GENERAL ASSEMBLY

ARRAYS LOOPS & FUNCTIONS

Jonathan Cheng

REVIEW

LEARNING OBJECTIVES

- Work with Javascript variables to store information
- Execute different blocks of code based on certain conditions
- Understand the types of data Javascript understands and how we write them (syntax)
- Perform basic arithmetic with numbers using Javascript
- Use the browser’s built in Javascript console to debug and experiment

LEARNING OBJECTIVES

LEARNING OBJECTIVES

- Define arrays and practice using loops and indexes to access array elements
- Describe arguments as they relate to functions.
- Predict values returned by a given function.
- Differentiate control flow between anonymous and named functions.

ARRAYS

ARRAYS

COMPUTERS NEED A WAY TO REFERENCE MULTIPLE VARIABLES AT THE SAME TIME

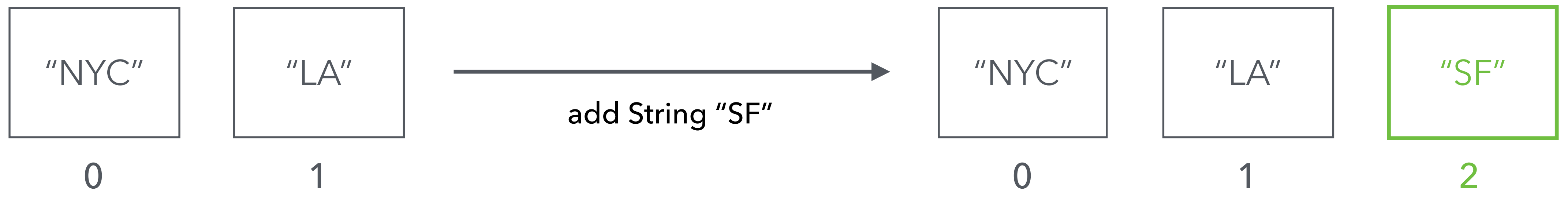
We can do this by grouping variables inside special collections called arrays

Examples:

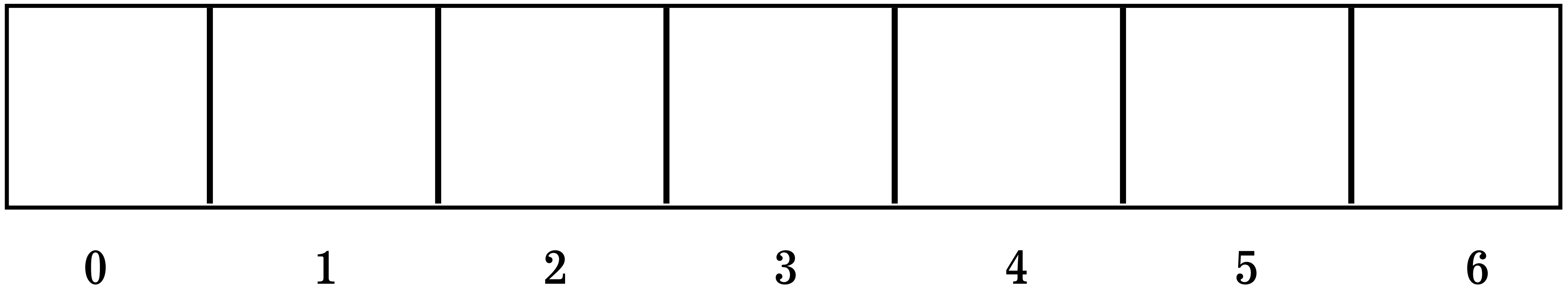
- *Grabbing all the restaurants within a 5 mi radius of your current location (yelp)*
- *Getting the 20 latest status updates of your friends (facebook)*
- *Grabbing 10 images to insert into an image carousel*

ARRAYS

- An **Array** is a data structure that can store multiple chunks of data in individual bins.
- Each bin holds one item, which is referred to as an **element** of the Array.
- Elements can be basic data types, variables, or even other arrays.
- Each element in the Array is labeled by an integer called an **index**.
- Arrays are zero-indexed, which means that the index of the first element is always **0**.
- Arrays can shrink and grow in size to accommodate removing/adding elements.



