Server-side vs. Client-side

- One way to make dynamic web pages is server-side
  - Example: project 1 and project 2 run Python/Flask on the server
  - Reloads entire web page with different content created by server
- A second way to make dynamic web pages is client-side
  - Client modifies existing web page without reloading
  - Need a programming language
  - HTML is a markup language, not enough
  - Enter JavaScript

JavaScript History

- Created in 1996
- By Brendan Eich at Netscape
- In 10 days
- Crazy mix of ideas from C, Self, and Scheme
- Not really associated with Java at all

Overview

- JavaScript (aka JScript, ECMAScript) is:
    - Everywhere
    - In the browser, with access to the DOM
    - Interpreted
    - Object-oriented, with prototypes
    - Loosely typed (AKA untyped)
    - Lexically scoped
    - Lambda-oriented (first-class functions)
    - Named like Java, with the syntax of C, and the design of Self
    - Easy to use, for both good and evil

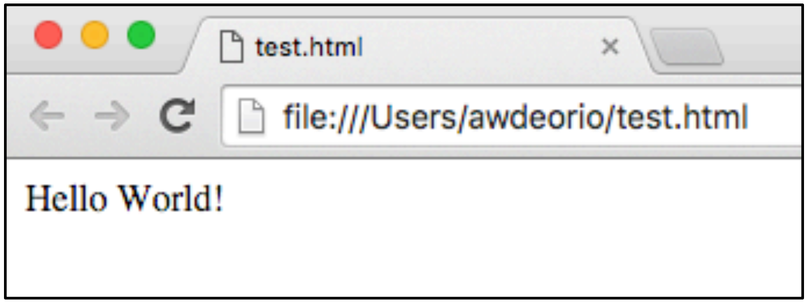
Hello, World!

- hello.html:

```
<html>
<head><script src="test.js"></script></head>
<body></body>
</html>
```

- test.js

```
document.write("Hello World!");
```



Hello World!

Output

- 4 ways to output
- Directly to HTML
  - `document.write("hello")`
- Pop up
  - `window.alert("hello");`
- Browser's debug console
  - `console.log("hello")`
- Modify HTML
  - `<p id="myp"></p>`
  - `<script>document.getElementById("myp").innerHTML = "hello";</script>`

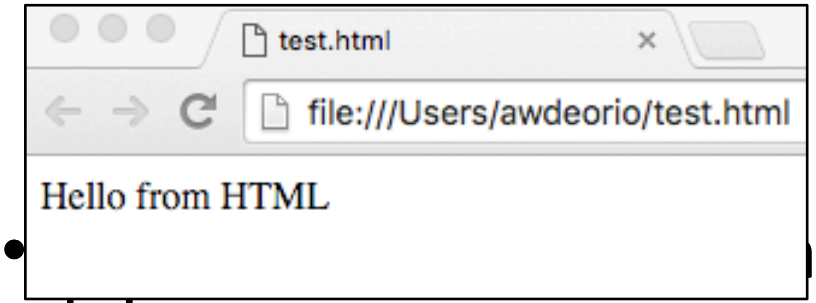
Modifying HTML

- hello.html:

```
<html>
<head><script src="test.js"></script></head>
<body><p id="hellop">Hello from HTML</p><body>
</html>
```

- test.js

```
document.getElementById("hello").innerHTML =
  "hello from JavaScript";
```

