# Lesson 12

## Functions and Scope Revisited: Sync vs. Async Programming, Callbacks, Scope Chains, and Closures

# Lesson Objectives

- Understand the difference between synchronous and asynchronous programming
- Write functions that take other functions as arguments
- Write functions that return functions
- Understand scope chains
- Understand what a callback does
- Understand closures and what they can do for us

# Going over the homework

# Starting out with all the exercises today:

- download http://bit.ly/jsdev2-lesson12-in-class
- unzip it
- move it to your GA-JS folder

# Some functions that take functions as arguments

```
$('.click-me').on('click', function() {
  console.log("I got clicked");
});

['apples', 'oranges'].forEach(function(fruit) {
  console.log(fruit);
});
```

1. What do these three-line sections of code have in common?

2. What will be console.logged first? Why?

3. What is one fundamental difference between those two sections above?

# Sync vs. Async

- The **.forEach()** function is taking in a function and running it immediately; that function runs "**synchronously**".

- The jQuery **.on()** function is taking a function and setting it aside to be run it later, that function runs "**asynchronously**".

- While the asynchronous code is set aside to be run later, the rest of the code in the file executes normally.

- Although one's meant to be run right away and one's meant to be run in the future, the functions that get passed into **.on()** and **.forEach()** are both called "callbacks."

- And they both operate by the same basic rules of functions and scope, which we'll go over in great detail today.

# Functions and Scope again

- Today we'll be diving back into what we covered in Lessons 4 and 5: functions and scope

- We'll look at:
  - anonymous functions,
  - functions as first-class values,
  - callbacks,
  - closures (not as scary as you might have heard)

# Where have we seen functions passed to functions so far?

- **.forEach()**
- jQuery's **.on()**
- What else?

# Anonymous functions review

- Here's another useful one: **setTimeout** (time argument is in milliseconds):

```
setTimeout(function(){
  console.log("Hello world");
}, 1000);
```

- We notice 2 important things:

1. The **setTimeout** function takes a function as one of its arguments.

2. The function we are passing into the on() method does not have a name; we are just passing a raw function, consisting of:

   - the function keyword,

   - a parameter list, and

   - a function body in curly braces.

# Anonymous and named functions interchangeable

- The function being passed into **setTimeout()** on the previous slide is called an **anonymous function expression** because it is just a function literal, not named in any way

- Note that you don't have to pass anonymous functions in cases like this -- you can pass named functions as well, whether they got their names through function declaration or being assigned to a variable:

```
function sayHi() {
  console.log("Hello world");
}

var sayBye = function() {
  console.log("Goodbye world");
}

setTimeout(sayHi, 1000);

setTimeout(function() {
  console.log("I'm halfway done");
}, 630720000000);

setTimeout(sayBye, 1261440000000);
```

# Functions as first-class values

- Along with closures, this is JavaScript's chief superpower!
- Saying that functions are **first-class** values means they can be used in any part of the code that strings, arrays, or data of any other type can be used. We can:
    - assign functions to variables,
    - pass them as arguments to other functions,
    - return them from other functions, or
    - just run them ad-hoc without the need to assign them to anything.
- A function that takes another function as an argument, or returns a function, is called a "**higher-order function**".

# The official image of slides that say “functions are first-class values”

# Defining our own higher-order function

- The syntax for defining our own "**higher-order function**" — our own function that takes a function as an argument — is no different than regular function syntax

- In this example, the function **launchRocket** takes another function as an argument.

```
function blastOff() {
  console.log("Blasting off!");
}

function launchRocket(rocketName, callback) {
  console.log("Launching " + rocketName);
  console.log("3... 2... 1...");
  callback();
}

launchRocket("Viking", blastOff);

// => Launching Viking
// => 3... 2... 1...
// => Blasting off!
```

# Callbacks

- **blastOff**, the function being passed into **launchRocket**, is acting as a **callback.**

- A callback is a function passed as an argument to another function, intended to invoked within the body of that other function.

```
function blastOff() {
  console.log("Blasting off!");
}

function launchRocket(rocketName, callback) {
  console.log("Launching " + rocketName);
  console.log("3... 2... 1...");
  callback();
}

launchRocket("Viking", blastOff);

// => Launching Viking
// => 3... 2... 1...
// => Blasting off!
```

