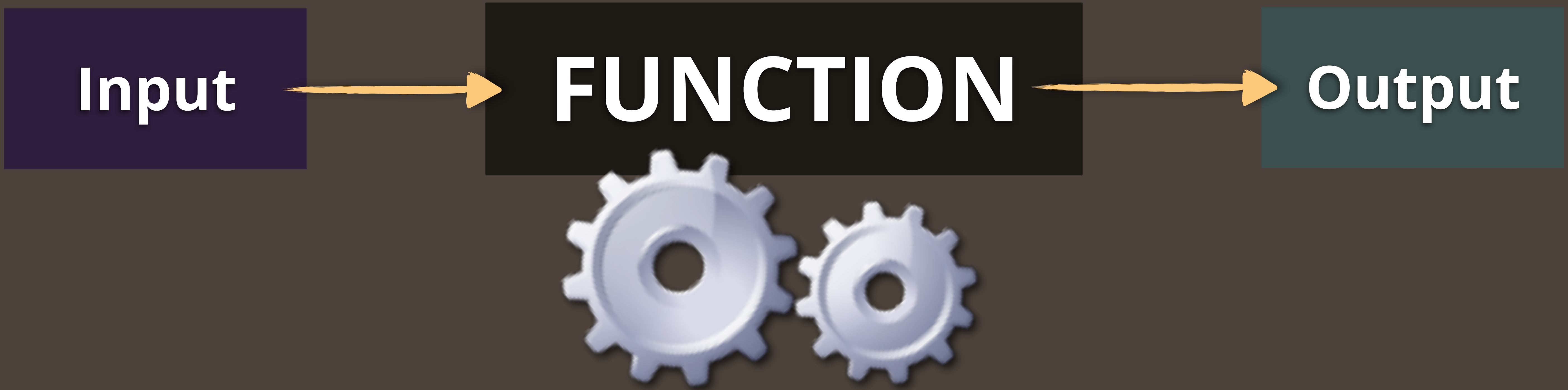
LEVEL 4

THE DESERT OF DECLARATIONS

WHAT'S A FUNCTION FOR?



Give the function some input…

…it does some stuff to or with the input…

…and it outputs some result.

FUNCTIONS SOLVE PROBLEMS

A function "does something" step-by-step that we need to do repeatedly

FUNCTION: The Sum of Two Cubes

1. Get two numbers 4 9
2. Cube each number $4^3 = 64$ $9^3 = 729$
3. Sum the cubes $64 + 729 = 793$
4. Return the answer $\boxed{793}$

WHAT ARE THESE STEPS IN CODE?

Syntax for finding a sum of cubes

$$4$$

$$9$$

$$4^3 = 64$$

$$9^3 = 729$$

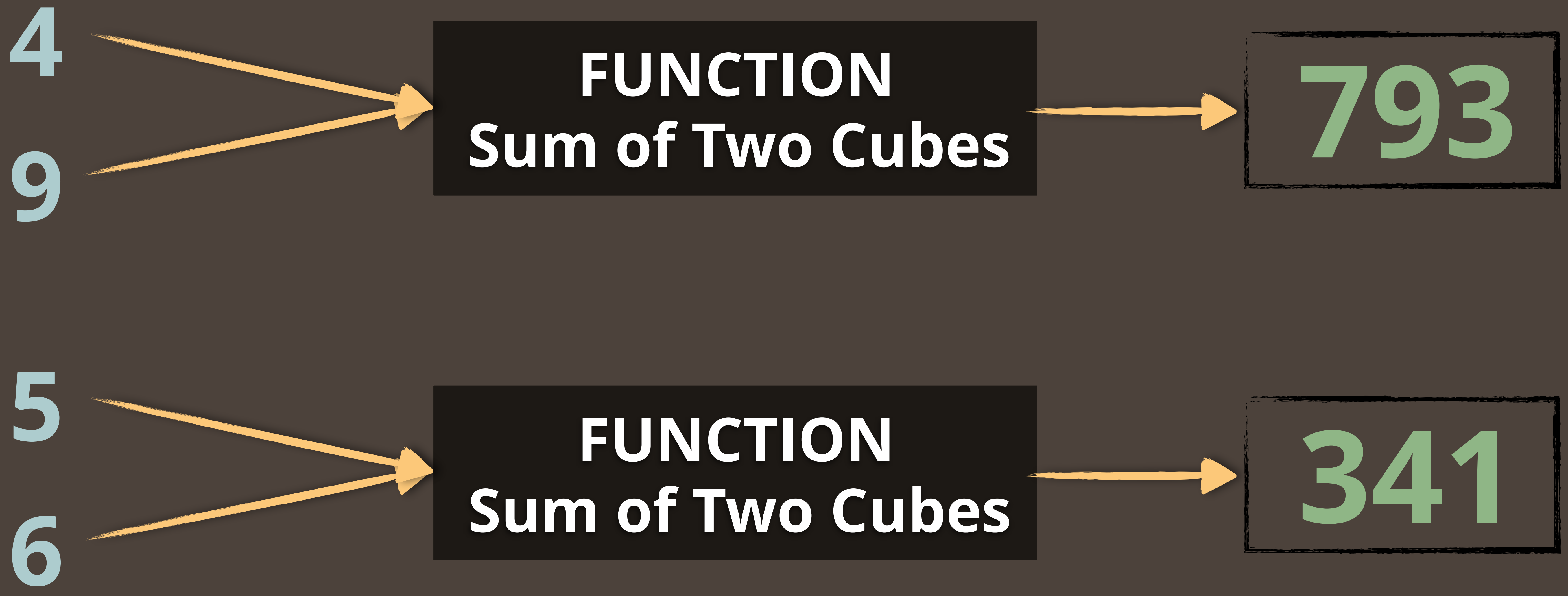
$$64 + 729 = 793$$

```
var a = 4;
var b = 9;
var aCubed = a*a*a;
var bCubed = b*b*b;
var sum = aCubed + bCubed;
```

Without a function, we'd have to write this code a lot!

