Organizing Code in Web Apps

SWE 432, Fall 2017

Design and Implementation of Software for the Web

Today

- HW1 assigned today. Due in 1 week

- Lecture: Organizing code in web apps
  - Some basics on how and why to organize code
  - Closures
  - Classes
  - Modules

JavaScript Files

- Can create a plain text file with .js extension that contains JavaScript code.
- Can contain whatever expressions and code you'd like.
- We'll look at how to run these next lecture.
- For now, use a pastebin.

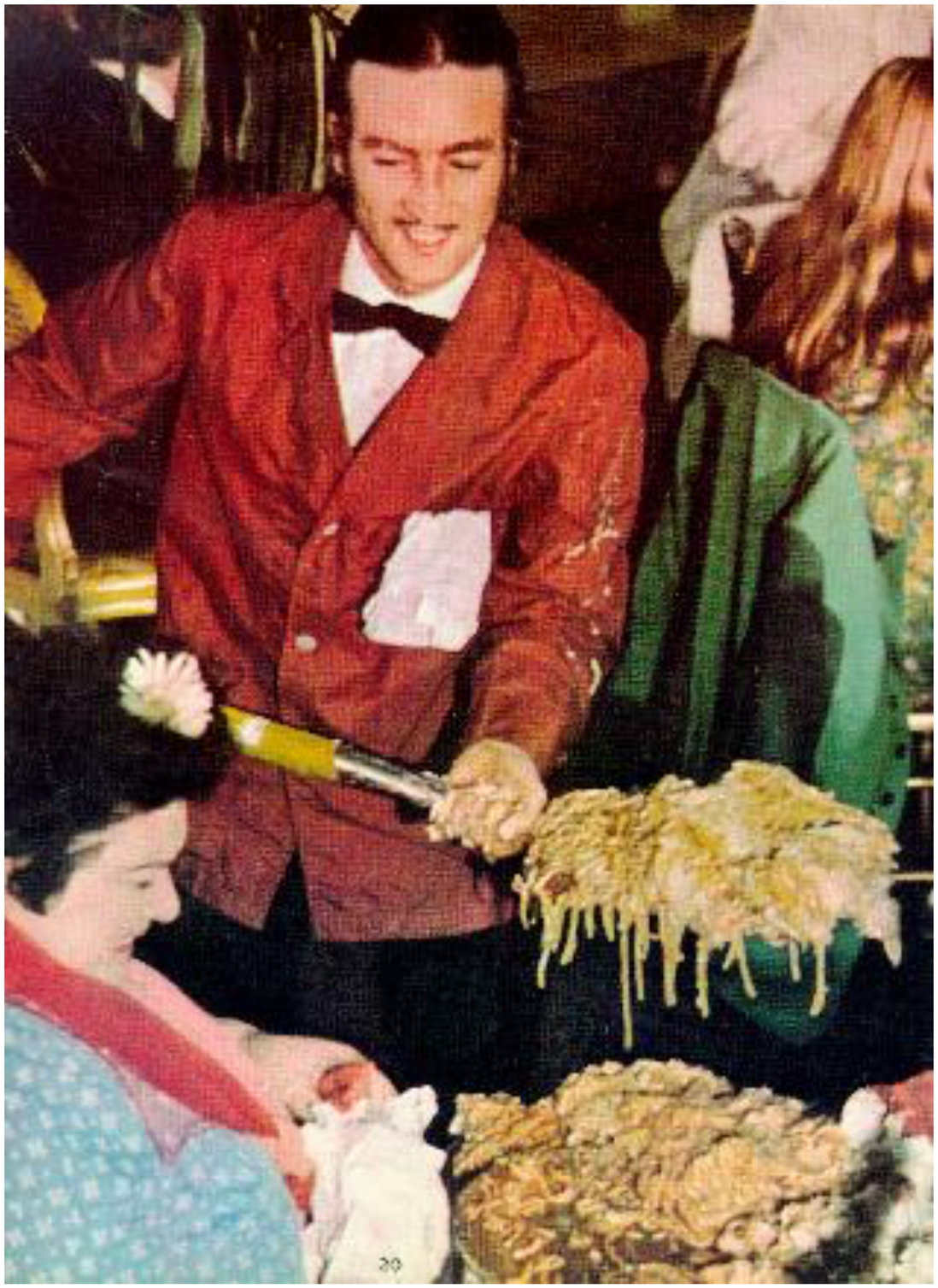
file:///Users/tlatoza/Downloads/classes/constructors.js

```
// The MIT License (MIT)
// Copyright (c) 2016 Training 4 Developers, Inc.
//
// Permission is hereby granted, free of charge, to any person obtaining a copy of thi
deal in the Software without restriction, including without limitation the rights to u
copies of the Software, and to permit persons to whom the Software is furnished to do
//
// The above copyright notice and this permission notice shall be included in all copi
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF C
SOFTWARE.

// Instructions: to run the following code demonstration,
// install Node.js 4 (https://nodejs.org/en/) or later,
// make sure node.exe is in your path, then execute the
// file from the terminal with the following command:
//
// > node constructors.js

"use strict";

// ES5 Constructor Function
// function Person(firstName, lastName) {
//      this.firstName = firstName;
//      this.lastName = lastName;
// }

// the name of the ES5 constructor
// function is name of the ES6 class
class Person {

        // observe there is no "function" keyword
        // also, the word "constructor" is used, not "Person"
        constructor(firstName, lastName) {

                // this represents the new object being
                // created and initialized
                this.firstName = firstName;
                this.lastName = lastName;

        }
}

var person = new Person("Bob", "Smith");

// outputs "Bob"
console.log(person.firstName);

// outputs "Smith"
console.log(person.lastName);
```

