Day 12 Data Structures

- Arrays in JS
- Manipulating objects
- JSON intro
- OOP principles
- Anonymous fucntions
- Scope and closures intro
- This keyword

Review

- Review Homework
- Review `null` and `undefined`

Recap

- Last class we learned about variables and flow control
- Tonight we'll be learning about Data Structures - Arrays and Objects

Arrays

- Arrays are ordered lists with methods to traverse and edit.
- Zero based index
- Dynamic length and types

Creating an Array

```
var teachers = ['Assaf', 'Shane'];
```

Addressing an Array

```
console.log(teachers[0]) //'Assaf'
```

Get Array Length

```
var a = [1,2,3]
teachers.length == 3;
```

Push and Pop (like a pez dispenser)

```
var teachers = ['Assaf', 'Shane'];
teachers.push('Zack'); //['Assaf', 'Shane', 'Zack']
var teacher1 = teachers.pop(); //teacher1 == 'Zack', teachers == ['Assaf', 'Shane']
```

Shift and Unshift (from the front)

```
var teachers = ['Assaf', 'Shane'];
teachers.unshift('Zack'); // ['Zack', Assaf', 'Shane']
var teacher = teachers.shift(); //teacher == 'Zack', teachers = ['Assaf', Shane']
```

Arbitrary Adding

```
teachers[4] = 'Cam Newton'; // ['Assaf', 'Shane', 'Zack', undefined, 'CamNewton'];
```

Finding an item

```
var a = [10,11,20];
a.indexOf(11); //1
a.indexOf(50); //-1
```

Slicing and Splicing

```
var a = [1,2,3,4];
//Slice - doesn't mutate array, slice(start,end)
a.slice(0,2); //[1,2]
//Splice - splice(start,numToRemove,...items to add) - so dumb
a.splice(1,2,'a','b'); //a is [1,'a','b',4]
```

Iterating Over an Array

```
//Iterating over Arrays using for loop and forEach
var teachers = ['Assaf', 'Shane', 'Zack']
for(var i = 0; i < teachers.length; i++) {
    console.log(teachers[i]);
}
//Uses a function, more on that next class
teachers.forEach(function(item, index) {
    console.log(item, index);
})
```

Converting Arrays to Strings

```
//Stringifying
teachers = ['Assaf', 'Shane'];
teachers.toString(); 'Assaf,Shane';
teachers.join('&'); 'Assaf&Shane';
```

Ordering Sorting

```
//Sorting
var a = [2, 1, 3]
a.sort(); //[1,2,3]
a.reverse(); //[3,2,1]
//Alternatively a.sort(mySortFunction);
```

Exercise 1

- Reference https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

You're going to the grocery store and decide to use an array to keep track of your shopping list.

1. Create an array to represent your shopping list with the following items: 'pop tarts', 'ramen noodles', 'chips', 'salsa', and 'coffee'.
2. Add 'fruit loops' to the list.
3. Update 'coffee' to 'fair trade coffee'
4. Replace 'chips' and 'salsa' with 'rice' and 'beans'
5. Create an empty array to represent your shopping cart.
6. Remove the last item from your shopping list and add it to your cart
7. Remove the first item from your shopping list and add it to the cart
8. Write a 'while' loop that takes items from your shopping list and moves them to your cart

until there are no items left on the list.
9. Sort the items in your cart alphabetically... backwards
10. Print the list of items in your shopping cart to the console as comma separated string.

Exercise 1 answer

```
// will be updated after all homework is submitted
```

Objects

An object is a set of keys and values, like a dictionary. Values can be

```
var course = {
    name: 'JavaScript Applications',
    awesome: true
}
```

Values can be primative objects, arrays, or other objects

```
var course = {
    name: 'JavaScript Applications',
    awesome: true,
    students: ['Jim', 'Katy'],
    instructor: {
        name: 'Assaf',
        company: 'Levvel'
    }
}
```

Addressing an Object

Object properties can be referenced in two ways. The more common *dot* notation, or the *bracket* notation, which is useful if you have a property name saved in a string.

```
course.name
course['name']
```

You can combine dot and bracket notation to address infinitely deeply nested values inside objects.

```
var course = {
    name: 'JavaScript Applications',
    awesome: true,
    teachers: ['Assaf', 'Shane']
}
console.log(course.teachers[0]); //Assaf
```

A more complex example:

```
var course = {
    name: 'JavaScript Applications',
    awesome: true,
    teachers: ['Assaf', 'Shane'],
    students: [
        {
            name: 'Steve',
            computer: {
                OS: 'Linux',
                type: 'laptop'
            }
        }
    ]
};
console.log(course.students[0].computer.OS);
```

