

SWE 432 -Web Application Development

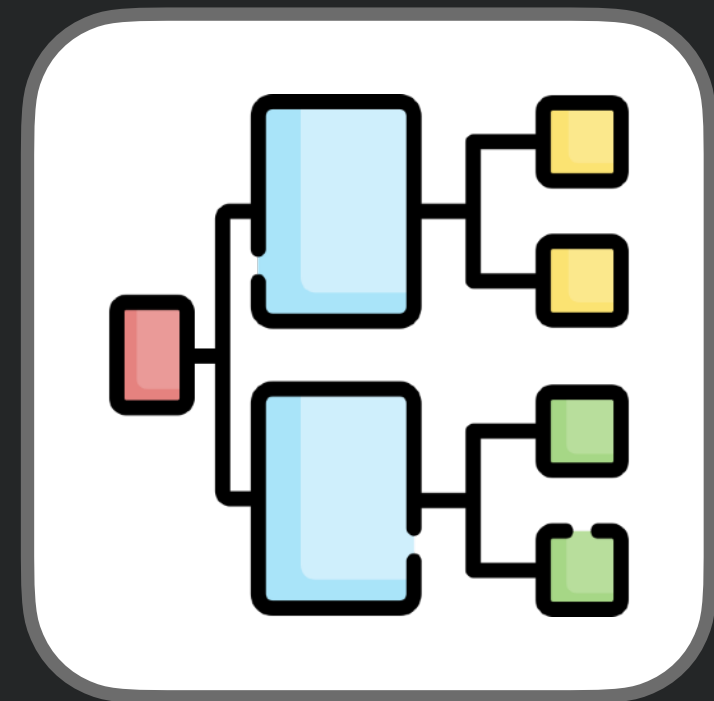
Fall 2022



George Mason
University

Dr. Kevin Moran

Week 2: Organizing Code





Administrivia

- *HW Assignment 1* - Due Before Class
Next Week (Sept. 8th)



HW Assignment I

Overview

In this homework assignment, you will download a JSON dataset from the web and write a simple data analytics package in JavaScript to answer eight questions about your dataset.

Assignment Instructions

Step 1: Download a JSON dataset from a website

In this step, you will collect a JSON dataset containing at least 100 rows (i.e., entries) from a website. You are free to choose whatever data source you'd like. Note that some, but not all, data sources may first require you to obtain an API key by creating an account with the data provider. You should not choose any API that requires you to authenticate using OAuth, as we have not yet covered OAuth.

You may, but are not required to, choose a data set from one of the following:

- [Open Data DC](#)
- [Public APIs](#) (many of these are APIs for performing computation and are NOT datasets, check carefully)
- [DC Metro](#)

After choosing an API, you should collect a dataset containing at least 100 records in JSON format. It's fine if your dataset contains more than 100 records. If your dataset is very large (> 10,000 records), you may wish to choose a subset of the dataset to enable you to test your code more quickly in the following steps.



HW Assignment I

Step 2: List eight (8) questions you will answer about your dataset

Now that you've found a dataset, what insights can you extract from this dataset? In this step, you will write a list of eight (8) questions about your dataset. Each question should describe a statistic to compute from your dataset.

For example, if your dataset is city demographic data, you might have the following questions:

1. What is the average age of residents?
2. What is the average year over year growth rate?
3. What is the fastest growing city?
4. What is the median population density?
5. What is the city with the highest population density?
6. Which is the average age of small cities with less than 100,000 people?
7. What is the city with the oldest population?
8. What is the city with the least amount of new home construction per capita?

In order to more easily satisfy the requirements of step 3, you are encouraged to have a diversity of question types.



HW Assignment I

Step 3: Implement a JavaScript program to answer your questions

In this step, you will now create a JavaScript program to compute the answers to your eight questions using your JSON dataset. For each of the eight questions, your program should output to the JavaScript console (1) the question and (2) the answer. For example, if your question was “What is the fastest growing city?”, your program should write to the console: “What is the fastest growing city? Springfield”

Your program must use all of the following JavaScript features:

- **Variable declarations**
 - Let statement
 - Const statement
- **Functions**
 - Arrow function
 - Default values
 - Array.map()
- **Loops**
 - For of statement



HW Assignment I

Your program must use all of the following JavaScript features:

- **Variable declarations**
 - Let statement
 - Const statement
- **Functions**
 - Arrow function
 - Default values
 - Array.map()
- **Loops**
 - For of statement
 - For in statement
- **Collections**
 - Instance of a Map or Set collection (only 1 is required)
- **Strings**
 - Template literal
- **Classes**
 - Class declaration
 - Constructor
 - Using an instance variable with this



HW Assignment I

Submission instructions

- Submit your HW through **replit**. Please follow [these instructions](#) for signing up for the replit account and accessing HW1. You should be able to complete this project using only the replit web interface. However, if you would like to work locally on your machine, you can code using your preferred environment and then upload the final `.js` files through replit.