ARRAYS

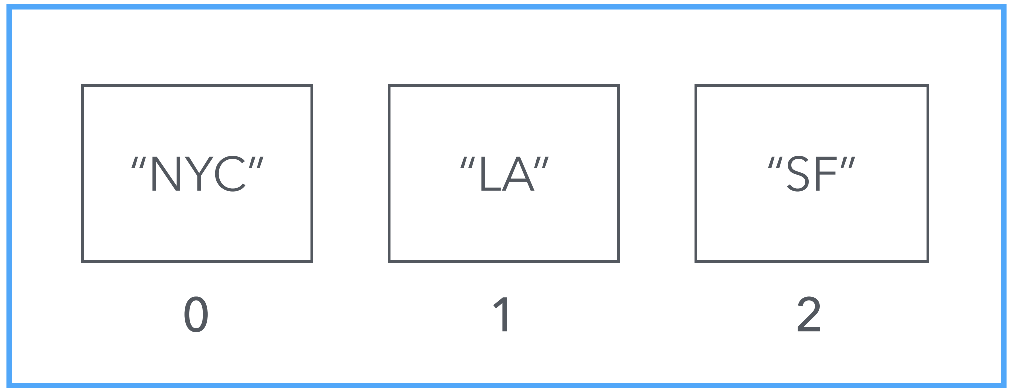


JAVASCRIPT ARRAYS

- To use an Array in Javscript, we'll create it using the [ ] syntax and separate each element with a comma
- The following code snippet creates an array of size 3 that holds three String objects.

```
var some_array = [“NYC”, “LA”, “SF”];
```

```
some_array.length;  // gives us 3
```

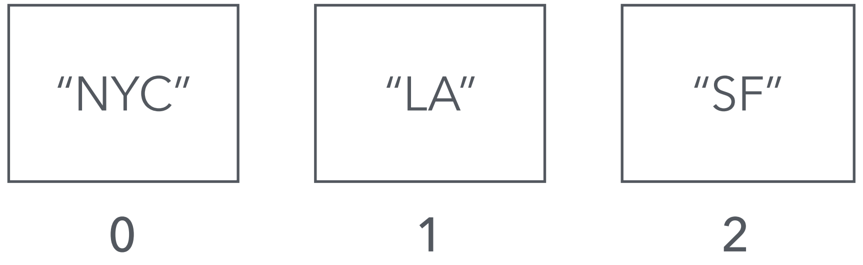


some_array (this is an Array)

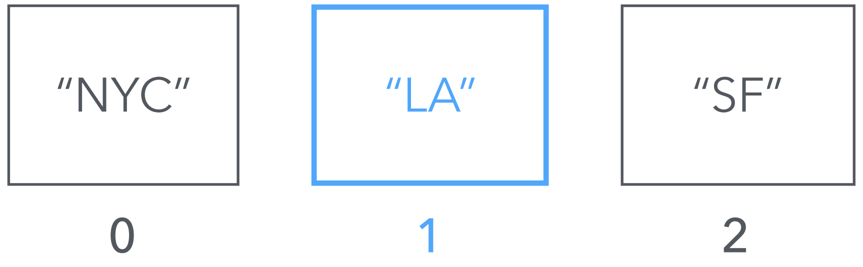
ACCESSING ARRAY ELEMENTS

- [] allows us to access the elements of a Ruby array by index (it’s a getter and a setter)

```
var some_array = [“NYC”, “LA”, “SF”];
```



```
some_array[1]
```