# Callbacks can take arguments

- Callback functions can also take arguments, even though we don't specify the need for arguments when we're passing the callback function itself as a variable:

```
function blastOff(destination) {
  console.log("Blasting off for " + destination + "!");
}

function launchRocket(rocketName, callback) {
  console.log("Launching " + rocketName);
  console.log("3... 2... 1...");
  callback("Mars");
}

launchRocket("Viking", blastOff);

// => Launching Viking
// => 3... 2... 1...
// => Blasting off for Mars!
```

# Returning functions

- Just as we can pass functions as arguments to other functions, we can also return functions from other functions:

```
function blastOff(destination) {
  console.log("Blasting off for " + destination + "!");
}

function makeRocketLauncher(rocketName, callback) {
  return function() {
    console.log("Launching " + rocketName);
    console.log("3... 2... 1...");
    callback("Mars");
  };
}

var launchViking = makeRocketLauncher("Viking", blastOff);
var launchMariner = makeRocketLauncher("Mariner", blastOff);

launchViking();
launchMariner();

// => Launching Viking
// => 3... 2... 1...
// => Blasting off for Mars!

// => Launching Mariner
// => 3... 2... 1...
// => Blasting off for Mars!
```

# Quiz

- What's a function that takes another function as an argument called?

1. A first-class function
2. A higher-order function
3. A callback

# Codealong: rockets

- cd ~/GA-JS/lesson12/rockets-codealong

- subl .

# Practice on your own (15 minutes)

- Change the code in the last codealong so that **makeRocketLauncher** takes both a spacecraft name and a destination. The result should be the same as before, if you pass in “Mars”. Have the code at the bottom say

```
var launchViking = makeRocketLauncher("Viking", "Mars", blastOff);
var launchGalileo = makeRocketLauncher("Galileo", "Jupiter", blastOff);
var launchCassini = makeRocketLauncher("Cassini", "Saturn", blastOff);

launchViking();
launchGalileo();
launchCassini();
```

- Which parts of the code that defines the functions will have to change?

# Independent Practice: Functions as Data (30 min)

- Find full instructions and code in the functions-as-data-exercise folder

- Write a function, **makeCountingFunction()**, that returns another function. The function returned by **makeCountingFunction()** should take an array as an argument, and return the number of odd integers in the array.

- **makeCountingFunction()** itself should take as its only argument something called a "predicate function", a function designed to run a test on each item in an array.

# Intro to Closures

- Notice how when you run **launchViking()** and **launchMariner()** in the previous section, you somehow have access to the original **rocketName**s, even though the function **makeRocketLauncher()** that you passed them into has run its course and is no longer executing?

- That's because **launchViking()** and **launchMariner()** are "closures", meaning they have "closed over" those **rocketName** variables.

- We'll go into what that means and why it's useful after a brief review of scope.

# Review of Scope

- **Scope** means *access to variables.* The scope of a given part of the code is all the parts of the rest of the code where we have access to the variables.

- If our part of the code has access to a given variable, we call that variable *in scope*.

- As we learned, scope is determined in JavaScript by what *function* a variable is defined in.

- Any global variable declared *outside* of any function (like **myGlobal**) is in scope everywhere

- Any variable declared *inside* a function (like **myLocal**) is in scope anywhere in the code of that function, but nowhere else:

```
var myGlobal = "global";

function foo() {
  var myLocal = "local";

  console.log(myGlobal);
  console.log(myLocal);
}

foo();
// => global
// => local

console.log(myGlobal);
// => global

console.log(myLocal);
// => Reference error: myLocal is not defined
```