[Click Here to View the Replit Instructions](#)

The HW assignment submission should consist of two `.js` files one called `data.js` containing the following:

1. A comment containing a URL where your JSON dataset from step (1) can be found
2. Your JSON dataset from step (1) (or a subset of the dataset that is at least 100 records)

and another call `script.js` containing the following:

1. A comment with your full name and G-number
2. Your program from step (3)

[Click Here to Access the Assignment via Replit](#)



Signing Up For Replit



Replit instructions

Using replit for Assignments

This semester, we will be using [replit](#) for certain homework assignments. Replit is a cloud-based IDE that allows for programming with a range of different web technologies. This will allow us to facilitate answering questions and providing detailed feedback on your assignments. Please follow the steps below to sign up for a replit account and access assignments.

Signing up for a replit account

The first thing that you need to do before you can access replit, is to sign up for a free account **using your gmu email address**. This will allow you to join our class.

You can sign up for a free replit account using [this link](#).

Joining the SWE-432 Course Team

In replit, our course is organized as a "team", and you will need to join this team in order to access the class assignments. you can join the SWE-432 replit team using the button below:

[Click Here to Sign Up for SWE-432 on Replit](#)

Table of contents

[Using replit for Assignments](#)

[Signing up for a replit account](#)

[Joining the SWE-432 Course Team](#)

[Accessing Individual HW Assignments](#)

Class Overview





Class Overview

- Today's Class - Organizing Code in Web

Apps: How can we build comprehensible and maintainable web apps?

- Thursday's Class - Javascript Tools and

Testing: Exploring Node and Testing Strategies

Organizing Code in Web Apps





First Half of Lecture

- Some basics on how and why to organize code (SWE!)
- Closures
- Classes
- Modules

For further reading:

<http://stackoverflow.com/questions/111102/how-do-javascript-closures-work>



Running Javascript

- More on this the next class
- Some options for now
 - a pastebin (e.g., Replit, JSFiddle)
 - an IDE (e.g, VSCode, Webstorm)
 - Webstorm is free for students:
 - <https://www.jetbrains.com/student/>

History + Motivation



“Back in my day before ES6 we didn’t have your fancy modules”

Spaghetti Code



```

window.onload = function () {
    eqCtl = document.getElementById('e');
    currNumberCtl = document.getElemen
};

var eqCtl,
    currNumberCtl,
    operator,
    operatorSet = false,
    equalsPressed = false,
    lastNumber = null;

function add(x,y) {
    return x + y;
}

function subtract(x, y) {
    return x - y;
}

function multiply(x, y) {
    return x * y;
}

function divide(x, y) {
    if (y == 0) {
        alert("Can't divide by 0");
        return 0;
    }
    return x / y;
}

function setVal(val) {
    currNumberCtl.innerHTML = val;
}

function setEquation(val) {
    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 == '')
        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 '-':
            newVal = subtract(lastNumber, currNumber);
            break;
        case '*':
            newVal = multiply(lastNumber, currNumber);
            break;
        case '/':
            newVal = divide(lastNumber, currNumber);
            break;
    }
    setVal(newVal);
    lastNumber = newVal;
}

```

```

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 == '')
        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 '-':
            newVal = subtract(lastNumber, currNumber);
            break;
        case '*':
            newVal = multiply(lastNumber, currNumber);
            break;
        case '/':
            newVal = divide(lastNumber, currNumber);
            break;
    }
    setVal(newVal);
}

```




Bad Code “Smells”