Combining files

- What happens when there are two or more places where JavaScript code is declared?
- Java: each has it's own scope (e.g., Class)
- JavaScript: code is concatenated together
  - Can reference variables declared in a different file
  - Sounds great! So convenient…
  - But is there a downside here?

Spaghetti Code



Brian Foote and Joe Yoder

```
    eqCtl.innerHTML = val;
}

function clearNumbers() {
    lastNumber = null;
    equalsPressed = operatorSet = false;
    setVal('0');
    setEquation('');
}

function setOperator(newOperator) {
    if (newOperator == '=') {
        equalsPressed = true;
        calculate();
        setEquation('');
        return;
    }

   if (!equalsPressed) calculate();
    equalsPressed = false;
    operator = newOperator;
    operatorSet = true;
    lastNumber = parseFloat(currNumberCtl.innerHTML);
    var eqText = (eqCtl.innerHTML == '') ?
        lastNumber + ' ' + operator + ' ' :
        eqCtl.innerHTML + ' ' + operator + ' ';
    setEquation(eqText);
}

function numberClick(e) {
    var button = (e.target) ? e.target : e.srcElement;
    if (operatorSet == true || currNumberCtl.innerHTML == '0') {
        setVal('');
        operatorSet = false;
    }
    setVal(currNumberCtl.innerHTML + button.innerHTML);
    setEquation(eqCtl.innerHTML + button.innerHTML);
}

function calculate() {
    if (!operator || lastNumber == null) return;
    var currNumber = parseFloat(currNumberCtl.innerHTML),
        newVal = 0;
   switch (operator) {
        case '+':
            newVal = add(lastNumber, currNumber);
            break;
        case '-':
```

…aka big ball of mud aka shanty town code

Brian Foote and Joe Yoder

Bad Code "Smells"

- Tons of not-very related functions in the same file
- No/uninformative comments
- Hard to understand

Design Goals

- Within a component
  - Cohesive
  - Complete
  - Convenient
  - Clear
  - Consistent
- Between components
  - Low coupling

Cohesion and Coupling

- Cohesion is a property or characteristic of an individual unit
- Coupling is a property of a collection of units
- High cohesion GOOD, high coupling BAD
- Design for change:
  - Reduce interdependency (coupling): You don't want a change in one unit to ripple throughout your system
  - Group functionality (cohesion): Easier to find things, intuitive metaphor aids understanding