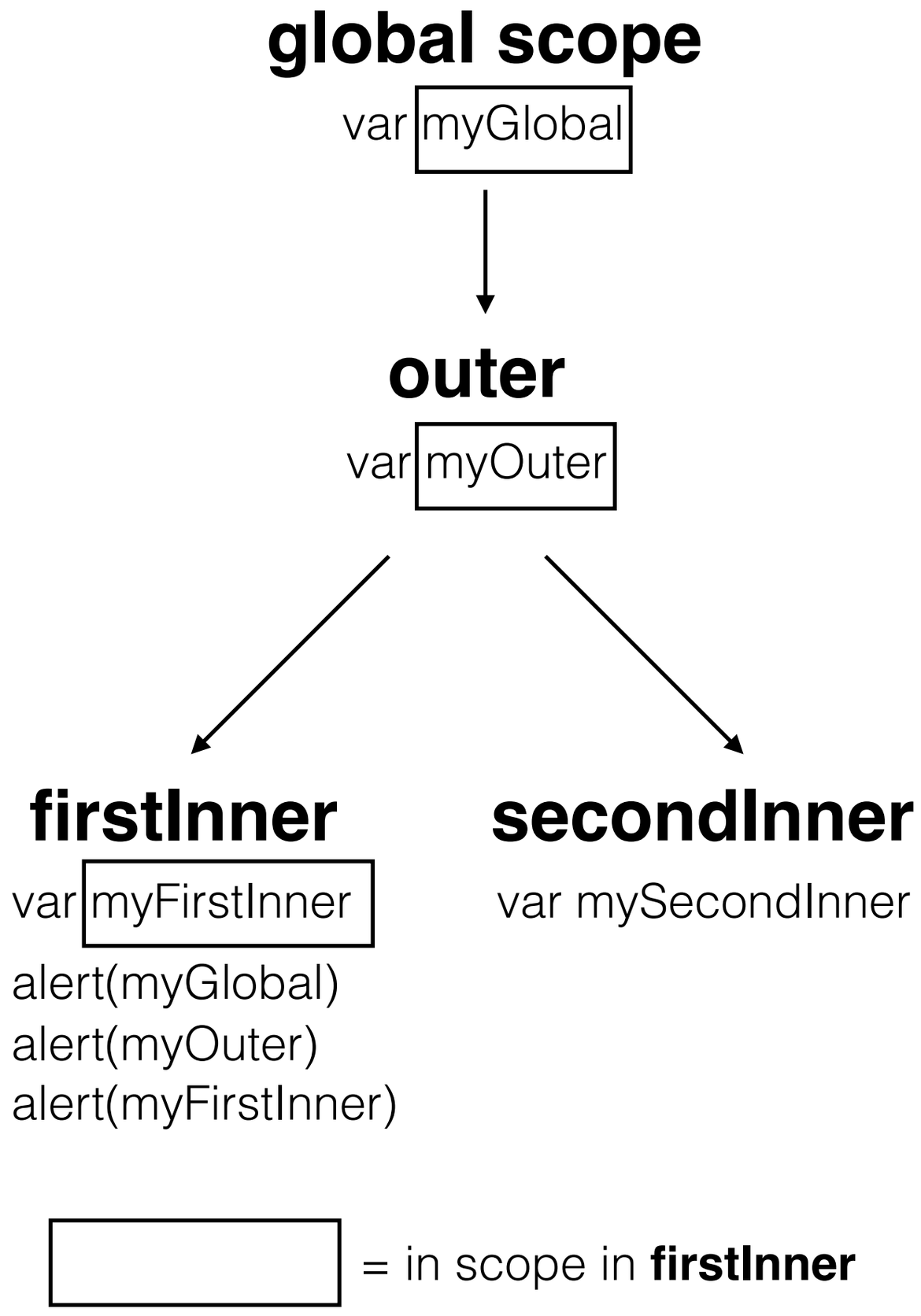
# Nested Scope

- (global scope not shown, but it's there)
- Variables declared in **outer** are in-scope in both of the **inner**s.
- Variables declared in the **inner**s are *not* in-scope in **outer**, *or in each other*.

```
function outer() {
  var myOuter = "outerVar";

  function firstInner() {
    var myFirstInner = "firstInnerVar";
    console.log(myOuter);
    console.log(myFirstInner);
  }

  function secondInner() {
    var mySecondInner = "secondInnerVar";
    console.log(myOuter);
    console.log(mySecondInner);
    console.log(myFirstInner);
  }

  firstInner();
  secondInner();

  console.log(myOuter);
  console.log(myFirstInner);
  console.log(mySecondInner);
}

outer();

// => outerVar
// => firstInnerVar

// => outerVar
// => secondInnerVar
// => Reference error: myFirstInner is not defined.

// => outerVar
// => Reference error: myFirstInner is not defined.
// => Reference error: mySecondInner is not defined.
```

# Scope tree

- It's a little bit like the DOM tree
- Inside the code of any function, all the variables that have been defined in its parent functions are *in scope.*
- For instance, inside **firstInner**, the variables myFirstInner, myOuter and myGlobal are in scope.