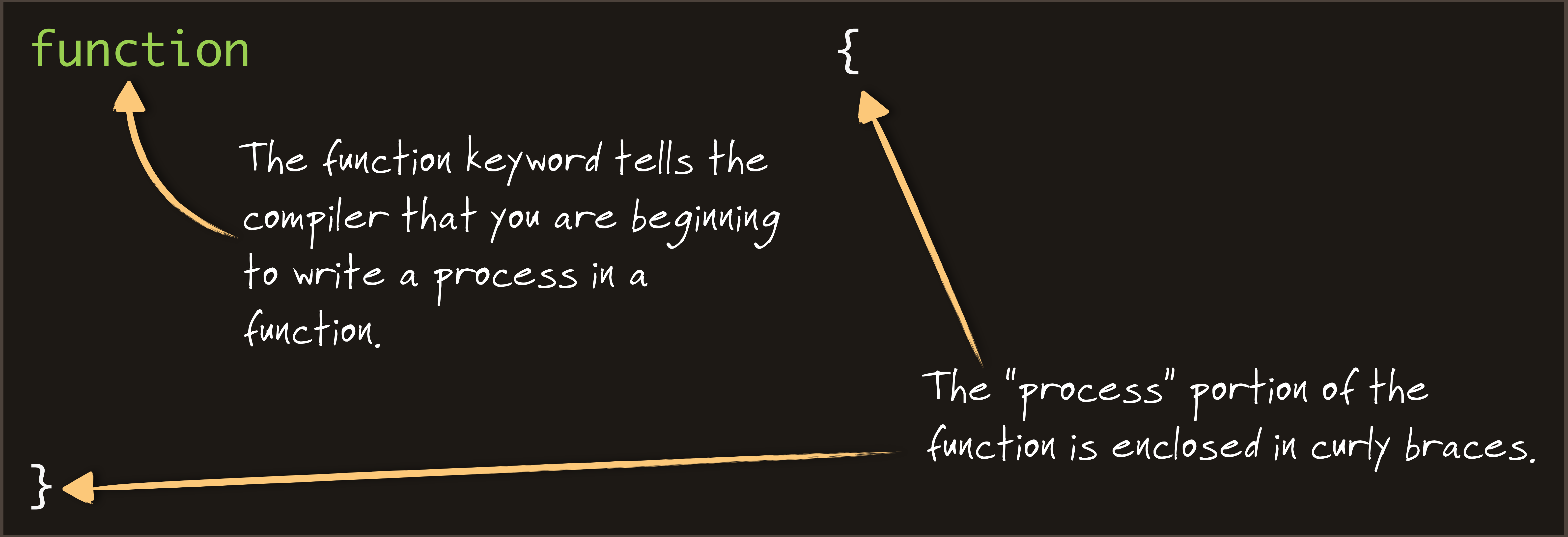
USEFULNESS THROUGH REUSABILITY

Wrapping our code in a function will allow us to reuse it



FUNCTIONS IN JAVASCRIPT CODE

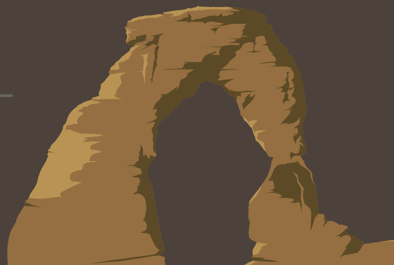
The syntax for a basic function structure



```
function                          {



}
```

The function keyword tells the compiler that you are beginning to write a process in a function.

The "process" portion of the function is enclosed in curly braces.



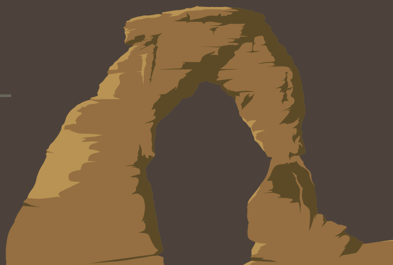
FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure

```
function  sumOfCubes  (a, b) {



}
```

The function's name follows the function keyword and should indicate briefly what's going on in the process.

Parameters are passed in a set of parentheses before the first curly brace. They are the "materials" the function will "work on".



FUNCTIONS IN JAVASCRIPT CODE

The syntax for a basic function structure

```
function  sumOfCubes  (a, b)  {

    *do some stuff*

    return *something (or nothing) from the process*
}
```

Inside the braces, the process occurs. In other words, the function does what it is intended to do.

This return keyword says to the function, "OK, we're done, now give us the result of what we did." It can be used anywhere in the function to stop the function's work. Here, that happens to be at the very end.

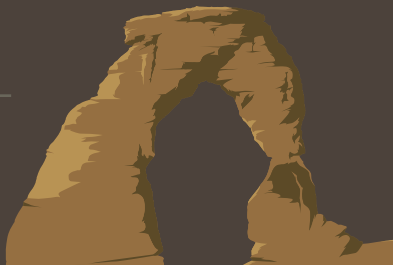
BUILDING OUR SUMOFCUBES FUNCTION

Assigning steps of the process to the function syntax

```
function  sumOfCubes (a, b) {

        2. Cube each number

        3. Sum the cubes

    return Sum
}
```

1. Get two numbers

4. Return the answer



BUILDING OUR SUMOFCUBES FUNCTION

Assigning steps of the process to the function syntax

```
function  sumOfCubes  (a, b)  {

     var aCubed = a*a*a;
     var bCubed = b*b*b;
     var sum = aCubed + bCubed;

     return sum;

}
```

Once the parameters are passed into the function, they are accessible at any point within the process.