Update an Object

Properties of objects can be updated after an object is created.

```
course.name = "super duper class";
```

Mutate an Object

You can also assign entirely new keys, delete existing ones.

```
course.fun = true; //add a property
delete course.name; //remove one
```

Exercise: Addressing Objects

Given the following object:

```
var course = {
    name: 'JavaScript Applications',
    awesome: true,
    teachers: ['Assaf', 'Shane'],
    students: [
        {
            name: 'Steve',
            computer: {
                OS: 'Linux',
                type: 'laptop'
            }
        },
        {
            name: 'Katy',
            computer: {
                OS: 'OSX',
                type: 'macbook'
            }
        },
        {
            name: 'Chuck',
            computer: {
                OS: 'OSX',
                type: 'macbook'
            }
        }
    ],
    preReqs : {
        skills : ['html', 'css', 'git'],
        equipment: {
            laptop: true,
            OSOptions: ['linux', 'osx']
        }
    }
};
```

Get the following values:

1. Name of the course ('JavaScript Applications')
2. Name of the second teacher ('Shane')
3. Name of the first student ('Steve')
4. Katy's computer type ('macbook')
5. The preReq equipment object
6. The second OSOption from equipment prereqs ('osx')
7. string listing the OSOptions separated by 'or' ('linux or osx')
8. An array of all the students that are using OSX.

Addressing Objects Answer

```
// will be updated after all homework is turned in
```

JSON

You've probably heard of JSON. It's a text based data format based on JavaScript object syntax. It's used to store data and exchange it between applications

An important difference between JavaScript objects and JSON is that proper JSON requires quotes around the property names. This is also valid in JS, but not required.

Proper JSON

```
{
    "name" : "JavaScript Applications",
    "awesome" : true
}
```

Value vs Reference types

In JavaScript, primative types like ints and strings are assigned by value. Objects and Arrays (which are also objects) are assigned by reference.

A *value* variable holds its value like you might expect. A *reference* variable points to an object in memory.

- Re-assigning a value type actually changes its value.
- Re-assigning a reference type makes it point to a different object in memory.
- Comparison of reference types compares the memory location, not value.

Before proceeding, take a moment to read this excellent StackOverflow post regarding primitive and reference values

Here are a few of examples to further illustrate:

```
//Value types
var x = 1;
var y = 1;
x === y; //true
var y = x;     // x == 1, y == 1
x === y;     // true
x = 2;         // x == 2, y == 1
x === y;     // false
```

```
//Reference types
var x = {name: 'Evan'}
var y = {name: 'Evan'}
x === y; //false;
var y = x;         //x and y are {name: 'Evan'}
x === y;        //true
x.name = 'Noah'
y.name; // 'Noah'
```

What do you suppose this means for Array's indexOf?

```
var matt = {name: 'matt'};
var julian = {name: 'julian'};
var students = [matt,julian];
students.indexOf(julian); //1
students.indexOf({name:'julian'}); //-1 (meaning nothing was found)
```

Exercise & Homework

###Due 7/26/2016

1. Complete the following class challenges
   - Exercise 1
   - Exercise 2
   - push the completed code to GitHub
   - use the naming convention `lesson_2_challenges_YOUR_INITIALS_HERE`
2. Create an object that models the data of your favorite email application.
   - Open the email application and take a look at the interface.
   - What information do you see? Make a short list (e.g. emails, my name, mailbox list, an email preview...)
   - Make a detailed outline of the data hierarchy. E.g -
   - Gmail
     - mailboxes
     - inbox
     - starred
     - sent
     - Chat Contacts
     - Shane
     - Eric
     - Katy
     - Emails
   - For each bullet in your outline, decide if it is a primative, array, or object.

- Use this information to create an object literal that models the application's data. E.g. -

  ```
  var appData = {
     name: 'Gmail',
     mailboxes: [
         'inbox',
         'starred',
         'sent',
     ],
     contacts: [
         {name: 'Shane', lastMessage: "I wont be in class today"},
         {name: 'Katy', lastMessage: "You're such a nerd"}
     ]
     //...
  }
  ```

   Add as much detail as you'd like. Experiment and have fun with it. Nest objects inside of arrays and arrays inside of objects multiple levels deep. Ask yourself if some of the primatives you've created could be objects instead.