Design for Reuse

- Why?
  - Don't duplicate existing functionality
  - Avoid repeated effort
- How?
  - Make it easy to extract a single component:
    - Low **coupling** between components
    - Have high **cohesion** with

Design for Change

- Why?
  - Want to be able to add new features
  - Want to be able to easily **maintain** existing software
    - Adapt to new environments
    - Support new configurations
- How?
  - Low **coupling** - prevents unintended side effects
  - High **cohesion** - easier to find things

Closures

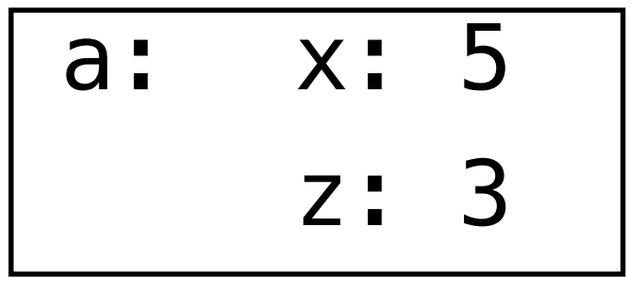
- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is that function and a **stack frame** that is allocated when a function starts executing and **not freed** after the function returns

Closures & Stack Frames

- What is a stack frame?
  - Variables created by function in its execution
  - Maintained by environment executing code

```
function a() {
    var x = 5, z = 3;
    b(x);
}
function b(y)  {
    console.log(y);
}
a();
```

Contents of memory:



Stack frame

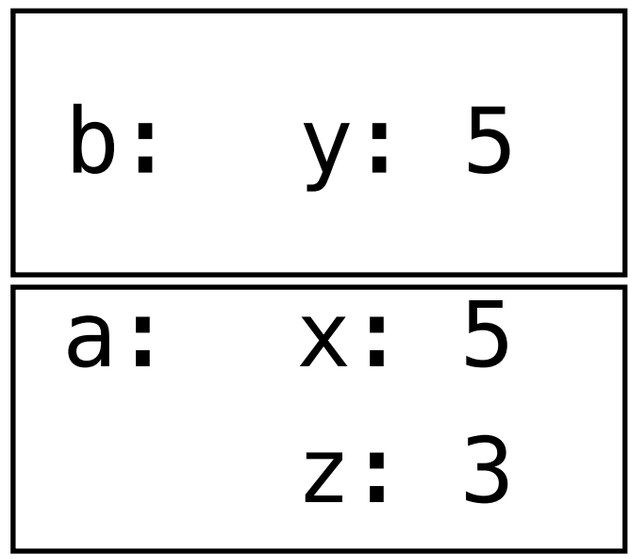
Function called: stack frame created

Closures & Stack Frames

- What is a stack frame?
  - Variables created by function in its execution
  - Maintained by environment executing code

```
function a() {
    var x = 5, z = 3;
    b(x);
}
function b(y) {
    console.log(y);
}
a();
```

Contents of memory:



Stack frame

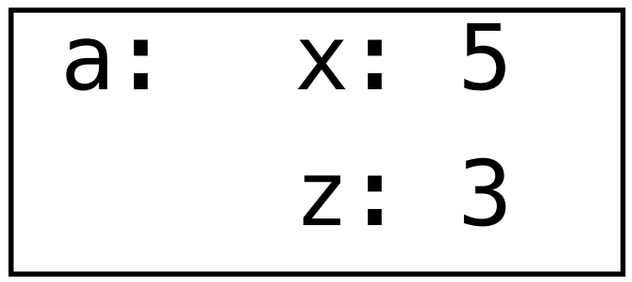
Function called: new stack frame created

Closures & Stack Frames

- What is a stack frame?
  - Variables created by function in its execution
  - Maintained by environment executing code

```
function a() {
    var x = 5, z = 3;
    b(x);
}
function b(y)  {
    console.log(y);
}
a();
```

Contents of memory:



Stack frame

Function returned: stack frame popped

Closures

- Closures are expressions that work with variables in a specific context
- Closures contain a function, and its needed state
  - Closure is a stack frame that is allocated when a function starts executing and not freed after the function returns
- That state just refers to that state by name (sees updates)