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

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height +
              'x' + width)
            this.resize(width, height).write(dest + 'w' + width +
              'h' + height)
            if (err) console.log(err)
          })
        }
      })
    })
  }
})
```


Code Smell Research



When and Why Your Code Starts to Smell Bad

Michele Tufano*, Fabio Palomba†, Gabriele Bavota‡, Rocco Oliveto§, Massimiliano Di Penta¶, Andrea De Lucia†, Denys Poshyvanyk¹
*The College of William and Mary, Williamsburg, VA, USA - †University of Salerno, Fisciano (SA), Italy, ‡Free University of Bozen-Bolzano, Italy - §University of Molise, Pesche (IS), Italy, ¶University of Sannio, Benevento, Italy

Abstract—In past and recent years, the issues related to managing technical debt received significant attention by researchers from both industry and academia. There are several factors that contribute to technical debt. One of these is represented by code bad smells, *i.e.*, symptoms of poor design and implementation choices. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on *when* and *why* bad smells are introduced by developers, and we conducted a large empirical study over the change history of 200 open source projects from different software ecosystems and the circumstances and reasons behind their introduction. Our study required the mining of over 0.5M commits, and the manual analysis of 9,164 of them (*i.e.*, those identified as *smell-introducing*). Our findings mostly contradict common wisdom stating that smells are being introduced during evolutionary tasks. In the light of our results, we also call for the need to develop a new generation of recommendation systems aimed at properly planning smell refactoring activities.

1. INTRODUCTION

Technical debt is a metaphor introduced by Cunningham to indicate “*not quite right code which we postpone making it right*” [18]. The metaphor explains well the trade-off between delivering the most appropriate but still “*not quite right*” product, in the shortest time possible [12]. While the repercussions of “*not quite right*” code quality have been empirically assessed, there is still only anecdotal evidence on *when* and *why* bad smells are introduced by developers, and we conducted a large empirical study over the change history of 200 open source projects from different software ecosystems and the circumstances and reasons behind their introduction. Our study required the mining of over 0.5M commits, and the manual analysis of 9,164 of them (*i.e.*, those identified as *smell-introducing*). Our findings mostly contradict common wisdom stating that smells are being introduced during evolutionary tasks. In the light of our results, we also call for the need to develop a new generation of recommendation systems aimed at properly planning smell refactoring activities.

When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)

Michele Tufano¹, Fabio Palomba², Gabriele Bavota³, Rocco Oliveto⁴, Massimiliano Di Penta⁵, Andrea De Lucia², Denys Poshyvanyk¹
¹The College of William and Mary, Williamsburg, VA, USA ²University of Salerno, Fisciano (SA), Italy, ³Università della Svizzera italiana (USI), Switzerland, ⁴University of Sannio, Benevento (BN), Italy, ⁵University of Molise, Pesche (IS), Italy, mtufano@email.wm.edu, fpalomba@unisa.it, gabriele.bavota@usi.ch, rocco.oliveto@unimol.it, dipenta@unisannio.it, adelucia@unisa.it, denys@cs.wm.edu

Abstract—Technical debt is a metaphor introduced by Cunningham to indicate “not quite right code which we postpone making it right”. One noticeable symptom of technical debt is represented by code smells, defined as symptoms of poor design and implementation choices. Previous studies showed the negative impact of code smells on the comprehensibility and maintainability of code. While the repercussions of smells on code quality have been empirically assessed, there is still only anecdotal evidence on *when* and *why* bad smells are introduced, what is their *survivability*, and *how* they are removed by developers. To empirically corroborate such anecdotal evidence, we conducted a large empirical study over the change history of 200 open source projects. This study required the development of a strategy to identify smell-introducing commits, the mining of over half a million of commits, and the manual analysis and classification of over 10K of them. Our findings mostly contradict common wisdom, showing that most of the smell instances are introduced when an artifact is created and not as a result of its evolution. At the same time, 80% of smells survive in the system. Among the 20% of removed instances, only 9% are removed as a direct consequence of refactoring operations.

Index Terms—Code Smells, Empirical Study, Mining Software Repositories

removed [14]. This represents an obstacle to effective and efficient management of technical debt. Understanding the typical life-cycle of code quality is of paramount importance in the context of software development. The actions undertaken by developers to improve code quality are of paramount importance in the context of software development. The actions undertaken by developers to improve code quality are of paramount importance in the context of software development.



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*** within a component



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

Organizing Code with Classes





Organizing Code

How do we structure things to achieve good organization?

	Java	Javascript
Individual Pieces of Functional Components	Classes	Classes
Entire libraries	Packages	Modules

Classes



- ES6 introduces the `class` keyword
- Mainly just syntax - still not like Java Classes

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 prof = new Faculty("Kevin", "Moran", "SWE432", "ENGR 4448");
```