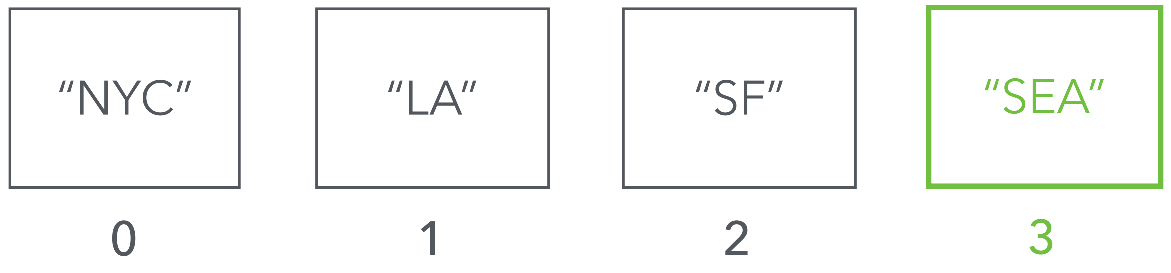
```
“LA”
```

ADDING/REMOVING ARRAY ELEMENTS

- There are several ways to add elements to and remove them from a Javascript Array

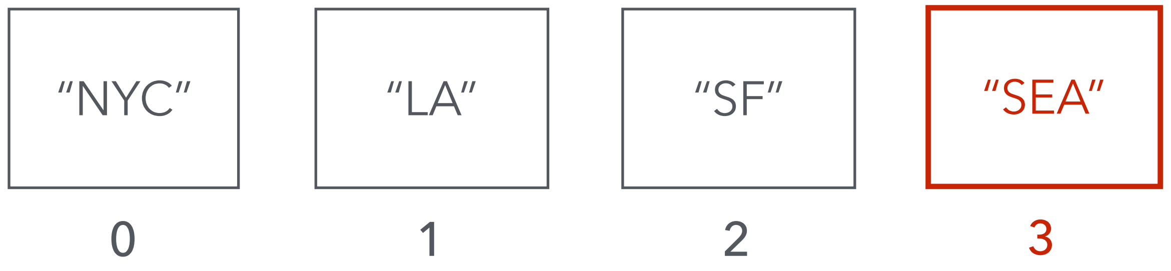
```
some_array.push("SEA") #returns the array
```

- Adds "SEA" to the end of the array
- Returns the length of the updated array



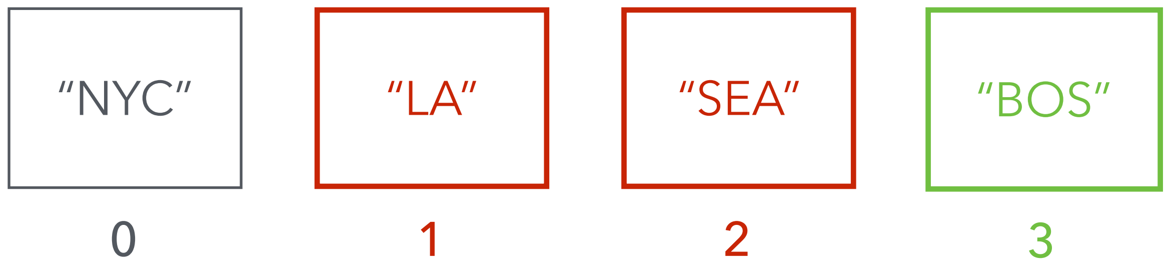
```
some_array.pop() #returns "SEA"
```

- Removes the last element from the array
- Returns the element that was removed



```
some_array.splice(1, 2, "BOS")
```

- Removes 2 elements starting at position 1
- Then adds "BOS" to position 1
- Returns the removed elements



ARRAYS

MORE ARRAY GOODIES!

There are a ton of functions built into Arrays that allow you to do all kinds of stuff. To explore more, check out MDN's array docs for examples and detailed breakdowns:

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

CODE ALONG

ARRAYS

LOOPS

FOR LOOPS

- Often times we need to access all of the elements in an array one by one
- There are several ways to do this in Javascript, but the most common is using a **for** loop
- A **for** loop steps through each element in an array and makes it available to us within the body of the loop via a variable that we name in the loop declaration

```
var cars = [“honda”, “toyota”, “ford”];

for ( var i = 0; i < cars.length; i++){
  console.log(cars[i]);
  //note: we can now access each element in the array
  //using cars[i]
}
```

FOR LOOPS EXPLAINED

For loops are broken into three parts, a setup step, a condition that is checked before each time it is run, and an ending step that runs after each time.