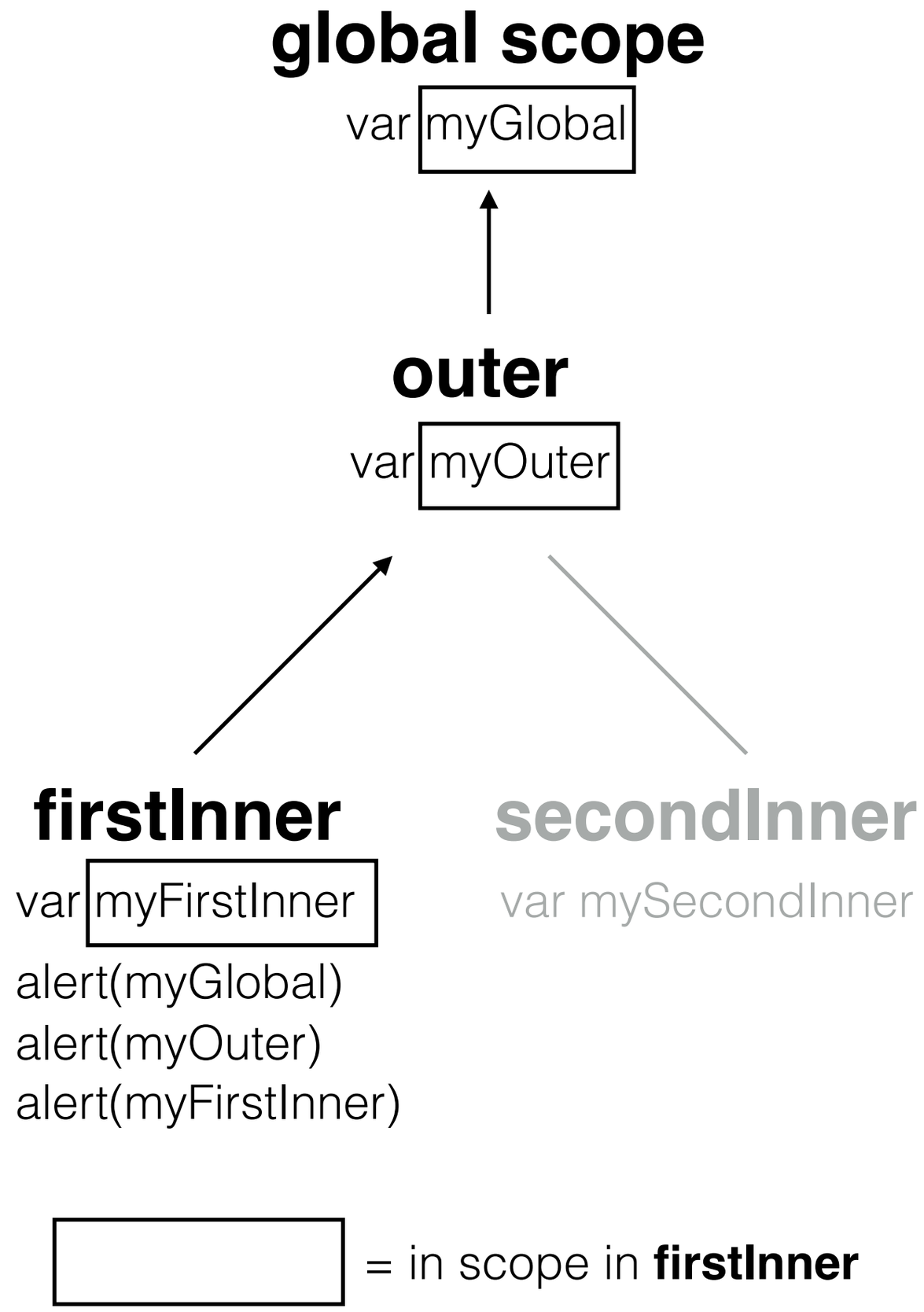


# Scope chain

- Let's look at it from the perspective of **firstInner**

- Looks like a chain going up. Parent, grandparent, etc.

- This is called a **scope chain**.



# Lexical Scope

- If you want to figure out all the variables in the scope chain of a given part of the code, all you have to do is read your code.

- The code in any function has access to all the variables that were defined *in the lines of code* in any of its parents or grandparents etc. — anywhere in its scope chain.

- That will not change at runtime. No matter where you pass a function around to, it will always have access to the variables that were in-scope in the part of the code where the function was *defined.*

- This determination of scope by where things are in relation to each other in the lines of code is called **lexical scope**.