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 prof = new Faculty("Kevin", "Moran", "SWE432", "ENGR 4448");
```



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;
    }
}
```

Modules





Modules (ES6)

- 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 Johnson", section: 2}, {name:"Prof Moran", section:1}];  
export function getFaculty(i) {  
    // ..  
}
```

Label each declaration with "export"

```
export var someVar = [1,2,3];
```

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

Or name all of the exports at once

```
export {getFaculty, someVar};
```

Can rename exports too

```
export {getFaculty as aliasForFunction, someVar};
```

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

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()...
```



Patterns for using/creating libraries

- Try to reuse as much as possible!
- Name your module in all lower case, with hyphens
- Include:
 - README.md
 - keywords, description, and license in package.json (from npm init)
- Strive for high cohesion, low coupling
 - Separate models from views
 - How much code to put in a single module?
- Cascades (see jQuery)



Cascade Pattern

- aka “chaining”
- Offer set of operations that mutate object and returns the “this” object
 - Build an API that has single purpose operations that can be combined easily
 - Lets us read code like a sentence
- Example (String):

```
str.replace("k", "R").toUpperCase().substr(0, 4);
```

- Example (jQuery):

```
$("#wrapper")  
  .fadeOut()  
  .html("Welcome")  
  .fadeIn();
```



Cascade Pattern

```
function number(value) {  
  this.value = value;  
  
  this.plus = function (sum) {  
    this.value += sum;  
    return this;  
  };  
  
  this.return = function () {  
    return this.value;  
  };  
  
  return this;  
}  
  
console.log(new number(5).plus(1).return());
```



Bind and This

```
var module = {  
  x: 42,  
  getX: function() {  
    return this.x;  
  }  
}
```

```
var unboundGetX = module.getX;  
console.log(unboundGetX());
```

// expected output: undefined



Binding This

```
var module = {  
  x: 42,  
  getX: function() {  
    return this.x;  
  }  
}
```

```
var unboundGetX = module.getX;  
console.log(unboundGetX()); // expected output: undefined
```

```
var unboundGetX = unboundGetX.bind(module);  
console.log(unboundGetX()); // expected output: 42
```

The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

If "this" is not this



What is _this?

Closures





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:

a:	x: 5
	z: 3

Stack frame

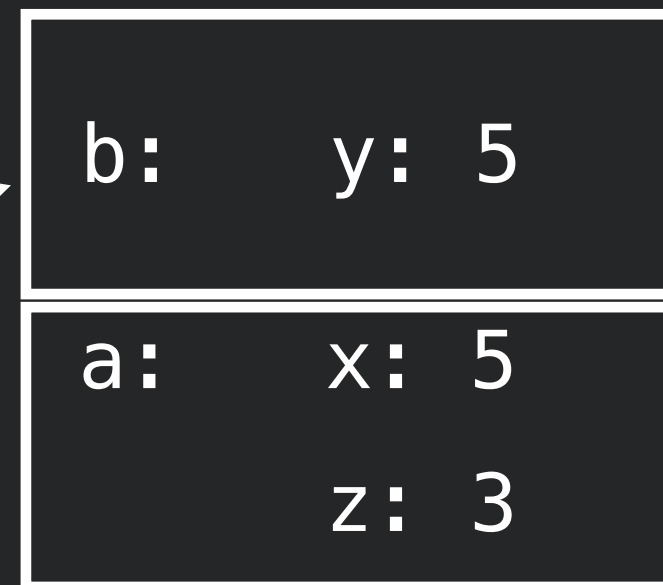
Function called: stack frame created

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: 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:



a:	x: 5
	z: 3

Stack frame

Function called: stack frame created



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

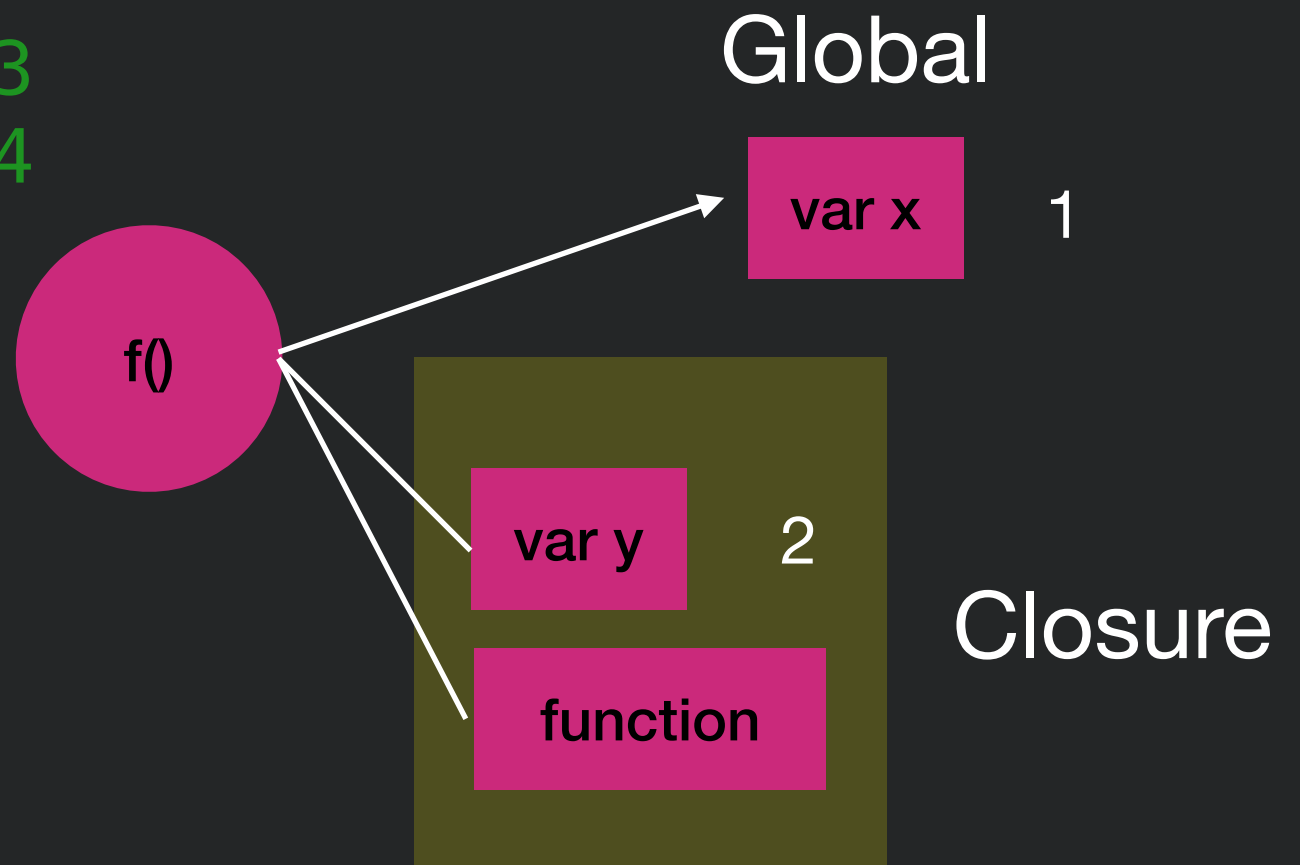
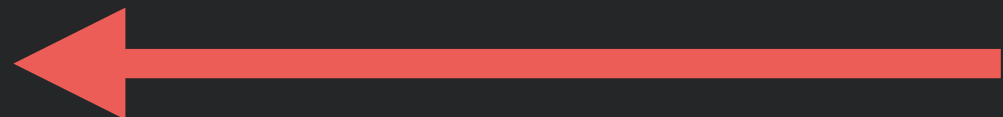


Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
};
```

```
var g = f();
g();
g();
```

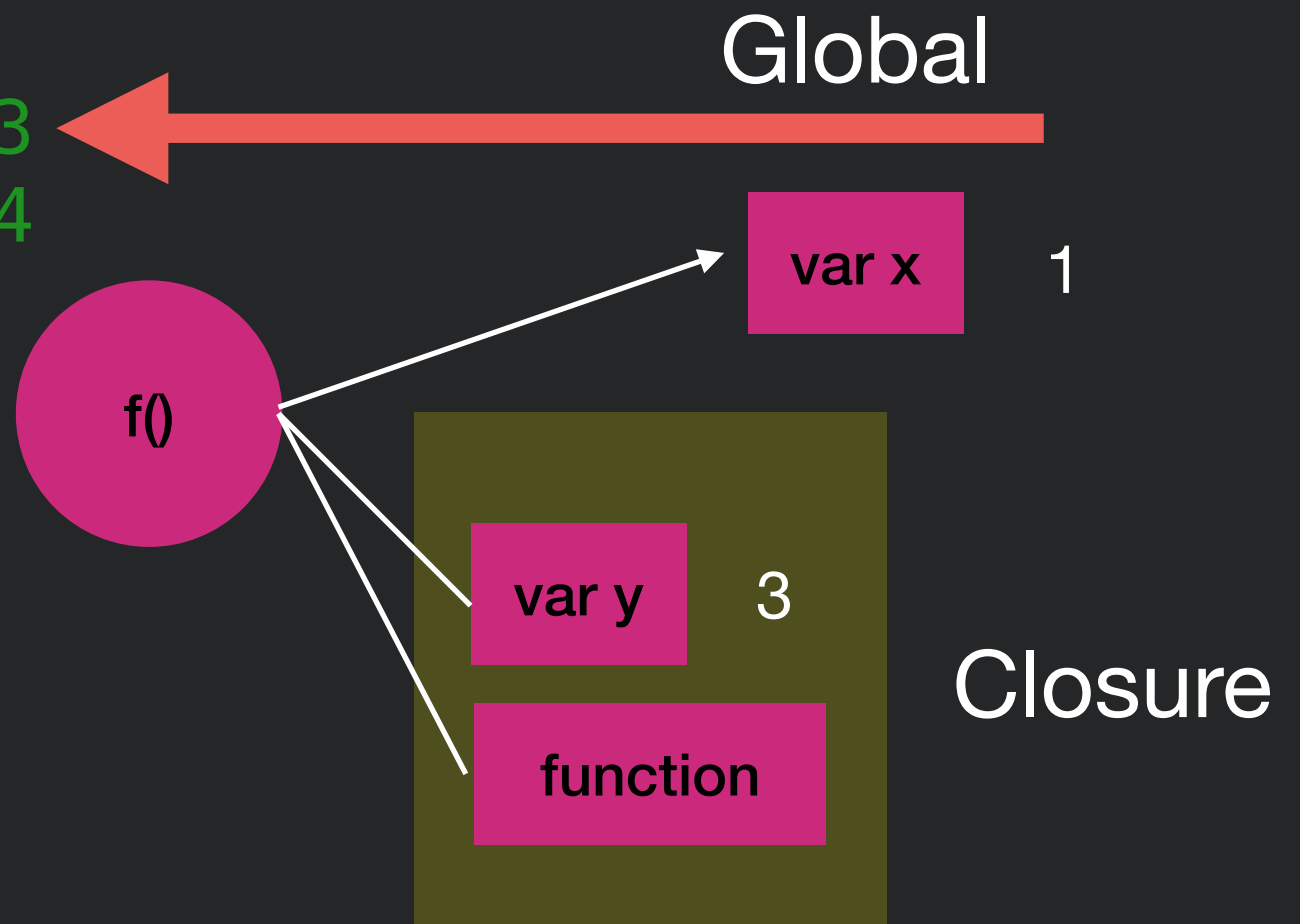
```
// 1+2 is 3
// 1+3 is 4
```



Closures