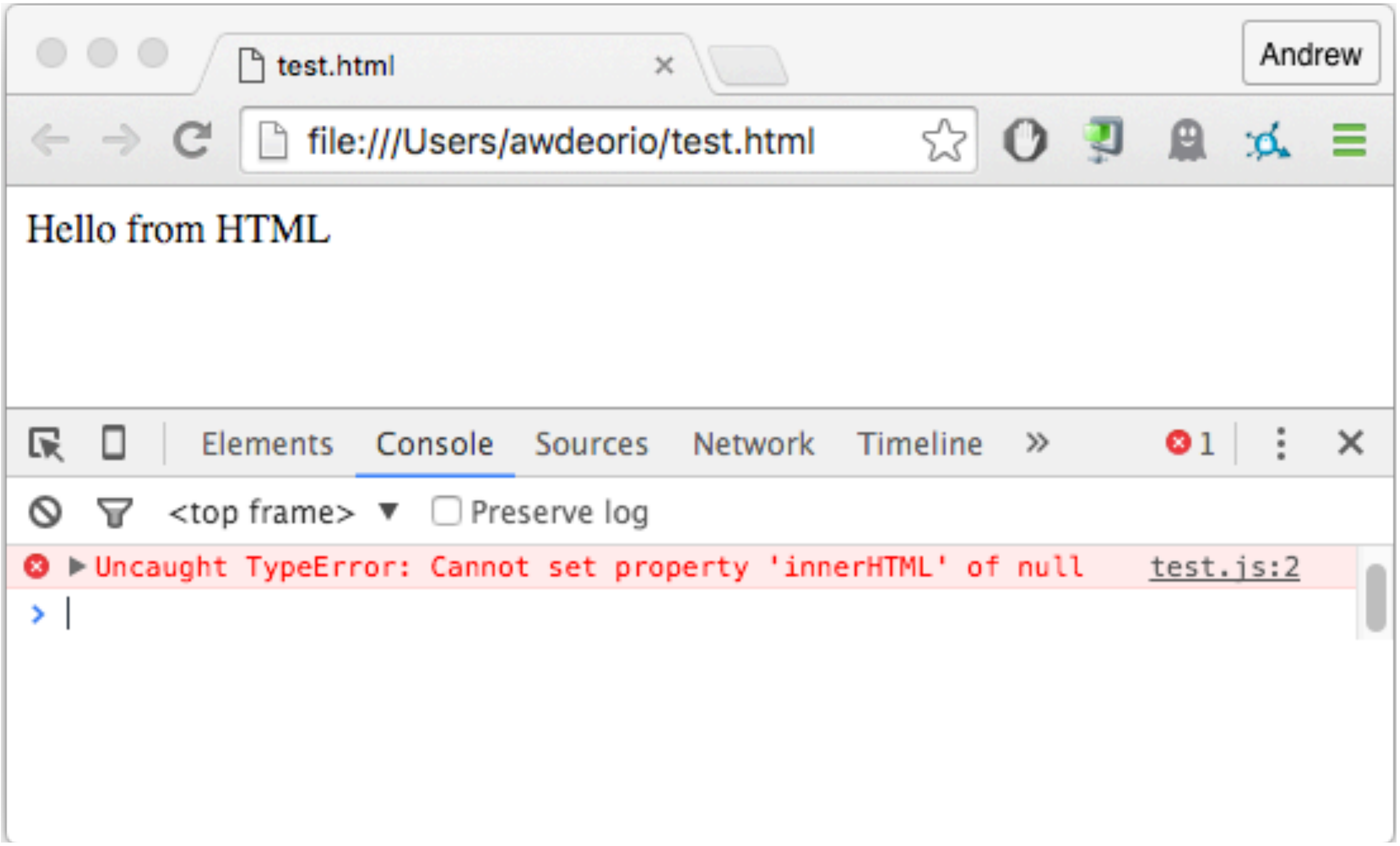


Hello from HTML

- ke, let's use the browser to debug

Chome Debugger

- Fire up the Chrome debugger ("Developer Tools")



Hello from HTML

```
Uncaught TypeError: Cannot set property 'innerHTML' of null    test.js:2
>
```

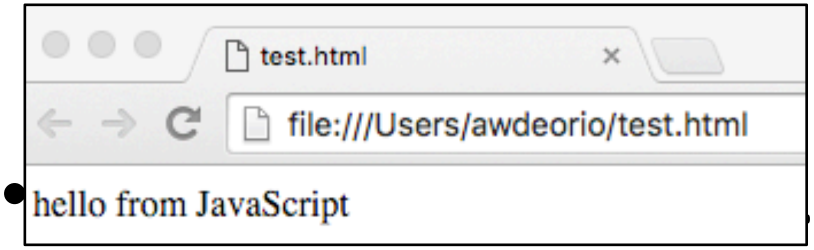
Modifying HTML

- hello.html:

```
<html>
<head><script src="test.js"></script></head>
<body><p id="hellop">Hello from
HTML</p><body>
</html>
```

- test.js

```
document.getElementById("hello").innerHTML =
  "hello from JavaScript";
```



hello from JavaScript

- . Fixed using debugger.