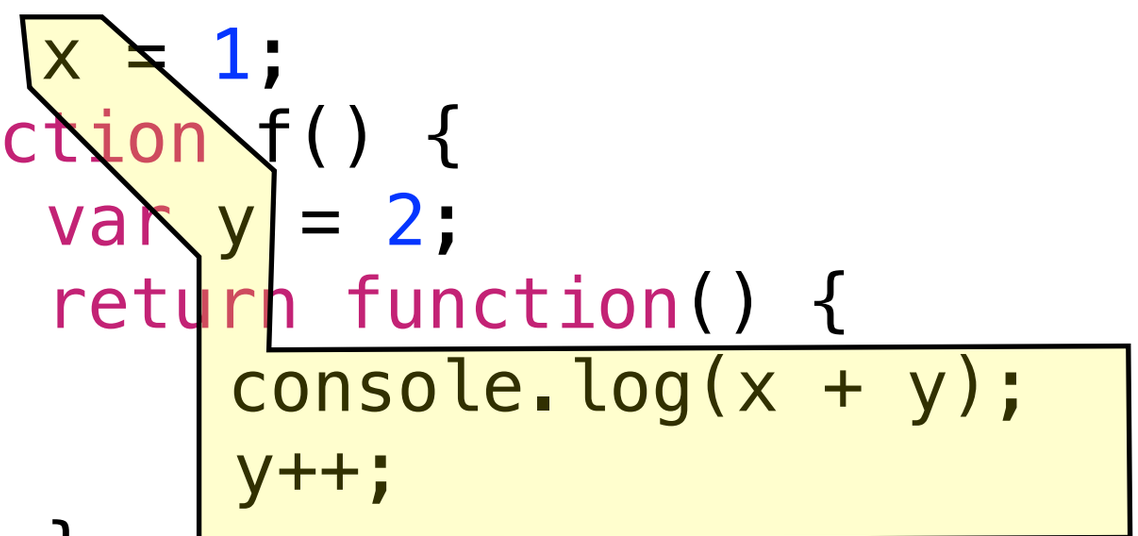
```
var x = 1;
function f() {
    var y = 2;
    return function() {
        console.log(x + y);
        y++;
    };
}
var g = f();
g();              // 1+2 is 3
g();              // 1+3 is 4
```