CALLING OUR SUMOFCUBES FUNCTION

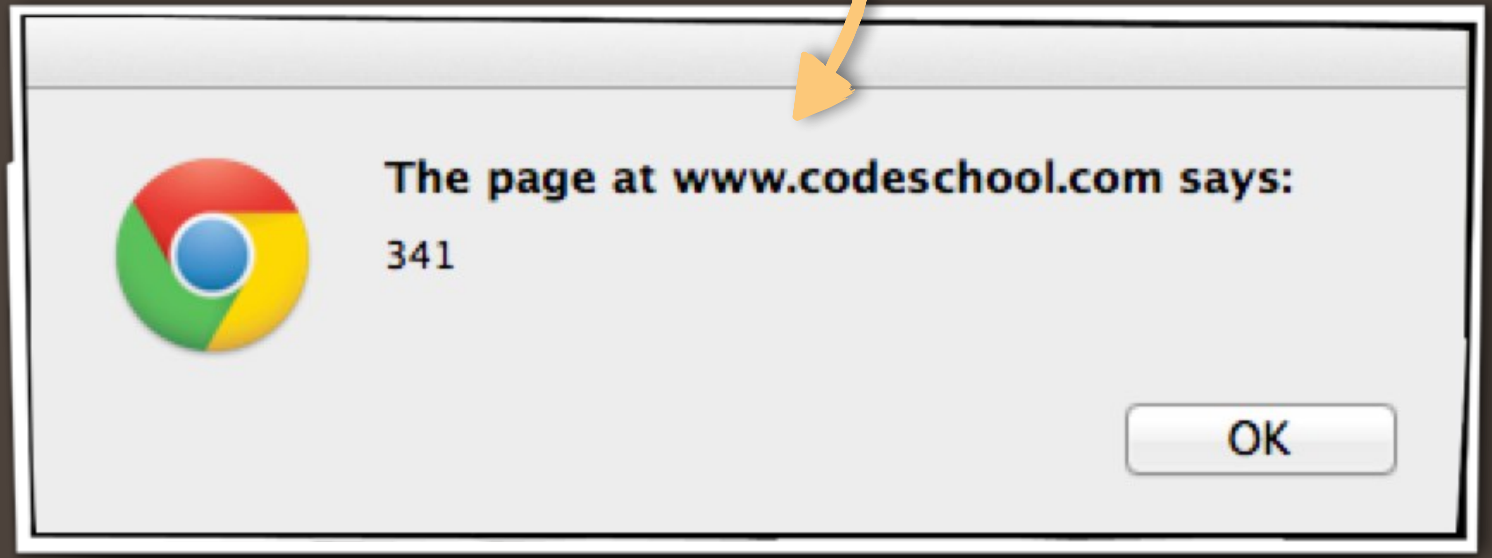
Now we can call the function using any parameter values we want!

```
function  sumOfCubes  (a, b)  {

   var aCubed = a*a*a;
   var bCubed = b*b*b;
   var sum = aCubed + bCubed;

   return sum;

}
```

```
sumOfCubes(4, 9);
```

$\rightarrow$ 793

```
var mySum = sumOfCubes(5, 6);
alert(mySum);
```



The page at www.codeschool.com says:

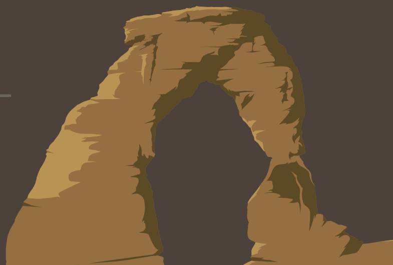
341

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

Our function does what it is supposed to, but it's not as efficient as it could be memory-wise. We've made three unnecessary variables that all have to be allocated in memory.



WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    return aCubed + bCubed;
}
```

The return keyword can calculate the results of an expression before actually returning from the function. One variable down!

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    return aCubed + bCubed;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    return aCubed + b*b*b;
}
```

One more variable down! Why make a bCubed when we can just use the calculation as a substitute? You can guess, then, what's coming next.

WRITING EFFICIENT FUNCTIONS

Being concise helps conserve memory and limits storage operations

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    var sum = aCubed + bCubed;
    return sum;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    var bCubed = b*b*b;
    return aCubed + bCubed;
}
```

```
function sumOfCubes(a, b) {

    var aCubed = a*a*a;
    return aCubed + b*b*b;
}
```

✓

```
function sumOfCubes(a, b) {

    return a*a*a + b*b*b;
}
```

Woohoo! One statement!

OUR FUNCTION IN ACTION

Calling a function involves the function name and some parameters

```
function sumOfCubes(a, b) {
    return a*a*a + b*b*b;
}
```

✓

```
sumOfCubes(4, 9);
```

$\rightarrow$ 793

Parameters can also be expressions, which the function will resolve before starting:

```
sumOfCubes(1+2, 3+5);
```

$\rightarrow$ 539

Same as (3, 8)

```
var x = 3;
sumOfCubes(x*2, x*4);
```

$\rightarrow$ 1494

Same as (6, 12)

NOW FOR A MORE COMPLEX FUNCTION!

Let's design a function that counts "E's" in a user-entered phrase

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

    }



}
```

By the way, sometimes functions don't need any parameters!

Always check that a user input is valid before any operations

Using the return keyword here will allow us to exit and inform the program of an invalid entry.

NOW FOR A MORE COMPLEX FUNCTION!

Let's design a function that counts "E's" in a user-entered phrase

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*
    }*otherwise*{



    }
}
```

This block will be where the function begins to actually check the phrase out and count the E's.

NOW FOR A MORE COMPLEX FUNCTION!

Let’s design a function that counts “E’s” in a user-entered phrase

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*
    }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*
    }
}
```

We have to count lowercase as well as uppercase!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    *ask user for a phrase to check*
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

    }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*
    }
}
```

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    var phrase = prompt("Which phrase would you like to examine?");
    *if the entry is invalid*{
            *alert the user*
            *exit function with a failure report*

    }*otherwise*{
            *make a counter for the E's*
            *for each character in the user's entry*{
                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*
    }
}
```

The prompt() method helps us get the user's entry.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {


     }
      *otherwise*{
          *make a counter for the E's*
          *for each character in the user's entry*{
               *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
               }
          }
          *alert the amount of E's in the phrase and return success*
     }
}
```

The typeof keyword allows us to determine whether the user has entered a valid string. This != expression returns true or false.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    var phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
        alert("That's not a valid entry!");
        return false;
    }
     *otherwise*{
        *make a counter for the E's*
        *for each character in the user's entry*{
            *if the character is an 'E' or an 'e'*{
                *increment the E counter*
            }
        }
        *alert the amount of E's in the phrase and return success*
    }
}
```

If the entry is not a string, we alert the user and exit the function, returning false.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
          alert("That's not a valid entry!");
          return false;
     } else {

          *make a counter for the E's*
          *for each character in the user's entry*{
               *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
               }
          }
          *alert the amount of E's in the phrase and return success*
     }
}
```

Else-blocks help us do the "otherwise" case!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
            alert("That's not a valid entry!");
            return false;
     } else {

            var eCount = 0;
            for (var index = 0; index < phrase.length; index++) {

                *if the character is an 'E' or an 'e'*{
                    *increment the E counter*
                }
            }
            *alert the amount of E's in the phrase and return success*
     }
}
```

We want to start at index 0, and go until one less than the length of the user's string. Remember that strings have zero-based indices!

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
    var phrase = prompt("Which phrase would you like to examine?");
    if ( typeof(phrase) != "string" ) {
            alert("That's not a valid entry!");
            return false;
    } else {

            var eCount = 0;
            for (var index = 0; index < phrase.length; index++) {
                if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E')
                    eCount++;

                }
            }
            *alert the amount of E's in the phrase and return success*

    }
}
```

This complex conditional checks whether the spot we're currently at along the string is either an E or an e.

If we found one, we'll increase our counter.

FILLING IN COUNTE( ) WITH CODE

How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
          alert("That's not a valid entry!");
          return false;
     } else {

          var eCount = 0;
          for (var index = 0; index < phrase.length; index++) {
               if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E')
                    eCount++;
               }
          }

          *alert the amount of E's in the phrase and return success*

     }
}
```

This complex conditional checks whether the spot we're currently at along the string is either an E or an e.

FILLING IN COUNTE( ) WITH CODE

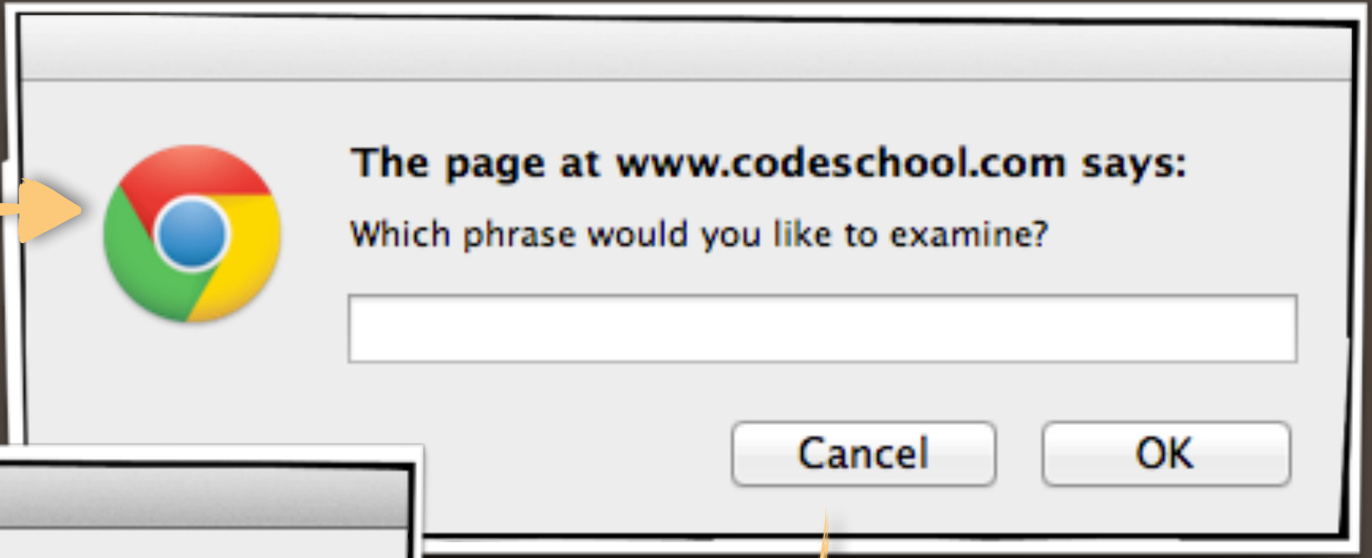
How can we get the behavior we've described in our pseudo-function?

```
function countE ( ) {
     var phrase = prompt("Which phrase would you like to examine?");
     if ( typeof(phrase) != "string" ) {
             alert("That's not a valid entry!");
             return false;
     } else {

             var eCount = 0;
             for (var index = 0; index < phrase.length; index++) {
                 if (phrase.charAt(index) == 'e' || phrase.charAt(index) == 'E')
                     eCount++;
                 }
             }
             alert("There are " + eCount + " E's in \"" + phrase + "\".");
             return true;
     }
}
```

After our for loop, eCount will contain the total number of E's and e's in our loop.