Caveat: to reproduce these results, you'll actually need the code below. We'll cover this in the next lecture.

```
time.window.onload = function () { document.getElementById("hellop").innerHTML = "hello from JavaScript"; }
```

The Basics

- Next, we'll discuss (very) quickly JavaScript basics
- For more details, check out JavaScript: The Good Parts by Douglas Crockford

Unearthing the excellence in JavaScript
JavaScript: The Good Parts
O'REILLY
YAHOO! PRESS
Douglas Crockford

Command line JavaScript

- Node.js helpful for trying out the following slides at the command line.
- JavaScript interpreter, works like Python interpreter

```
$ node
> console.log("hello")
hello
```

Data Types

- Numbers
  - `var pi = 16;`
  - Everything is a float
- Strings (no char)
  - `+` concats strs
  - `"Hello World".length === 11;`
  - `"Hello World".toUpperCase(),`etc.
- Arrays
  - `var chickens = [ "Myrtle II", "Magda" ];`
- Objects
  - `var name = { first:"Drew", last:"DeOrio" };`

Data Types

- Types change at runtime
- AKA untyped
- AKA loosely typed
- `var x = 10;        //it’s a number`
- `x = "Hello World"; //now it’s a string!`

Simple functions

- Write a simple function

```
function add(a, b) {
    return a + b;
}
```

- Call it

```
var total = add(1,10);
document.write(total);
```

Call-by-what?

- JavaScript variables are call-by-object
    - AKA call by object-sharing
- Similar to Python, Java, Ruby, *et. al.*

1. Primitive types are passed by value
2. Objects are passed by reference
3. References themselves are passed by value

- Exercise: write three functions that demonstrate these three rules

Call-by-object

1. Primitive types are passed by value

```
function f(input) {
    input = 17;
}
var num = 0;
f(num);
console.log(num); //0
```

Call-by-object

2. Objects are passed by reference

```
function f(input) {
    input.first = ”Mike";
    input.last = ”Cafarella";
}

var professor = {first:"Drew", last:"DeOrio"};
f(professor);
console.log(professor);
   //Object {first:”Mike", last:”Cafarella"}
```

Call-by-object

3. References themselves are passed by value

```
function f(input) {
    input = {first:”Mike", last:”Cafarella"};
}
var professor = {first:"Drew", last:"DeOrio"};
f(professor);
console.log(professor);
  //Object {first:"Drew", last:"DeOrio"}
```

A Fast Introduction

- The usual loops: while, for, do

```
for (var x = 1; x < 10; x++) {
  document.write(x + "<p>");
}
```

- Usual controls: if, switch, break, return

```
if (x === 10) {
  document.write("Success!");
} else {
  document.write("Failure!");
}
```

- Values evaluate to true unless:
  - `false, null`
  - `Undefined`
  - `''` (empty string)
  - `The numbers 0, NaN`

Equality and type coercion

- == does type coercion, === does not
- false === ‘false’ is false
- false === ‘0’ is false
- BUT
- false == ‘false’ is false
  false == ‘0’ is true

- **Moral**: always use ===

Blocks and scope

- Functions create scopes (like C/C++)

```
function add (a, b) {
    return a + b + x; //error “x”
}
```

- Blocks do not create scopes (unlike C/C++)

```
if (true) {
    var x = 10;
}
document.write(x); //no problem
```

- **Moral:** Declare all variables at top of function

Global variables

- Like other languages, variables in the outermost scope are global

```
var x = 10; //global
function add(a, b) {
    return a + b + x;
}
document.write(add(2, 2)) //14
document.write(x) //10
```

Global variables

- Unlike many other languages, variables declared without the `var` keyword are also global!

```
function add(a, b) {
    x = 10; //global!
    return a + b + x;
}
document.write(add(2,2)) //14
document.write(x) //10 !!!
```

- **var** will create a variable in local scope;

- no **var** will climb scope chain until it overrides the variable, or adds a global

Global variables

- `"use strict"` at the top of your file helps avoid bugs

```
"use strict";
function add(a, b) {
    x = 10; //Error x is not defined
    return a + b + x;
}
```

Functions

- Functions are first class
  - This means they can be created, destroyed, passed as inputs, and returned as outputs
- Functions as variables

```
var add = function(a, b) {
    return a + b;
}
```

- Call it

```
var total = add(1,10);
console.log(total);
```

Functions

- Lots of ways to use functions
- 1. Standalone:

```
function add(a, b) { return a + b; }
var total = add(1,10);
```

- 2. As a method of an object

```
var cseclass = {
  "name" : "eecs485",
  "timeslot" : "MW1030-12",
   getgrade : function(student) {
     return "A";
   }
};

cseclass.getgrade("jane");
```

Functions

- 3. Apply-style:

```
  var numlist = [1, 2, 3, 4];
  var total = add.apply(null, numlist);
  // total === 10
```