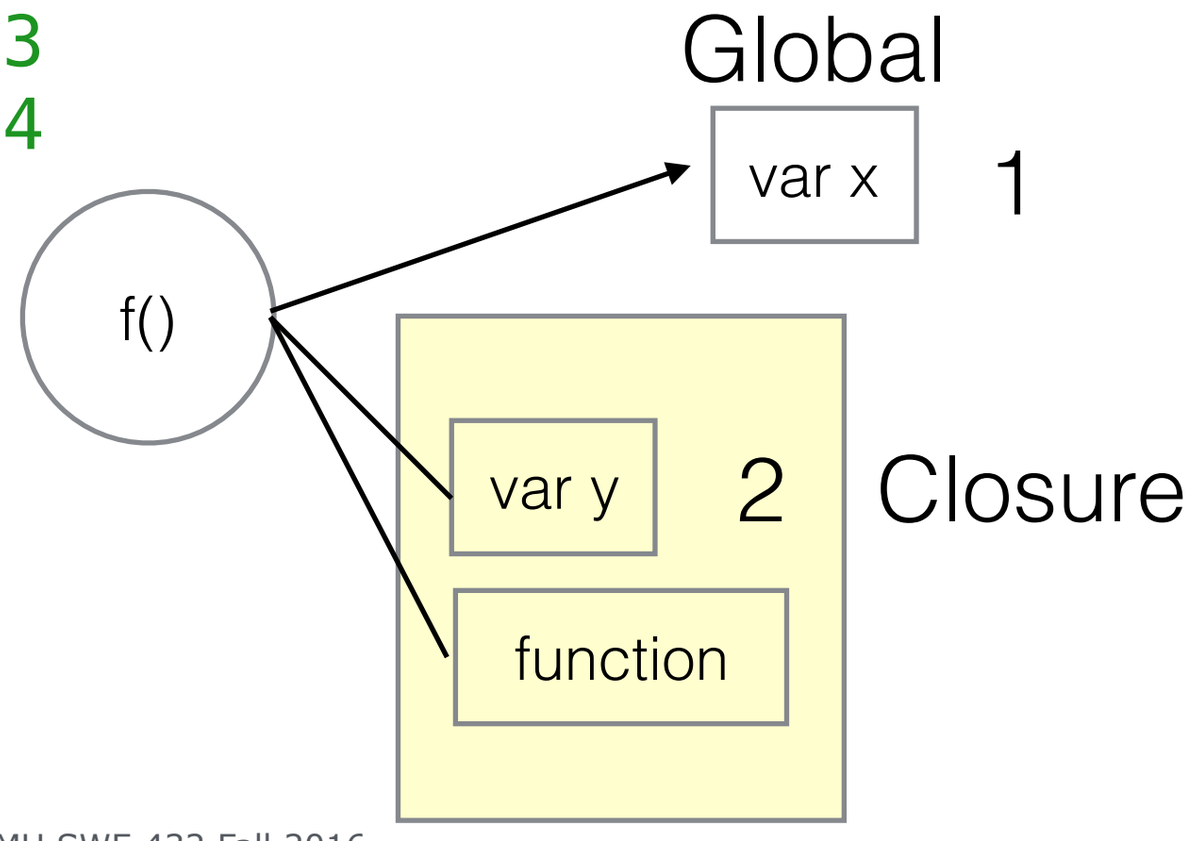
This function attaches itself to x and y so that it can continue to access them.

It "**closes up**" those references

Closures

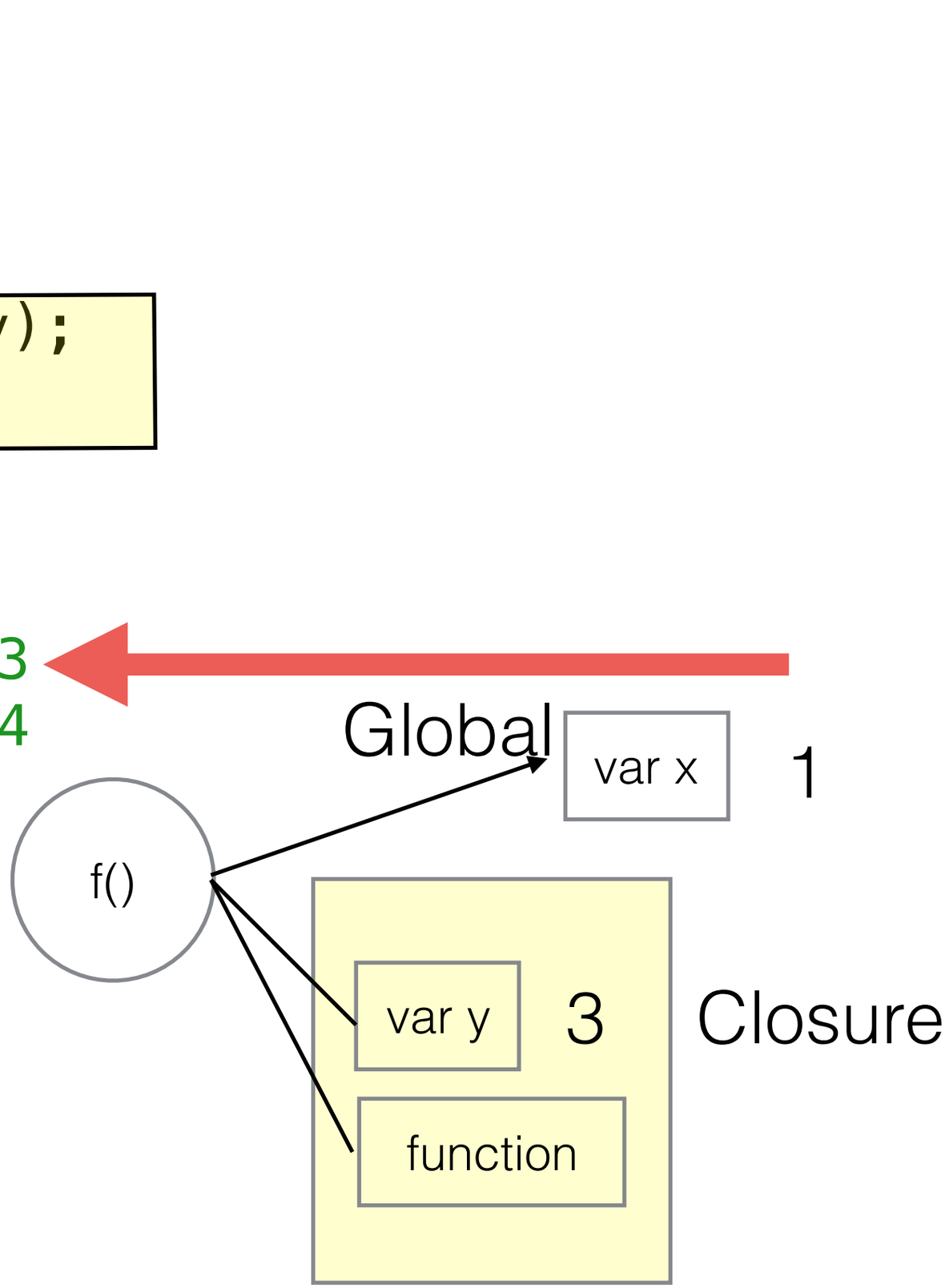
```
var x = 1;
function f() {
    var y = 2;
    return function() {
        console.log(x + y);
        y++;
    };
}
var g = f();
g();          // 1+2 is 3
g();          // 1+3 is 4
```