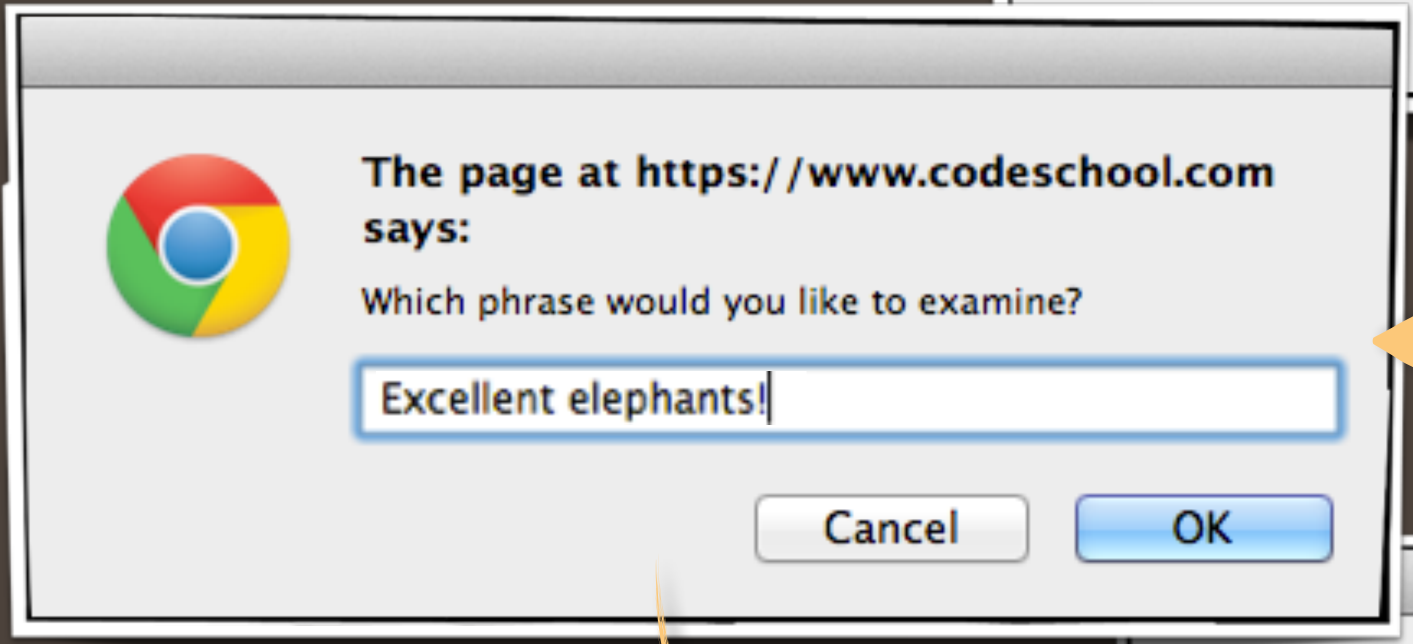
THE SEQUENCE OF ENTRY

```
> countE()
```



The page at www.codeschool.com says:

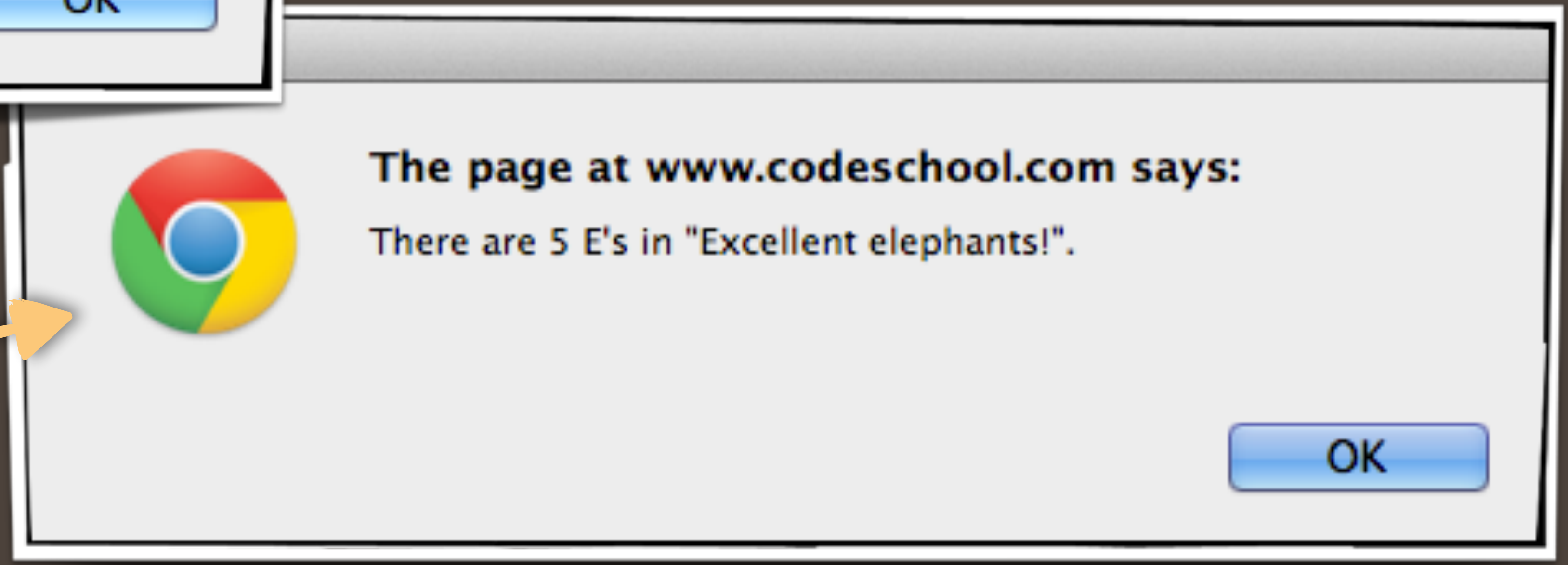
Which phrase would you like to examine?



The page at https://www.codeschool.com says:

Which phrase would you like to examine?

Excellent elephants!