Our goal is to use the array indexes and count from 0 to array.length - 1 in order to access every element in the cars array.

```
for ( var i = 0; i < cars.length; i++){
  console.log(cars[i]);
  //note: we can now access each element in the array
  //using cars[i]
}
```

FOR LOOPS EXPLAINED

First, we'll need to setup a variable as our counter so we can call cars[0], cars[1], cars[2] … cars[cars.length-1]. We can start the variable at 0, then add 1 to it each time the loop runs. To set this up, we can use the setup step to create a new variable i, and we can use the ending setup to increment i by 1 each time.

In this case, we'll use the handy incrementer ++

```
for ( var i = 0; i < cars.length; i++){
  console.log(cars[i]);
  //note: we can now access each element in the array
  //using cars[i]
}
```

FOR LOOPS EXPLAINED

The next step is establishing the conditions under which the loop runs. In this case, we'd like the loop to run over each array index, and stop at the last one, so our counter shouldn't be bigger than the array's length. We'll make sure to check that each time before the loop is run, and if the condition is false, we'll stop running the loop.

```
for ( var i = 0; i < cars.length; i++){
  console.log(cars[i]);
  //note: we can now access each element in the array
  //using cars[i]
}
```

FOR LOOPS EXPLAINED

Lastly, we'll want to decide what we'd like to do for each element in the array. We can do this by putting code in between the curly braces which will be run until the loop's condition is false.

In this case, we'll print each element out to the console using console.log. We do this by using the array accessing syntax we saw before: cars[0], cars[1], cars[2]. Only this time, we've got an int stored in the variable **i**. Since **i** gets bigger by 1 each time the loop is run, we can access all of the elements in the cars array by using it in our accessor.

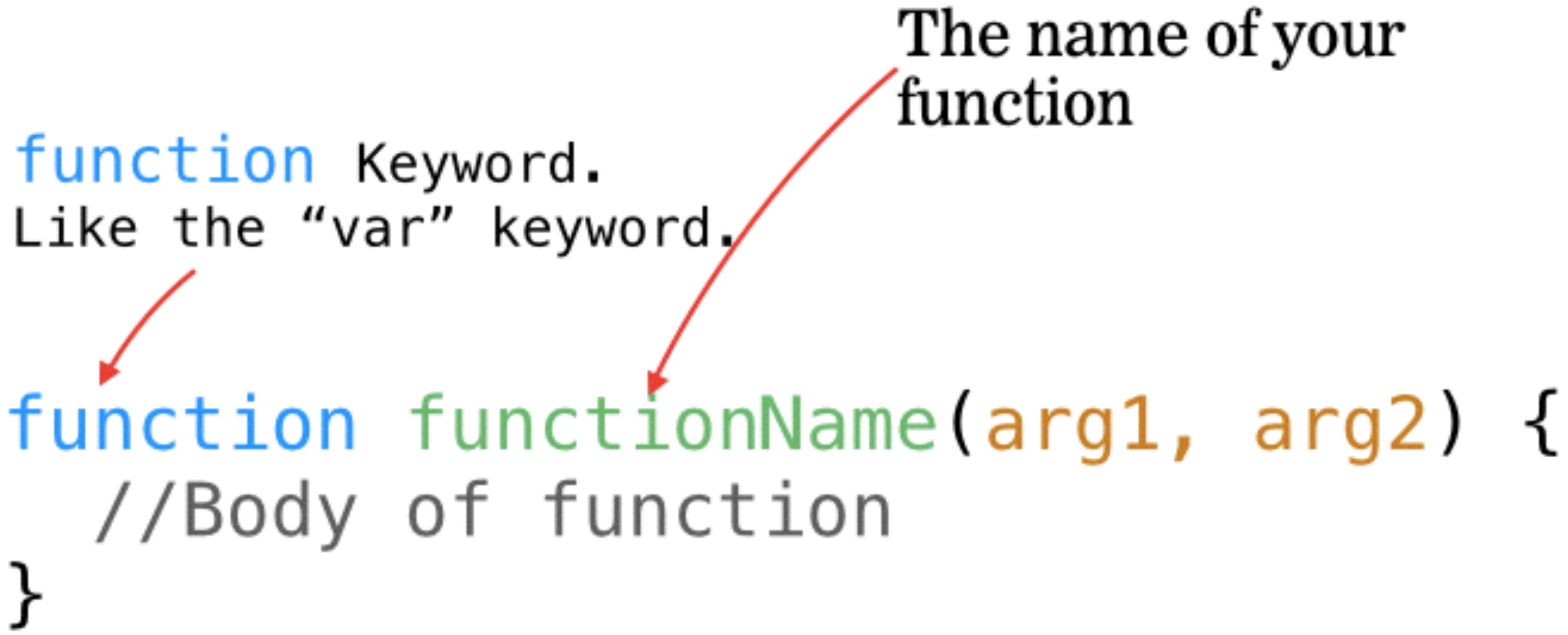
```
for ( var i = 0; i < cars.length; i++){
  console.log(cars[i]);
}
```

