2

- We can access a constructor's `prototype` --- which is assigned to any objects it constructs --- directly
-
```
function Professor(first, last) {
    this.first = first;
    this.last = last;
};
var deorio = new Professor("Drew", "DeOrio");
var mjc = new Professor(”Mike", ”Cafarella");
```

```
Professor.prototype.name = function() {
  return this.first + " " + this.last;
}
console.log(deorio.name()); //Drew DeOrio
console.log(mjc.name()); //Mike Cafarella
```

- *Existing objects now have this method!*

Prototype Details

- Some more details on prototypes:
  - Used only during retrieval from object
  - Nothing to do with updates
  - If field isn't in object, check its prototype!
  - Keep checking up the prototype tree
  - Works with any field, including fns
  - Be careful of `a.__proto__` vs `a.prototype`
- All object literals have `__proto__` of `Object.prototype`
- Functions have `prototype` AND `__proto__`
- Functions' `__proto__` is `Function.prototype`

Differential inheritance

- Differential inheritance is the name for JavaScript's "normal" prototype style inheritance
- Differential inheritance means not needing constructors or `new`
- Differential inheritance
  - Creates an object from a target object
  - Creates & initializes the members
  - Then you modify only the stuff that should be *different*

Differential inheritance

- Create an object from a **target object**

```
var professor = {
    first: "",
    last: "",
    name: function() {
       return this.first + " " + this.last;
    },
    says: function() {
       return "Prof. " +this.last+ " says computers";
    }
};
console.log(professor)

//{}
```

Differential inheritance

- **Create an object** from a target object
- Modify only the properties that are *different*

```
var professor = { /*...*/ };
var deorio = Object.create(professor);
deorio.first = "Drew";
deorio.last = "DeOrio";
console.log(deorio.says());

//Prof. DeOrio says computers
```

Differential inheritance

- **Create an object** from a target object
- Modify only the properties that are *different*

```
var professor = { /*...*/ };
var mjc = Object.create(professor);
mjc.first = ”Mike";
mjc.last = ”Cafarella";
console.log(mjc.says());

//Prof. Cafarella says computers
```

Differential inheritance

- We can modify any property, including functions

```
deorio.says = function() {
  return "Prof. " + this.last + " says chickens!"
}
console.log(deorio.says());

//Prof. DeOrio says chickens!
```

Differential inheritance

- This gives us behavior that is similar to hierarchies of polymorphic types in Python/C++/etc.

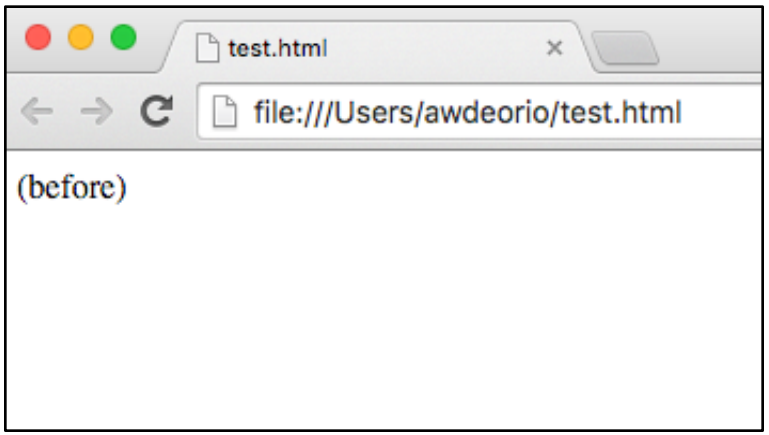
```
profs = [deorio, mjc]
for (var i=0; i<profs.length; i+=1) {
    console.log(profs[i].says());
}

//Prof. DeOrio says chickens!
//Prof. Cafarella says computers
```

The DOM

The DOM

- JavaScript interacts with the browser via the Document Object Model (DOM)
- The DOM is a tree of objects that reflect the current page's HTML
  - JavaScript can read it to see what's there
  - JavaScript can write to it to change the screen

Accessing the DOM




```
//test.html
<html>
<head><script src="test.js"></script>
</head><body><p id="hello">(before)</p></body>
</html>
```

```
//test.js
document.getElementById("hello").innerHTML =
"Hello World!";
```

- This changes the HTML inside the paragraph to say "Hello World!"