The page at www.codeschool.com says:

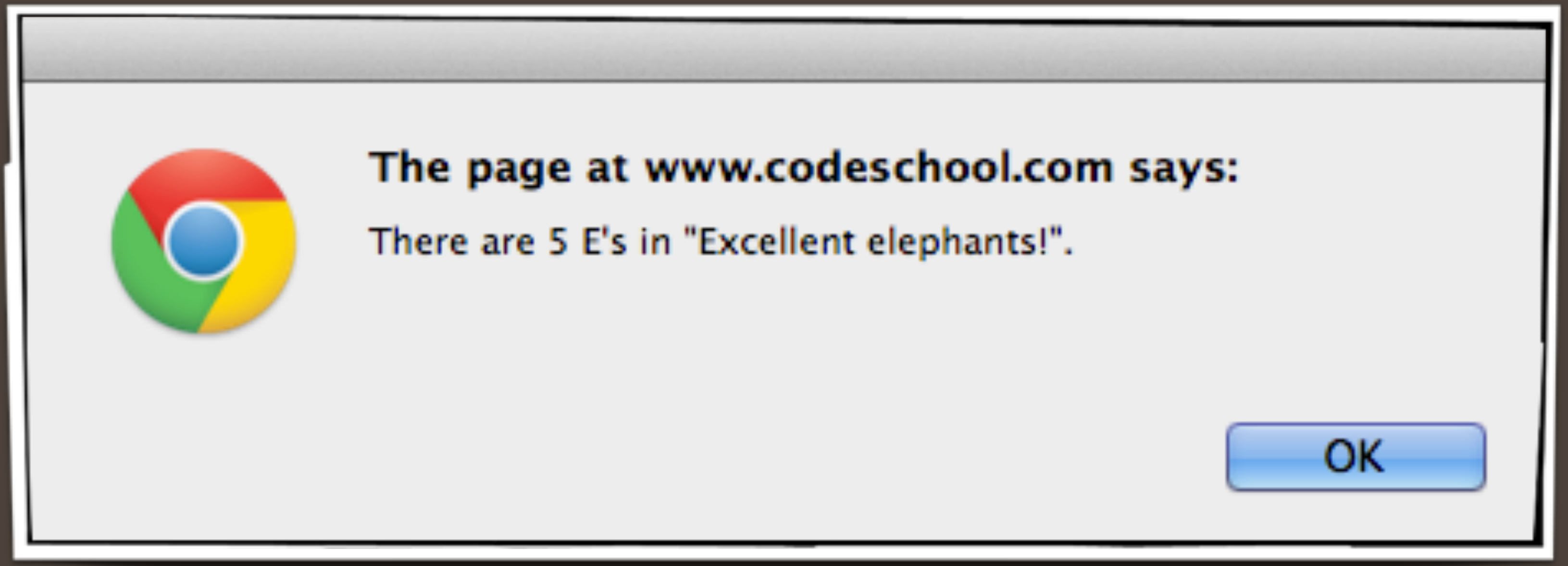
There are 5 E's in "Excellent elephants!".

TRACING OUR E-COUNTER

Following our function's code as it counts E's in "Excellent elephants!"

index	LOOP: index < length?	charAt (index)	is charAt(index) an E or e?	eCount
0	TRUE	E	TRUE	1
1	TRUE	x	FALSE	1
2	TRUE	c	FALSE	1
3	TRUE	e	TRUE	2
4	TRUE	l	FALSE	2
5	TRUE	l	FALSE	2
6	TRUE	e	TRUE	3
7	TRUE	n	FALSE	3
8	TRUE	t	FALSE	3
9	TRUE	(space)	FALSE	3
10	TRUE	E	TRUE	4
11	TRUE	l	FALSE	4
12	TRUE	e	TRUE	5
13	TRUE	p	FALSE	5

index	LOOP: index < length?	charAt (index)	is charAt(index) an E or e?	eCount
14	TRUE	h	FALSE	5
15	TRUE	a	FALSE	5
16	TRUE	n	FALSE	5
17	TRUE	t	FALSE	5
18	TRUE	s	FALSE	5
19	TRUE	!	FALSE	5
20	FALSE	STOP!		


The page at www.codeschool.com says:

There are 5 E's in "Excellent elephants!".

UNDERSTANDING LOCAL AND GLOBAL SCOPE

Visualizing worlds within worlds…

```
var x = 6;
var y = 4;

function add (a, b){

    var x = a + b;
    return x;
}

function subtract (a, b){

    y = a - b;
    return y;
}
```

Out here, in the main program, the scope is "global", which means that variables declared are potentially accessible from everywhere.

Inside functions, the scope is "local", like cities within a state. Each has their own "government" and stuff that happens in here stays in here.

FUNCTIONS CREATE A NEW SCOPE

Variables declared in a function STAY in the function

```
var x = 6
function add (a, b){

    var x = a + b;
    return x;
}
```

```
add(9, 2);
```

→ 11

```
console.log(x)
```

→ 6

The circled variable only exists in the function's local scope. Because it has been declared with var, it doesn't modify the same-named variable "outside" the function.

```
var x = 6
function add (a, b){

    x = a + b;
    return x;
}
```

```
add(9, 2);
```