- 4. Constructor-style (invoked w/new):

```
var List = function(v, n) {
  this.v = v;
  this.n = n;
}

varl = new List(1, 2);
```

this

- Available in all functions
- Bound according to how fn is called

How invoked?	**`this` binding**
Standalone	To global object
Method-style	To object that holds fn
Apply-style	`add.apply(null, numlist)`
Constructor-style	To a new object returned by `new`

- When new is used, it changes return
  - List(v, n) returns the obj, unless return yields an obj

Functions

- Secret parameter for all functions: `arguments`

```
var emitAll = function() {
  for (var i = 0; i < arguments.length; i++) {
    document.write(arguments[i] + “<p>”);
  }
}
```

- Remember, functions are objects and (weirdly) they can have methods!

```
add.apply(null, numlist)
```

Introducing Closures

- In JavaScript, we can define inner functions

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

Introducing Closures

- In JavaScript, we can define inner functions

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

- Inner functions have access to outer variables
- Lexically scoped name binding
- This is called a closure

Closures

```
function counter() {
  var count = 0;
  function increment() {
      count += 1;
      return count;
  }
  increment();
  return count;
};
var i = counter()
console.log(i); //1
console.log(i); //1
console.log(i); //1
```

- No way to access `increment()` from the outside

Closures

```
var count = 0;
var counter = {
    increment: function() {
      count += 1;
    },
    getValue: function() {
      return count;
    }
}
var i = counter;
console.log(i.getValue()); //0
i.increment(); console.log(i.getValue()); //1
i.increment(); console.log(i.getValue()); //2
```

- HACK: use an Object
- Problem: global variable!

Closures for encapsulation

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

- Can make `count` a private member with a closure

Closures for encapsulation

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

- return creates a new Object
- Exercise: write a test to see if this works

Closures for encapsulation

```
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!

Closures for encapsulation

```
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

Closures Explained Again

- 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();
```

Closures Explained Again

- `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
```

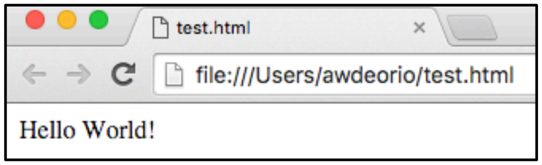
Closures Explained Again

- 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*

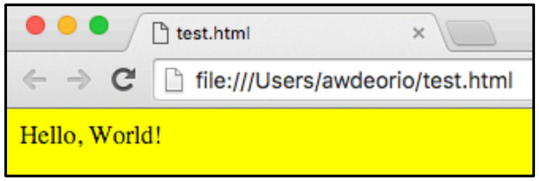
```
var i = counter();
console.log(i.getValue()); //0
i.increment();
console.log(i.getValue()); //1
i.increment();
console.log(i.getValue()); //2
```

Exercise

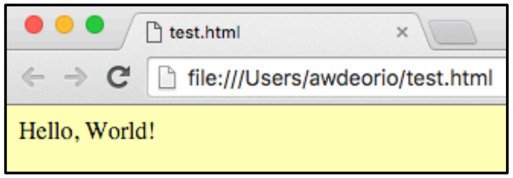
- Write a function called `fade` that changes the background color to yellow, then fades to white



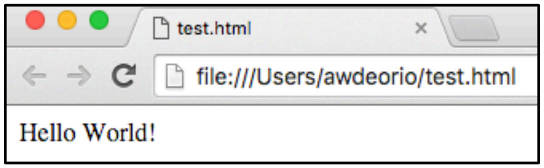
Hello World!



Hello, World!



Hello, World!



Hello World!

Exercise

- Write a function called `fade` that changes the background color to yellow, then fades to white
- Write an inner function called `fade_step` and schedule it to be called again with `setTimeout(fade_step, 100);`
- Use a closure to remember the "step" of the fade
- This code changes the background, as step changes, it fades

```
var step = 1;
var hex = step.toString(16);
var color = '#FFFF' + hex + hex;
var node = document.body;
node.style.backgroundColor = color;
```

Caveat: to test the above code, you'll need to wrap it with

```
window.onload = function () { /*...*/ }
```

Solution

```
var fade = function (node) {
  var step = 1;
  var fade_step = function () {
    var hex = step.toString(16);
    var color = '#FFFF' + hex + hex;
    node.style.backgroundColor = color;
    if (step < 15) {
      step += 1;
      setTimeout(fade_step, 100);
    }
  };
  setTimeout(fade_step, 100);
};

fade(document.body);
```