CODE ALONG

LOOPS

FUNCTIONS

FUNCTIONS



```
function functionName(arg1, arg2) {
  //Body of function
}
```

FUNCTIONS

Arguments let you pass data into the function

```
function functionName(arg1, arg2) {
   //Body of function
}
```

The functions executed code goes between the { } brackets. Much like an “if” statement.

FUNCTIONS

```
function helloWorld() {
  console.log("Hello Functions");
}

helloWorld(); //Prints “Hello Functions to the
console.
```

The brackets execute the function. Try calling the function without them to see what happens.

FUNCTIONS

```
function addAndPrint(num1, num2) {
  var sum = num1 + num2;
  console.log(sum);
}

addAndPrint(1, 2); // Result is 3

addAndPrint(8, 2); // Result is 10
```

RETURNING DATA

FUNCTIONS

RETURNING VALUES FROM FUNCTIONS

Often times, once we finish using a function, we'd like to do more work with the end result of the function or store it for use later. With what we know now, here's what we could do:

```
var product;
function square(x) {
  product = x * x;
}
```

The problem here is we've only ever got one variable to work with since this function can only ever update the product variable.

FUNCTIONS

We'd like the square function to square any numbers and store the result to any variable. Here's how we can do that:

```
function square(x) {
  return x * x;
}

var product = square(2);
//product is 4

var product2 = square(3);
//product2 is 9
```

ANONYMOUS FUNCTIONS

FUNCTIONS

Functions don't require a name in order to be created. We can create an anonymous function (a function without a name) and set it to a variable to be used later. A function like this will work identically to the first function, with a few key differences. Here we've created an anonymous function and set it to a variable called square.

```
var square = function (x) {
  return x * x;
}
```

FUNCTIONS

When setting an anonymous function to a variable (function expression), calling the function before it is defined won't work. If we create a function in the normal sense (function declaration), calling it before the declaration works fine.

✗

```
square(4);

var square = function (x) {
  return x * x;
}
```

✓

```
square(4);

function square(x) {
  return x * x;
}
```

FUNCTIONS

When you start using more advanced javascript libraries, you'll probably see something that looks like this:

```
function addImage(callback) {
   //this function adds an image to the page
}
```

FUNCTIONS

In the parameter named "callback" you can insert an anonymous function to execute after the initial function has completed.

```
function addImage(callback) {
  //this function adds a hidden image to the page
}
```

We call the addImage function to add a hidden image to the page with a callback, an anonymous function that fades the new image in.

```
addImage(function(){
  //an anonymous function that reveals the image with a nice fade in
});
```

ANONYMOUS FUNCTIONS

If anonymous functions sound a little crazy, relax!

CODE ALONG

FUNCTIONS

LAB
IMAGE CAROUSEL

FINAL PROJECT

WRAP UP

LEARNING OBJECTIVES

- Define arrays and practice using loops and indexes to access array elements
- Describe arguments as they relate to functions.
- Predict values returned by a given function.
- Differentiate control flow between anonymous and named functions.