3. Once you've composed your object, write some code to address it.
   - Get a list of inbox names
   - Get a list of emails
   - Get the text of the second email in the visible list
   - Mark an email as sent
   - Add a draft email to the drafts mailbox
   - etc. etc. etc.
4. push the completed code to our GitHub channel
   - use the naming convention `email_app_YOUR_INITIALS_HERE`

Reading

Array Reference https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Javascript types https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types

JSON tutorial http://www.w3schools.com/json/default.asp

Lesson 3 - Functions Part 1

Recap & Intro

- Last week we talked about arrays and objects, which hold structured data.
- What can we do with that data? This is where functions come in.
- Today we'll talk about functions, parameters, scope, closures
- JavaScript is a functional language, so it's all about the functions!
- Functions can be difficult for people that are new to JavaScript, because they work differently than in other languages like Ruby, Java or C#.

Function Basics

- Functions are procedures that take arguments and return values.
- Think about them like a sheet of paper with instructions on it.

Define a function

- A function has a name, an argument list, and a body.
- Arguments can be named anything you want

```
//Define a function
function saySomething(something) {
    console.log(something);
}
```

Running a function

Execute a function by calling it's name with () and passing in arguments.

```
saySomething('Hello function!'); //logs 'hello function!'
```

Return values

Functions can 'return' a value. A function evaluates to its `return` value when run.
(note to Ruby Developers: Ruby implicitly returns values. JS requires an explicit return statement if you are expecting a return value)

Wrong

```
function add(number1, number2) {
    number1 + number2;
}
var sum = add(1,2);
console.log(sum) // undefined
```

Right

```
function add(number1, number2) {
    return number1 + number2;
}
var sum = add(1,2);
console.log(sum); // 3
```

Function Arguments

All functions take any number of arguments, regardless of their declared signature.

```
function add(a,b) {
    console.log(a,b)
}
add(1); // '1,undefined'
add(1,2,3,4,5) // '1,2'
```

The arguments list simply creates variables that reference the arguments in order they were passed. For functions that take an unknown number of arguments, use the `arguments` object.

```
function add() {
    var sum = 0;
    for(var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
add(1,2,3,4,5,6,7,8);
```

Exercise 1 (function basics)

Calculating gratuity is a repetitive task, so let's create a couple of functions that do the work for us.

- create a variable titled `billAmount` and store a random number (ie: 100)
- create a function titled `gratuity()`

    - gratutity should:
    - multiply the value of billAmount by 20%
      hint: use 0.2
    - return the value
- create a function titled `totalWithGrat()`
    - totalWithGrat should:
    - take in the amount as an argument
    - call the gratutity function
    - add that to the original bill amount
    - return the total bill + gratuity
- log the total (with gratuity) to the console
    - append new total to the following phrase:
    - "your total including gratuity is:"
- Limitation: You can only invoke the totalWithGrat function when logging the result

Mariel's Bonus Question

- Find a way to fix the decimal point to only 2 places, ie: 100.00
  (hint: the answer is in the sentence above)

Exercise 1 Answer:

```
var billAmount = 100.58;
function gratutity(){
    return billAmount * 0.2;
  }
function totalWithGrat(amount){
  return gratuity() + amount;
}
console.log("your total, including gratutity is: $" +  totalWithGrat(billAmount).toF
```

Functions as Objects

- Functions are first class Objects in JavaScript.
- This means they can be:
    - instantiated
    - assigned
    - reassigned
    - and passed around just like any other variable.
- Again, think of them as a physical piece of paper.

Functions as Variables

Like other objects, functions can be assigned to variables.

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

The difference between declaring a function that way (Function assignment) and the `function add(){}` syntax we've been using (Function Declaration) is that the latter hoists both the declaration and definition. For example:

Function Declaration

```
hoisted(); // logs "foo"
function hoisted() {
  console.log("foo");
}
```

Function Assignment

```
notHoisted(); // TypeError: notHoisted is not a function
var notHoisted = function() {
   console.log("bar");
};
```

In the above example, we get a `TypeError` because `notHoisted()` is declared, but undefined *until* the assignment expression `var notHoisted =`.

Anonymous Functions

- It's often handy to declare a function on the fly without a name.
- This is VERY common

```
var calculator = {
    add: function(a,b) {
        return a + b;
    }
}
calculator.add(2,3) // 5
```

So, what is the point of an Anonymous function?

- Cleaner code
- Scope management, used to create private scope (more on that later)
- Super useful with Closures (more on that later as well)

While, this might look strange, but you've already been doing this with Arrays and Objects. Functions are no different.

```
var arrayOfMystery = [
    ['anonymous','array'],
    { name: 'anonymous object' },
    function(){ return 'Anonymous Function!'}
]
console.log(arrayOfMystery[0][1]) // array
console.log(arrayOfMystery[1].name) // anonymous object
console.log(arrayOfMystery[2]()) // anonymous function!
```

- `[ ]` creates an array in memory
- `{ }` creates an object in memory
- `function(){ }` creates an object in memory

Exercise 2 (RPS Revisited)

Let's revisit Rock Paper Scissors...