→ 11

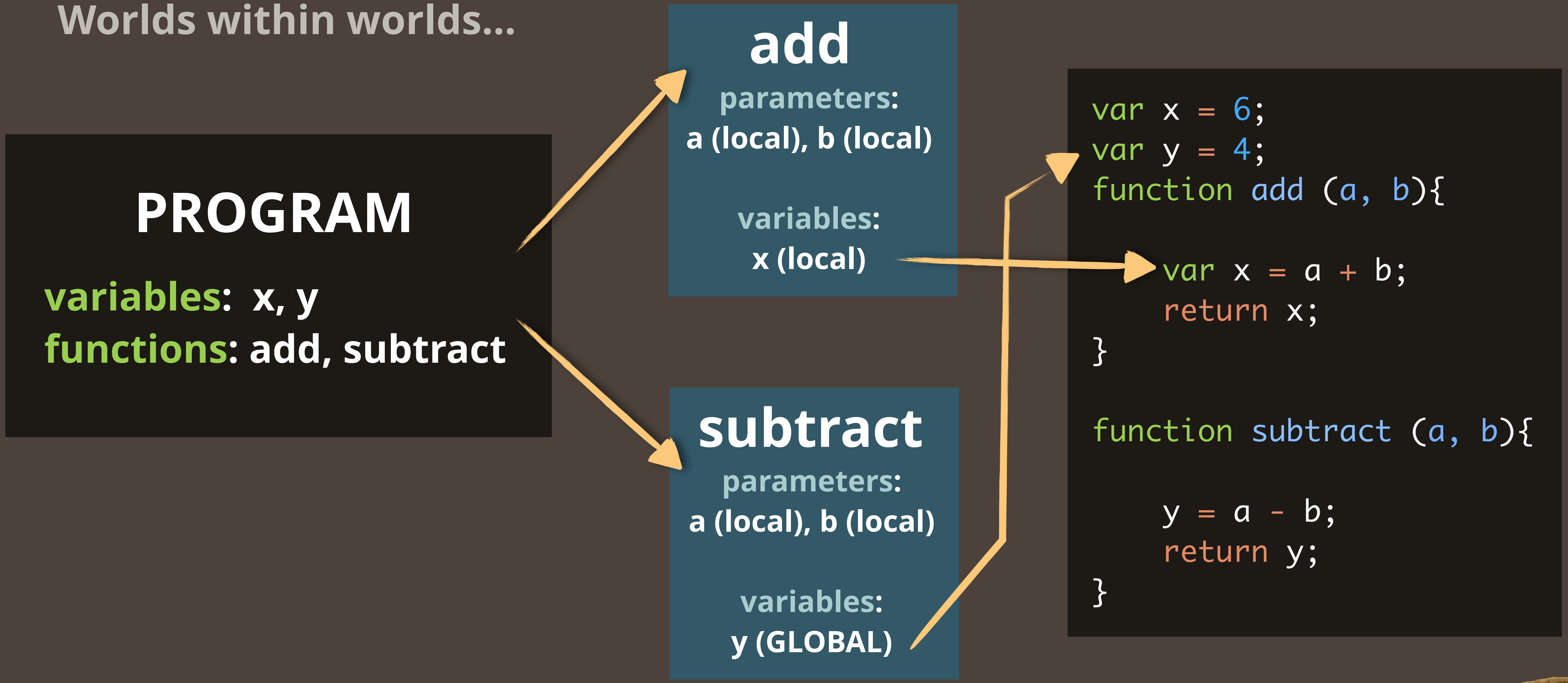
```
console.log(x)
```

→ 11

If the x were not declared with var, it "shadows" the same-named variable from the nearest external scope!

VISUALIZING LOCAL AND GLOBAL SCOPE

Worlds within worlds...



```
var x = 6;
var y = 4;
function add (a, b){

    var x = a + b;
    return x;
}

function subtract (a, b){

    y = a - b;
    return y;
}
```

THE ARRAY ARCHIPELAGO

WHAT IF WE WANTED A PASSENGER LIST?

How would we structure a list of passengers inside our train.js system?

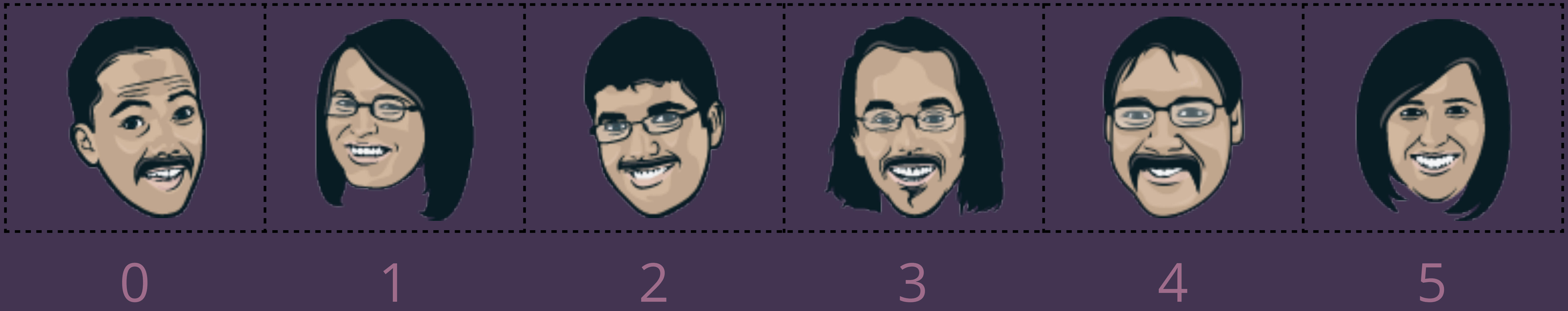
trains.js

```
...
function makeList ( ) {

    var passengerOne = "Gregg Pollack";
    var passengerTwo = "Aimee Simone";
    var passengerThree = "Thomas Meeks";
    var passengerFour = "Olivier Lacan";

    ...and on and on, typing through a list
    of sixty passengers, that might
    even change later?? No way.
}
...
```