- If this seems esoteric or confusing, don't worry — it will become much clearer next week when we go over the **this** variable, which does NOT use lexical scope, so you can see the difference.

# Quiz

- What variables are in scope inside the **baz** function?

1. **y** and **z**
2. **w, x, y** and **z**
3. **w, x** and **z**

```
var w = 5;

function foo() {
  var x = 7;
  function bar() {
    var y = 9;
  }
  function baz() {
    var z = 11;
    // ??? what variables do
    // I have access to here?
  }
}
```

# Codealong: scope chain

- cd ~/GA-JS/lesson12/scope-chain-codealong

- subl .

# Closures

- Not scary!
- A **closure** is a combination of two things:

    1. a function, and
    2. references to all the variables it had access to when it was defined.

- In other words, we just use the word “closure” to refer to a function plus the variables in its scope chain.
- We say that the function has “closed over” the variables.

# Closures

- So what does that mean, and why is it important?
- It wouldn't be very interesting at all if functions weren't first-class values — if you couldn't pass them around as data.
- But because you can, it means you can end up with functions that have access to variables originally local to other functions, which have now finished executing

# Closure example

```
<button class="show">Show number</button>
```

```
function setClickListener() {
  var a = 1;
  $('.show').on('click', function() {
    alert(a);
  });
}
setClickListener();
```

- The anonymous event handler function “closes over” **setClickListener**’s variable **a,** to form a closure.

- This gives that anonymous function something we call *state* — meaning a reference to a variable **a** — even though the click handler didn’t actually declare any variables of its own.

- After we run **setClickListener**, it’s over and done with, but **a** still exists in memory, ready to be alerted if the user clicks.

# Closure example

```
<button class="show">Show number</button>
```

```
function setClickListener() {
  var a = 1;
  $('.show').on('click', function() {
    alert(a);
  });
}
setClickListener();
```

- This bears reiterating: **Closures have access to the outer function's variable even after the outer function has returned and might not ever be run again!**

- Even though **setClickListener** has returned, our inner function is still able to call upon the variables declared by the outer function!

# Closure example

```
<button class="show">Show number</button>
```

```
function setClickListener() {
  var a = 1;
  $('.show').on('click', function() {
    a = a + 1;
    alert(a);
  });
}
setClickListener();
```

- Now we have added not just showing but mutating (adding one each time).
- What will happen when we click? What if we click again? And again?
- Even though **setClickListener** only ran once and will never run again, **a** is still available to us because it has been “closed over” in a closure.

# How closures work

- How does this happen? Answer: it's not magic

- When our inner function was defined, it closed over the variable **a**, which just means it was it was given a reference to **a**.

- Because of this reference, as long as the inner function might still be run (which is the case here because it's been attached to a DOM element as a click listener), the value of **a** will still be stored in memory.

- Once the computer detects that the function which first defined **a**, and all functions that have access to **a** through closures, will never run again, *then and only then* is **a** garbage-collected, meaning removed from memory.

# Many closures, one variable

- Important note: like any variable, the variables that a closure has closed over may also be mutated by any other part of the code where they're in scope.

- In this example we have two click listeners, one to show the number and one to increase it

- The “increase” listener affects the number that the “show” listener shows, because they both have access to *the same variable* ***a*** *in memory.*

```
<button class="show">Show number</button>
<button class="increase">Increase number</button>
```

```
function setClickListeners() {
  var a = 1;

  $('.show').on('click', function() {
    alert(a);
  });

  $('.increase').on('click', function() {
    a = a + 1;
  });
}
setClickListeners();
```

# Careful of mutation!

- The ability of different closures to mutate the same variables can easily be as harmful as it is helpful.

- Different parts of your code changing the same variables can be a recipe for disaster. (We'll see an example of this in the homework).

- So watch out! Don't change the values of variables unless you're doing it for a good reason!