1. Define a `hands` array with the values 'rock', 'paper', and 'scissors';
2. Define a function called `getHand()` that returns a hand from the array using `parseInt(Math.random()*10)%3`
3. Define two objects for two players. Each player has `name` and `getHand()` properties.
4. Define a function called `playRound()` that
    - Takes two player objects as arguments
    - Gets hands from each
    - Determines the winner
    - Logs the hands played and name of the winner.
    - If its a tie, log the hands played and "it's a tie".
    - Returns the winner object (null if no winner)
5. Define a function called `playGame()` that takes arguments `player1`, `player2`, and `playUntil`.
    - Play rounds until one of the players wins `playUntil` hands
    - When one player has won enough games, return the winning player object
6. Play a game to 5 wins

Mariel's Bonus Questions

- Define a function caled `playTournament()`
    - Take 4 players and `playUntil` as arguments
    - Play a game between the first two players, and the second two players
    - Play a game between the winners of the first round.
    - Announce the tournament winner's name "[name] is the world champion";

Homework

###Due on 7/28/2016

- Complete the new Rock Paper Scissors Challenge
    - Push completed to GitHub with the name: `RPS_functions_YOUR_INITIALS_HERE`

###Due on 8/2/2016

- Complete the JavaScripting Module at NodeSchool
    - Upload a completed screenshot to Slack
- Read the You Don't Know JS book on closures. (chapters 1 - 5)
    - Complete the code examples
    - push to github with the name: `YDKJS_closures_YOUR_INITIALS_HERE`

Reading

- http://javascriptissexy.com/understand-javascript-callback-functions-and-use-them/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments

#Lesson 4 - Functions - Part 2 - Scopes, Closures, & this

Recap & Agenda

- Over the past 3 lessons, we've seen all the major building blocks of JS
    - variables
    - arrays
    - objects
    - the basics of functions
- Tonight we'll be covering the more advanced parts of functions
    - Scope
    - Closures
    - this

Functions (a slight review)

What are functions?

- A function is a reusable piece of code that performs a task
- A function is an object, just like objects and arrays
- A function and its return value are not the same thing.
- Defining a function requires PBR (Parameters, body, return value)

    - Parameters - What input does the function need in order to run?
    - Body - What will the function do with that information?
    - Return - What output will executing the function give back?

```
function nameOfMyFunction(list, of, parameters) {
    //BODY: Logic and operations based on parameters e.g.
    //RETURN: Value returned to the caller of the function;
    return list;
}
nameOfMyFunction(1,2,3); // 1
```

Scope

- Scope is the set of variables a piece of code has access to.
- Only functions create scope
- Parameters and variables declared inside a function are LOCAL to that function's scope
    - Variables are only visible inside that scope they are defined in and in its child scopes

So, what is a lexical scope? Lexical scoping simply means that variables refer to their local environment.

```
function add() {
    var a = "This variable is only visible inside the add function";
}
```

```
console.log(a) // Undefined
```

```
function getGreeting(name) {
    var greeting = "hello ";
    return greeting + name;
}
getGreeting('Shane'); // 'Hello Shane'
console.log(greeting) //undefined
```

Blocks DO NOT have their own scope

```
for(var i = 0; i < 10; i++) {
    var x = i;
}
console.log(i,x); //9,9
```

Scope lets you create private variables in a JS program, but be careful, variables defined without the 'var' keyword are global.

```
function greet(name) {
    greeting = "hello ";
    return greeting + name
}
greet('Shane');
greeting //'hello'!
```

Scope Chain and Closures

- When a scope is created, it is nested inside another scope.
- To resolve a variable, JS looks at the immediate scope, then parent scopes in order.
- A closure is an inner function that has access to the outer function’s variables. Both scopes together are the scope chain.

For example, the `city` variable is visible inside the `greet` function because the greet function creates a closure.

```
var city = 'Charlotte';
var greet = function() {
    console.log('Hello ' + city);
}
greet(); // Hello Charlotte
```

Accessing `city` inside the greet function will first look for a variable named `city` in the greet function, then to the parent scope, and so on up the scope chain.