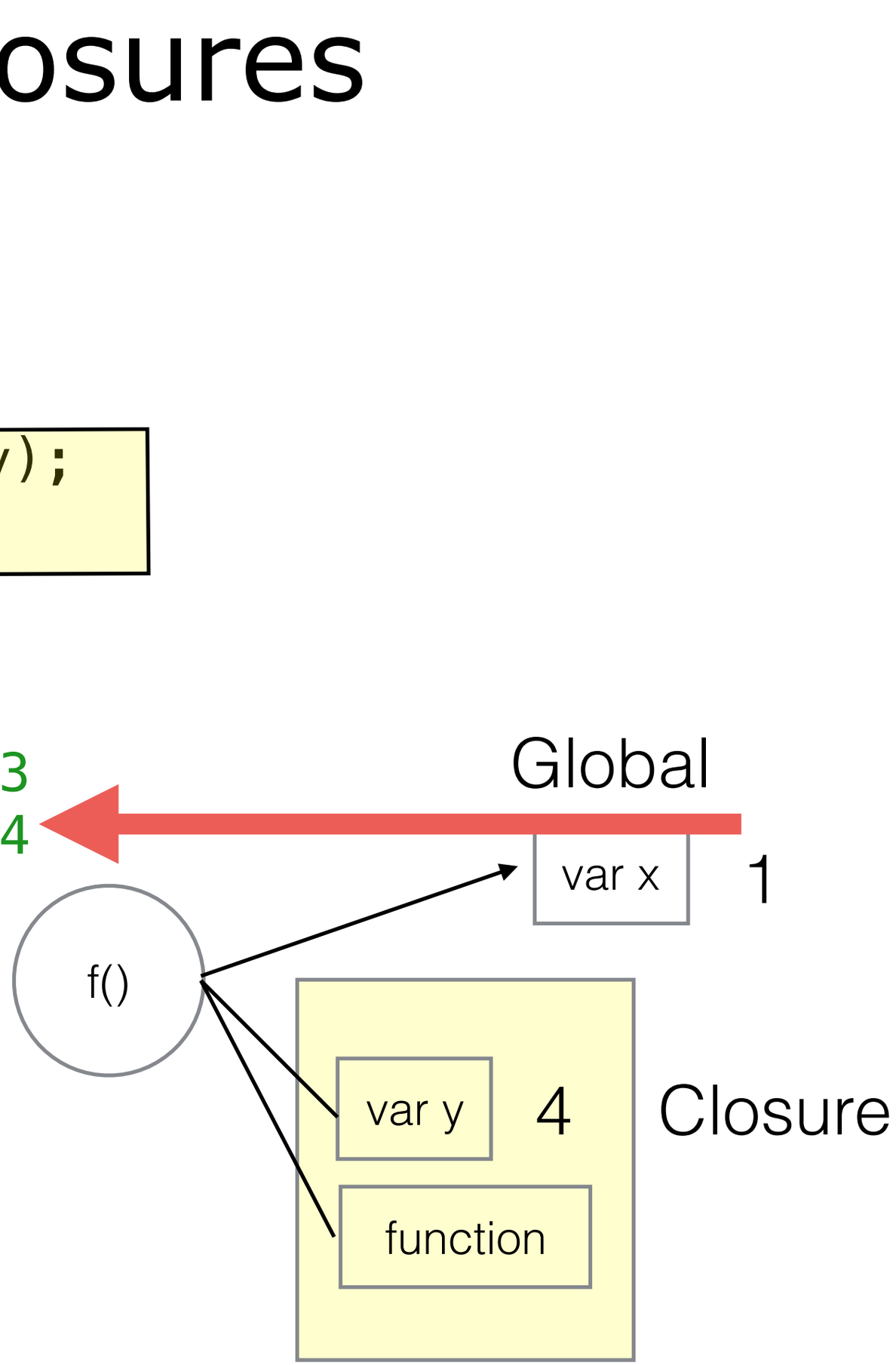
Closures

```
var x = 1;
function f() {
     var y = 2;
     return function() {
          console.log(x + y);
          y++;
     };
}
var g = f();
g();              // 1+2 is 3
g();              // 1+3 is 4
```