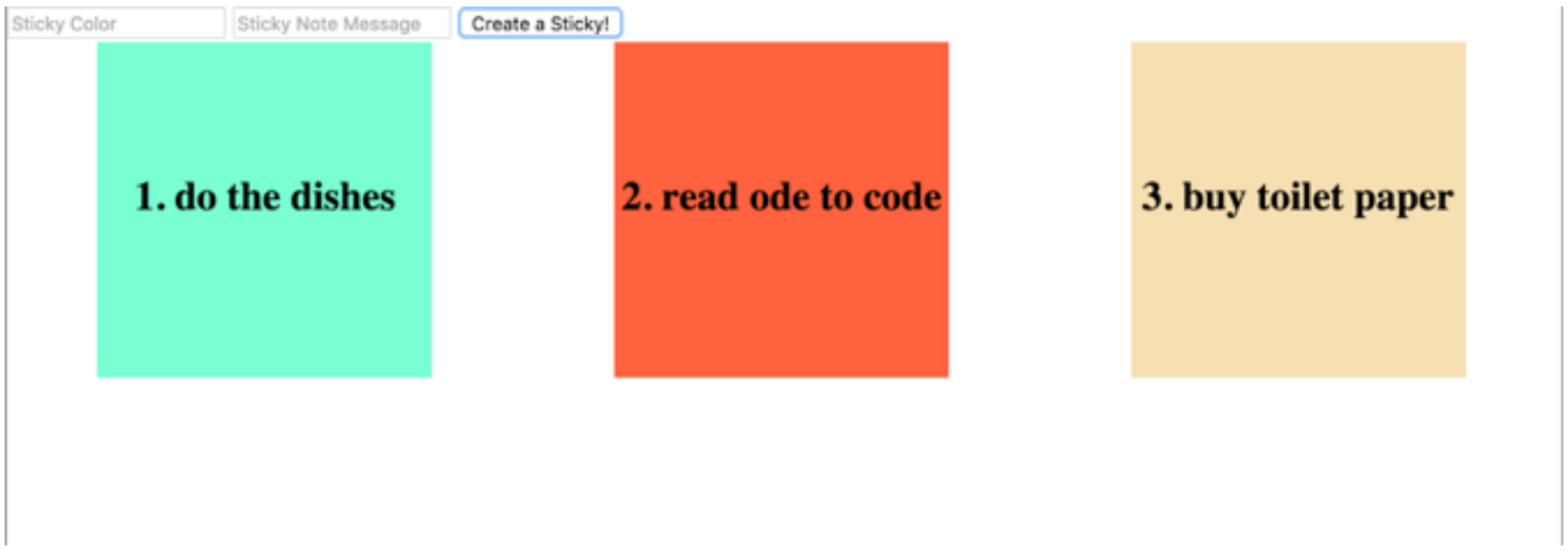
# Codealong

- cd ~/GA-JS/lesson12/clojures-codealong

- subl .

# Independent Practice: Create a Closure (20 min)

- Code in the sticky-note-exercise folder
- Create sticky notes dynamically on a page.
- CSS and most of the HTML all set up
- Sticky note color and message should come from user input
- Each note should be numbered (this is where you'll need the closure)



# Rocket Review

- Go back to the final version of the rocket exercise from earlier in the class.

- Identify the scope chains and the closures.

# Private data

- Another use of closures: they help us create private data
- Here **kind** and **wheelCount** are variables local to the factory function.
- Once the factory function runs, they can *never be accessed directly.*
- But the **car.startEngine()** method can get at them through a closure, to print them out.

```
function carFactory(kind) {
  var wheelCount = 4;
  var start = function() {
    console.log('started the ' +
                wheelCount + ' wheel ' +
                kind + '.');
  };

  return {
      startEngine: start
  };
}

var car = carFactory('Delorean');

car.startEngine();
// => started the 4 wheel Delorean.
```

# The flexibility and power of closures

- Unlike most other object-oriented languages, Javascript doesn't have a built-in concept of private variables.

- However, closures allow us to abstract away private variables and functions so that the public doesn't need to worry about the complicated details of how the program works

- In the previous example, we return an object literal which has one method, that is the only way to access the private variables

- There are many, many other examples of ways that closures — along with functions as first-class values — provide us with the flexibility and power to do most of what can be done in any other programming language.

# Further resources

- Guided tutorial on scope chains and closures (by a teacher of this class in Sydney!)
- Functions are first-class objects in JavaScript
- Brief intro to callbacks
- Demystifying JavaScript Closures, Callbacks and IIFEs
- More in-depth article on callbacks

# Homework

- **Part 1:** Set up an account with the API we'll be using on Wednesday. Instructions here.
  - *Part 1 needs to be done by this Wednesday.*

- **Part 2:** Functions, callbacks, scope and closures. This will be assigned tomorrow, we'll answer questions about it Wednesday, and it will be due next Monday.