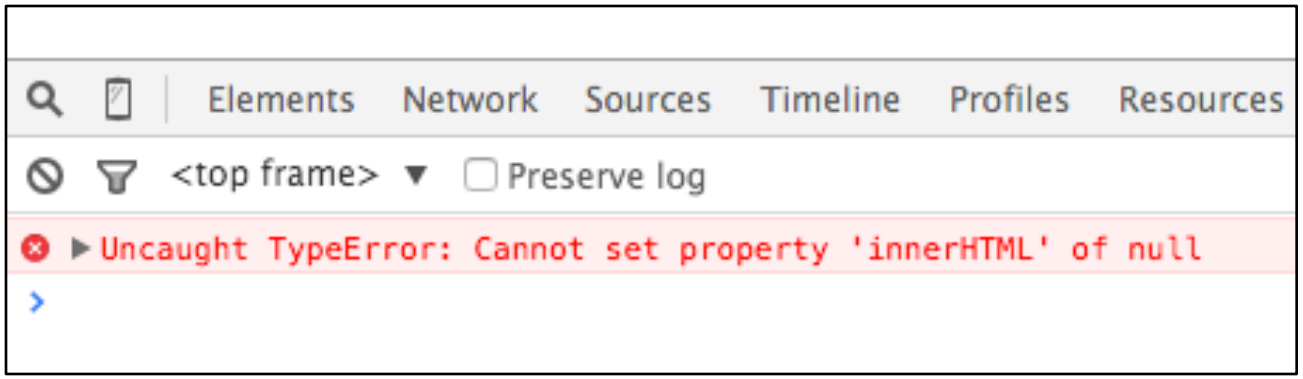
DOM and Recursion

- The DOM is a recursively-defined tree
- Easy to navigate using recursive JavaScript

```
var walkDOM = function walk(node, func) {
  func(node);
  node = node.firstChild;
  while (node) {
    walk(node, func);
    node = node.nextSibling;
  }
};

// Apply walkDOM to document.body
```

Pitfall: page isn't loaded

- Our hello world example won't work if the JavaScript program runs *before* the HTML has loaded
- Use an event and a callback function to fix this



Event-driven programming

- In event-driven programming, the flow of the program is determined by events
  - Example: page load
  - Example: user clicks a button
- Event-driven programming is useful for GUIs like web applications

Event-driven programming

- A main loop listens for events and triggers a callback function
- A callback function is just a normal function, waiting to be executed

Back to our example

- Let's put our previous code in a function, and save it to a variable

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

- Then, we'll register our function as an *event handler*
- That means telling the browser "please run this function when X event occurs"

```
window.onload = hello_func;
```

Back to our example

- Simplify the code:

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

The event queue

- In C/C++, Java, 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

The event queue

- A conceptual model

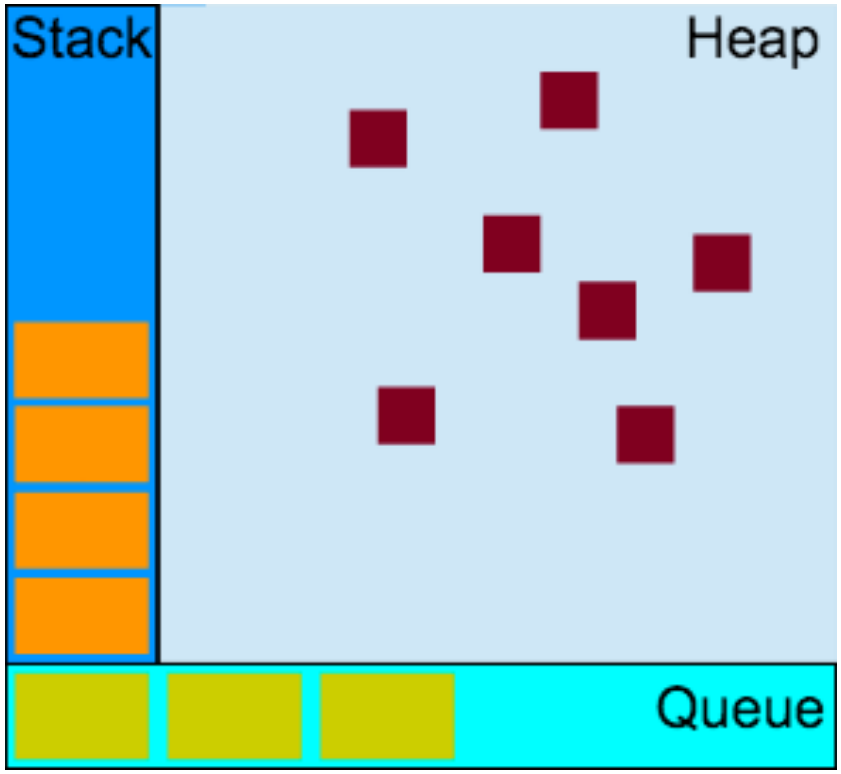


Image: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

Adding events to the queue

- Example: You can schedule an event on the queue for a later time
- This function will run approx. 1s in the future

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

Exercise

- What is the output of this code?

```
console.log('this is the start');
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000); //1s
console.log('this is just a message');
function callback2() {
    console.log('this is a msg from callback2');
}
setTimeout(callback2, 2000);//2s
console.log('this is the  end');
```

Solution

- What is the output of this code?

```
console.log('this is the start');
function callback1() {
    console.log('this is a msg from callback1');
}
setTimeout(callback1, 1000); //1s
console.log('this is just a message');
function callback2() {
    console.log('this is a msg from callback2');
}
setTimeout(callback2, 2000);//2s
console.log('this is the  end');
```