```
var x = 1;
function f() {
  var y = 2;
  return function() {
    console.log(x + y);
    y++;
  };
};
```

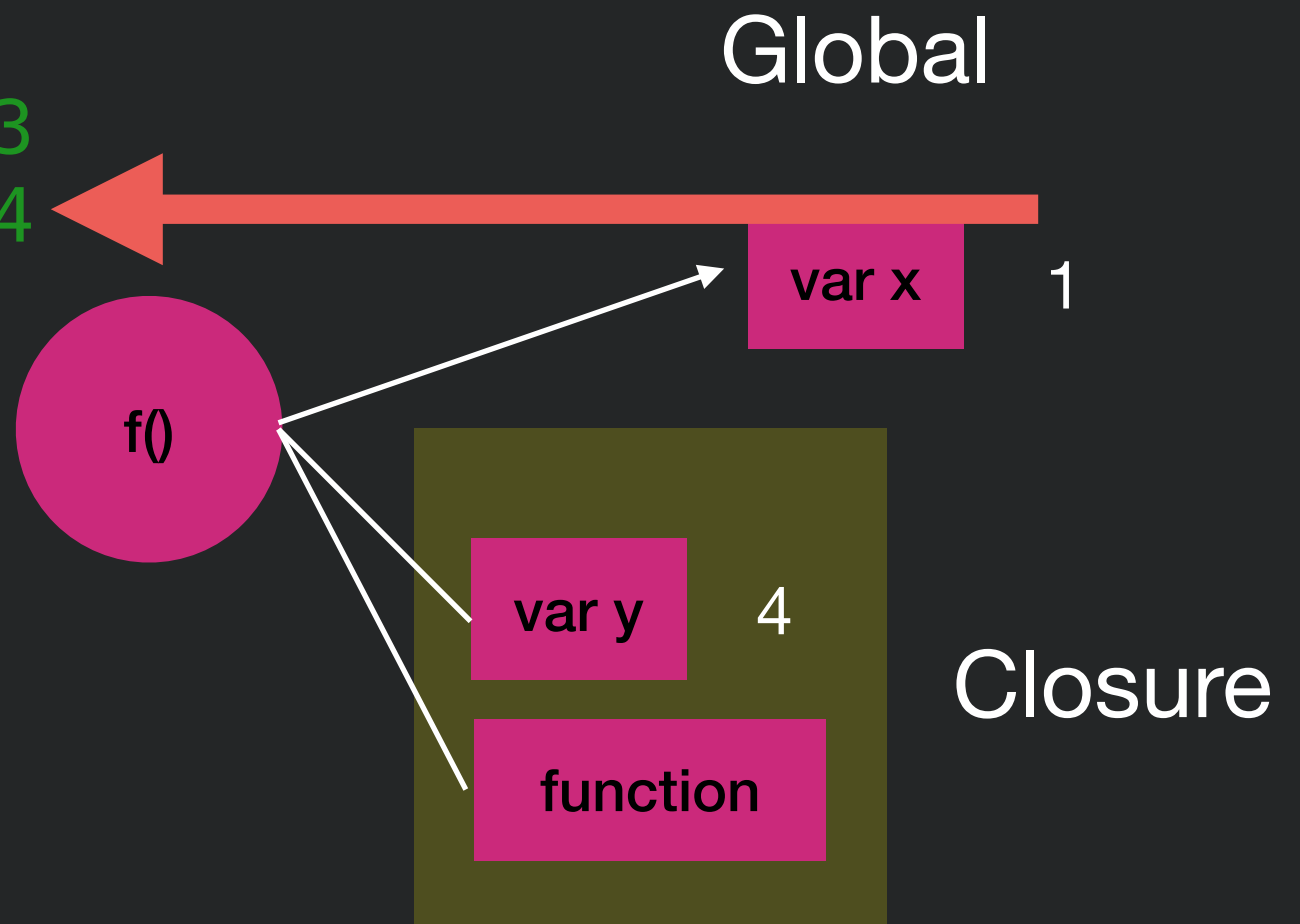
```
var g = f();
g(); // 1+2 is 3
g(); // 1+3 is 4
```



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 Johnson", section: 2}, {name:"Prof
Moran", 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 result = [];  
for (var i = 0; i < 5; i++) {  
  result[i] = function() {  
    console.log(i);  
  };  
}
```

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

```
result[0](); // 5, expected 0  
result[1](); // 5, expected 1  
result[2](); // 5, expected 2  
result[3](); // 5, expected 3  
result[4](); // 5, expected 4
```

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++) {
    result[i] = makeFunction(i);
}

```

Why does it work?

```

result[0](); // 0, expected 0
result[1](); // 1, expected 1
result[2](); // 2, expected 2
result[3](); // 3, expected 3
result[4](); // 4, expected 4

```

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

Shortcut syntax:

```

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

```



In Class Exercise: Closures & JEST

- Modify our `FacultyAPI` closure with the capability of adding a new faculty member, and then use `getFaculty` to view their formatted name.

<https://replit.com/@kmoran/SWE-432-Week-2-Closure-Exercise#script.js>



Exercise: Closures

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

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

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

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



Acknowledgements

Slides adapted from Dr. Thomas LaToza's
SWE 632 course