Closures

```
var x = 1;
function f() {
     var y = 2;
     return function() {
          console.log(x + y);
          y++;
     };
}
var g = f();
g();              // 1+2 is 3
g();              // 1+3 is 4
```



Modules

- We can do it with closures!
- Define a function
  - Variables/functions defined in that function are “private”
  - Return an object - every member of that object is public!
- Remember: Closures have access to the outer function’s variables even after it returns

Modules with Closures

```
var facultyAPI = (function(){
  var faculty = [{name:"Prof Bell", section: 2}, {name:"Prof
LaToza", section:1}];

  return {
   getFaculty : function(i)
   {
    return faculty[i].name + " ("+faculty[i].section +")";
   }

 };
})();

console.log(facultyAPI.getFaculty(0));
```

This works because inner functions have visibility to all variables of outer functions!

Closures gone awry

```
var funcs = [];
for (var i = 0; i < 5; i++) {
    funcs[i] = function() { return i; };
}
```

What is the output of `funcs[0]()`?

>5

Why?

Closures retain a *pointer* to their needed state!

Closures under control

Solution: IIFE - Immediately-Invoked Function Expression

```
function makeFunction(n)
{
    return function(){ return n; };
}
for (var i = 0; i < 5; i++) {
  funcs[i] = makeFunction(i);
}
```

Why does it work?

Each time the anonymous function is called, it will create a **new variable `n`**, rather than reusing the same variable `i`

Shortcut syntax:

```
var funcs = [];
for (var i = 0; i < 5; i++) {
  funcs[i] = (function(n) {
    return function() { return n; }
  })(i);
}
```

Exercise: Closures

```
var facultyAPI = (function(){
  var faculty = [{name:"Prof LaToza", section: 1},
{name:"Prof Bell”, section:2}];

  return {
   getFaculty : function(i)
   {
    return faculty[i].name + " ("+faculty[i].section +")";
   }

 };
})();

console.log(facultyAPI.getFaculty(0));
```

https://jsfiddle.net/hkcq5vpa/

Here’s our simple closure. Add a new function to create a new faculty, then call **`getFaculty`** to view their formatted name.