this is the start

this is just a message

this is the end

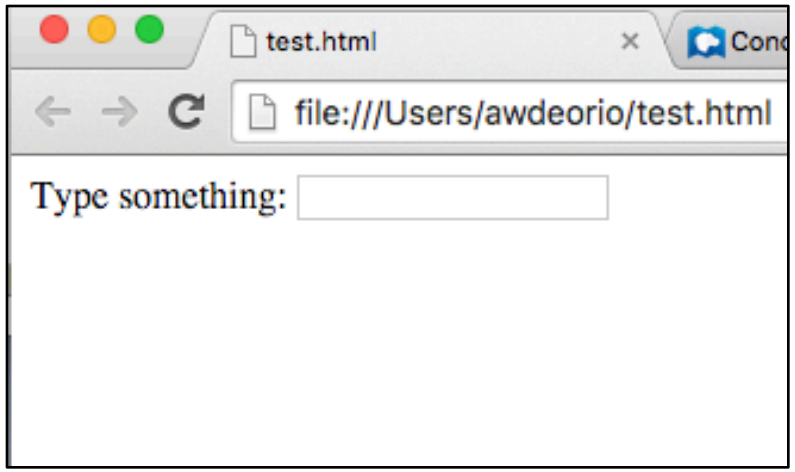
this is a msg from callback1

this is a msg from callback2

Events and HTML

- You can couple JavaScript functions with events in HTML
- test.html

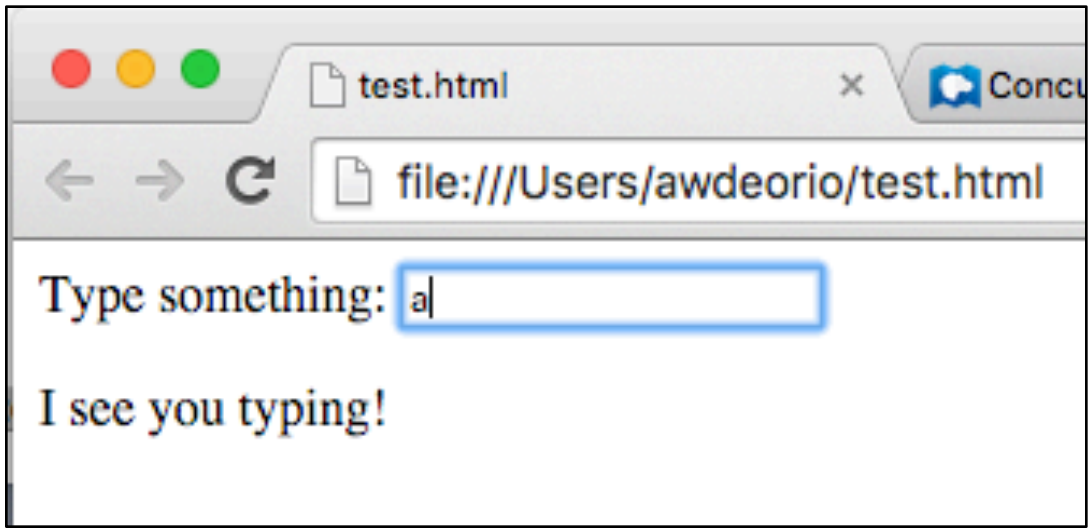
```
<html>
<head><script src="test.js"></script></head>
<body>Type something
<input type="text" onkeyup="hello()">
<p id="hello"></p>
</body>
</html>
```



Events and HTML

- You can couple JavaScript functions with events in HTML
- test.js

```
function hello() {
    document.getElementById("hello").innerHTML =
"I see you typing!";
}
```



Many Other Events

- `onmouseover`
  - Mouse is moved over HTML element
- `onmouseout`
- `onclick`
- `onchange`
  - HTML element has been changed
- `onload`
  - Page has finished loading (in browser)
- …

A few more odds and ends

- Node.js
- Development tools
- jQuery

Node.js

- Node.js is a JavaScript framework for writing **servers**
  - Especially web apps with huge numbers of live but mainly-idle connections (e.g., powering JavaScript chat clients)
- Server-side interpreter (V8, from Chrome), plus libraries
- Uses *events*, not threads
  - If an OS thread has 2MB of overhead, then each chat client --- even idle ones! --- are expensive

Everyday Node

- Nice primitive: callback function that…
  - Takes request as parameter #1
  - Generates a response, by writing to parameter #2
- All polling & event-processing handled by framework

```
var express = require("express");
var app = express();
var port = 3700;

app.get("/", function(req, res){
  res.send("It works!");
});

app.listen(port);
console.log("Listening on port " + port);
```

JavaScript tools

- A few tools make JavaScript go down more easily
  - Development tools
  - JavaScript libraries
- JSLint
  - Tries to warn you about all the bad things in your code
- Chrome Debugger
  - Already covered this last time

Google Web Toolkit

- For programmer, looks like Java

```
public class SW implements EntryPoint {
  private VerticalPanel mp= new VerticalPanel();
  private FlexTable sft = new FlexTable();
  //...
  private ArrayList<String> stocks =
    new ArrayList<String>();
}
```

- Gets compiled into browser-independent JavaScript!
- Looks like Java, but needs:
  - Its own compiler
  - Its own implementation of all class libraries!

TypeScript

- Again, compiled into browser-independent JavaScript

ex5.ts*

<global> greeter (function)

```
class Student {
    fullname : string;
    constructor(public firstname, public middleinitial, public lastname) {
        this.fullname = firstname + " " + middleinitial + " " + lastname;
    }
}

interface Person {
    firstname: string;
    lastname: string;
}

function greeter(person : Person) {
    return "Hello, " + person.firstname + " " + person.;
}

var user = new Student("Jane", "M.", "User

document.body.innerHTML = greeter(user);
```

lastname: string

firstname

lastname

jQuery

- Very popular javascript library.
- Makes many things much easier to do.
- Like STL with C++

Summary

- JavaScript is an interpreted language built into the browser
- Lots of unusual language features AND sharp edges
- It's great for writing client-side applications that run in the browser