Nested scope Example 1

```
function outer() {
    var x = 'x';
    function inner() {
        var y = 'y';
        console.log(x); //'x'
    }
    console.log(x); // 'x'
    console.log(y); // ReferenceError: y is not defined
}
```

Nested scope Example 2

```
var landscape = function() {
    var result = "";
    var flat = function(size) {
        for (var count = 0; count < size; count++)
            result += " _ ";
    };
    var mountain = function(size) {
        result += "/";
        for (var count = 0; count < size; count++)
            result += " '";
        result += "\\";
    };
    flat(3);
    mountain(4);
    flat(6);
    mountain(1);
    flat(1);
    return result;
};
console.log(landscape());
```

- `flat` and `mountain` are functions only available within `landscape`
- The `result` variable is available inside `flat` and `mountain` because

Variable name conflicts

Sometimes parent scopes have variables with the same name.

```
var name = 'Shane';
var greet = function() {
    var name = 'Matt'
    console.log(name);
}
console.log(name); // Shane
console.log(greet(name)); // Matt
```

Declaring the variable `name` in the inner scope hides the variable with the same name in the outer scope. This is a new variable, so assigning to the inner variable won't affect the outer.

Sometimes a function defines argument names with the same name as variables in the parent scope. This works the same way as the example above.

```
var name = 'Shane';
var greet = function(name) {
    console.log(name);
}
greet('joe'); // 'joe'
//equivalant to
var name = 'Shane';
var greet = function(mySuperUniqueVariableName) {
    var name = mySuperUniqueVariableName;
    console.log(name);
}
greet('joe'); // 'joe'
```

Exercise 1 (Scope & Closures)

Start with the following code template. After each step, run the program and see how the output changes.

```
function outer(){
  function inner() {
  }
  inner();
}
outer();
```

1. Declare two variables, `a` and `b` in the outer function's scope and set them to a string and an object respectively. Log their values immediately.
2. Log the values of `a` and `b` in the inner function.
3. Update the inner function to take two parameters named `a` and `b`.
4. Pass `a` and `b` in as arguments when you execute inner().
5. Inside the inner function, assign new values to `a` and `b` and log them at the end of the function AND after the execution of `inner(a,b)`.

6. Inside the inner function, update a property of the `b` object.

7.
    -
        -

▪

One more thing on Scope and Closures...

- A Function has access to the variables that were in scope when it was defined, *not executed*.
- Sorry, this next example is just evil...

```
//Scope, closure, and hoisting, oh my!
function createFunction() {
    var a = "Hans Zimmer Rules!";
    inception = function() {
        console.log(a);
    }
}
var inception;
createFunction();
inception(); // "Hans Zimmer Rules!"
```

This Keyword

Inside a function, the keyword `this` refers to the executor of the function.

- PAPER EXERCISE

Again, think of a function as a piece of paper with instructions, a procedure of sorts. One of those instructions might say "touch your nose". But who is this "you" it speaks of? Obviously, the person executing the instructions. Similarly, the keyword 'this' refers to the object that's executing the function.

```
var teacher = {
    name: 'Assaf',
    sayName: function() {
        console.log(this.name);
    }
}
teacher.sayName(); //'Assaf'
```

Different objects can execute the same function, can produce different results because `this` is different.

```
function sayName() {
    console.log(this.name);
}
var teacher1 = {
    name: 'Assaf',
    speak: sayName
}
var teacher2 = {
    name: 'Shane',
    speak: sayName
}
teacher1.speak(); // 'Assaf'
teacher2.speak(); // 'Shane'
```

Exercise: This

Create a single object named `slideshow` that represents the data and functionality of a picture slideshow. There should be NO VARIABLES OUTSIDE THE OBJECT. The object should have properties for:

1. An array called `photoList` that contains the names of the photos as strings
2. An integer `currentPhotoIndex` that represents which photo in the `photoList` is currently displayed
3. A `nextPhoto()` function that moves `currentPhotoIndex` to the next index `if` there is one, and:
    iv. logs the current photo name.
    v. Otherwise, log "End of slideshow";
4. A `prevPhoto()` function that does the same thing, but backwards.
5. A function `getCurrentPhoto()` that returns the current photo from the list.

Exercise Answer

```
// Have fun :)
```

Homework

###Due 8/2/16:

- Complete the Scope Chain & Closures Module at NodeSchool
    - Upload a completed screenshot to slack
- Complete the Slideshow challenge
    - Push the finished code to GitHub using the naming convetion: `slideshow_YOUR_TEAM_INTIALS_HERE`
    - Paste a link to the finished code in Slack

- Read this article on Scope and Closure

- Read the You Don't Know JS book on closures. (chapters 1 - 5)
    - Complete the code examples
    - push to github with the name: YDKJS_closures_YOUR_INITIALS_HERE
- Watch this TeamTreehouse video on scope