Classes

- ES6 introduces the `class` keyword
- Mainly just syntax

Old

```
function Faculty(first, last, teaches, office)
{
  this.firstName = first;
  this.lastName = last;
  this.teaches = teaches;
  this.office = office;
  this.fullName = function(){
    return this.firstName + " " + this.lastName;
  }
}
var profLaToza = new Faculty("Thomas", "LaToza", "SWE432", "ENGR 4431”);
```

New

```
class Faculty {
    constructor(first, last, teaches, office)
    {
        this.firstName = first;
        this.lastName = last;
        this.teaches = teaches;
        this.office = office;
    }
    fullname() {
        return this.firstName + " " + this.lastName;
    }
}
var profJon = new Faculty("Thomas", "LaToza", "SWE432", "ENGR 4431”);
```

“Member” variables

- Can’t declare member variables explicitly like in Java.
- Instead, create them implicitly by referencing “this”

```
class Faculty {
    constructor(first, last, teaches, office)
    {
        this.firstName = first;
        this.lastName = last;
        this.teaches = teaches;
        this.office = office;
    }
    fullname() {
        return this.firstName + " " + this.lastName;
    }
}
var profJon = new Faculty("Thomas", "LaToza", "SWE432", "ENGR 4431”);
```

Classes - Extends

`extends` allows an object created by a class to be linked to a "`super`" class. Can (but don't have to) add parent constructor.

```
class Faculty {
    constructor(first, last, teaches, office)
    {
        this.firstName = first;
        this.lastName = last;
        this.teaches = teaches;
        this.office = office;
    }
    fullname() {
        return this.firstName + " " + this.lastName;
    }
}

class CoolFaculty extends Faculty {
    fullname() {
        return "The really cool " + super.fullname();
    }
}
```

Classes - static

`static` declarations in a `class` work like in Java

```
class Faculty {
    constructor(first, last, teaches, office)
    {
        this.firstName = first;
        this.lastName = last;
        this.teaches = teaches;
        this.office = office;
    }
    fullname() {
        return this.firstName + " " + this.lastName;
    }
    static formatFacultyName(f) {
        return f.firstName + " " + f.lastName;
    }
}
```

Demo: Classes

Modules

- With ES6, there is finally language support for modules
- Module must be defined in its own JS file
- Modules **`export`** declarations
  - Publicly exposes functions as part of module interface
- Code **`imports`** modules (and optionally only parts of them)
  - Specify module by path to the file

Modules (ES6) - Export Syntax

```
var faculty = [{name:"Prof Bell", section: 2}, {name:"Prof
LaToza", section:1}];
export function getFaculty(i) {
    // ..
}
export var someVar = [1,2,3];
```

Label each declaration with "export"

```
var faculty = [{name:"Prof Bell", section: 2}, {name:"Prof
LaToza", section:1}];
var someVar = [1,2,3];
function getFaculty(i) {
    // ..
}
export {getFaculty, someVar};
export {getFaculty as aliasForFunction, someVar};


export default function getFaculty(i){...
```

Or name all of the exports at once

Can rename exports too

Default export

Modules (ES6) - Import Syntax

- Import specific exports, binding them to the same name

```
import { getFaculty, someVar } from "myModule";
getFaculty()...
```

- Import specific exports, binding them to a new name

```
import { getFaculty as aliasForFaculty } from
"myModule";
aliasForFaculty()...
```

- Import default export, binding to specified name

```
import theThing from "myModule";
theThing()... -> calls getFaculty()
```

- Import all exports, binding to specified name

```
import * as facModule from "myModule";
facModule.getFaculty()...
```

Classes Exercise

- Build a Course Class
- Should have following member variables:
  - students (Array of objects with firstname and lastname)
  - instructor (object with firstname, lastname, office #)
  - meeting times (Array of objects with string of meeting time)
  - classroom (String)
- Public functions:
  - Constructor
  - Add a student to the class
  - Get an Array of meeting times for a course

Readings for next time

- Intro to Git: https://try.github.io
- Intro to Node Package Manager (NPM): https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/