THE ARRAY

An array is a data structure with automatically indexed positions

A 6-cell Array of Passengers

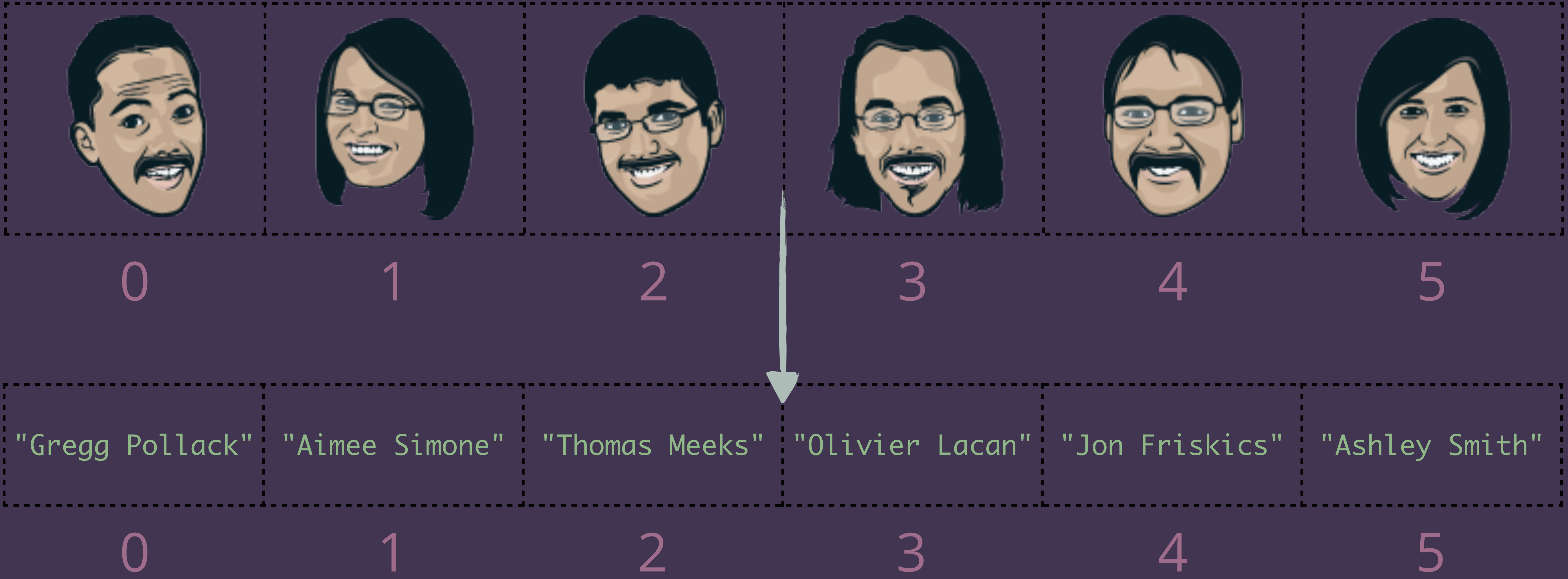


Just like Strings, Arrays have indices that are zero-based.

Despite his excellent disguise, it looks like Jon is in index 4. We mustache him a question.

ARRAY CELLS CAN HOLD ANY VALUE

Our picture array could also be an array of strings.



BUILDING AND ACCESSING ARRAYS

Easy to build, easy to access with indices

"Gregg Pollack"	"Aimee Simone"	"Thomas Meeks"	"Olivier Lacan"	"Jon Friskics"	"Ashley Smith"
0	1	2	3	4	5

To build this array in code, we write:

```
var passengers = [ "Gregg Pollack", "Aimee Simone", "Thomas Meeks",
                   "Olivier Lacan", "Jon Friskics", "Ashley Smith"];
```

The brackets indicate to the compiler to make an array and fill it with the comma-separated values between the brackets.

If we wanted to access any particular index's value, we use:

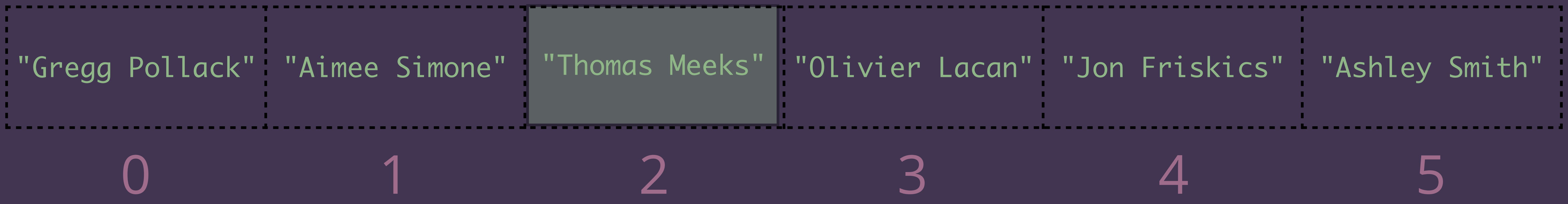
```
passengers[5];
```

→ "Ashley Smith"

Returns the value at index 5.

CHANGING ARRAY CONTENTS

We can also reference and change specific cells with indices



If we wanted to change the value contained at any index, we use:

```
passengers[2] = "Eric Allam";
```

This syntax says "Go over to index 2, and change its value to whatever comes after the = sign.

CHANGING ARRAY CONTENTS

We can also reference and change specific cells with indices

"Gregg Pollack"	"Aimee Simone"	"Eric Allam"	"Olivier Lacan"	"Jon Friskics"	"Ashley Smith"
0	1	2	3	4	5

If we wanted to change the value contained at any index, we use:

```
passengers[2] = "Eric Allam";
```

This syntax says "Go over to index 2, and change its value to whatever comes after the = sign.

Like Strings, we can access the length of Arrays:

```
passengers.length;
```

→ 6

The length of an array is the actual number of cells, including any empty cells.

THE POP() FUNCTION

Removing a cell from the back of the array

"Gregg Pollack"	"Aimee Simone"	"Eric Allam"	"Olivier Lacan"	"Jon Friskics"	~~"Ashley Smith"~~
0	1	2	3	4	5

The pop( ) function deletes the last position and retrieves its value:

```
passengers.pop();
```

→ "Ashley Smith"

pop() will automatically "pop" the last existing cell off the array while returning that cell's contents.

"Gregg Pollack"	"Aimee Simone"	"Eric Allam"	"Olivier Lacan"	"Jon Friskics"
0	1	2	3	4

The array length automatically adjusts!

THE PUSH() FUNCTION

Adding a cell and its contents to the back of the array

"Gregg Pollack"	"Aimee Simone"	"Eric Allam"	"Olivier Lacan"	"Jon Friskics"
0	1	2	3	4

The push( ) function adds a cell in the last position and enters a value:

```
passengers.push("Adam Rensel");
```

push() will "push" a cell onto the back of the array and automatically increase the array length.

"Gregg Pollack"	"Aimee Simone"	"Eric Allam"	"Olivier Lacan"	"Jon Friskics"	"Adam Rensel"
0	1	2	3	4	5

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var comboArray1 = ["One", "fish", 2, "fish"];
```

"One"	"fish"	2	"fish"
0	1	2	3

```
var poisson = "fish";
```

The variable name disappears in the array and just the contents remain.

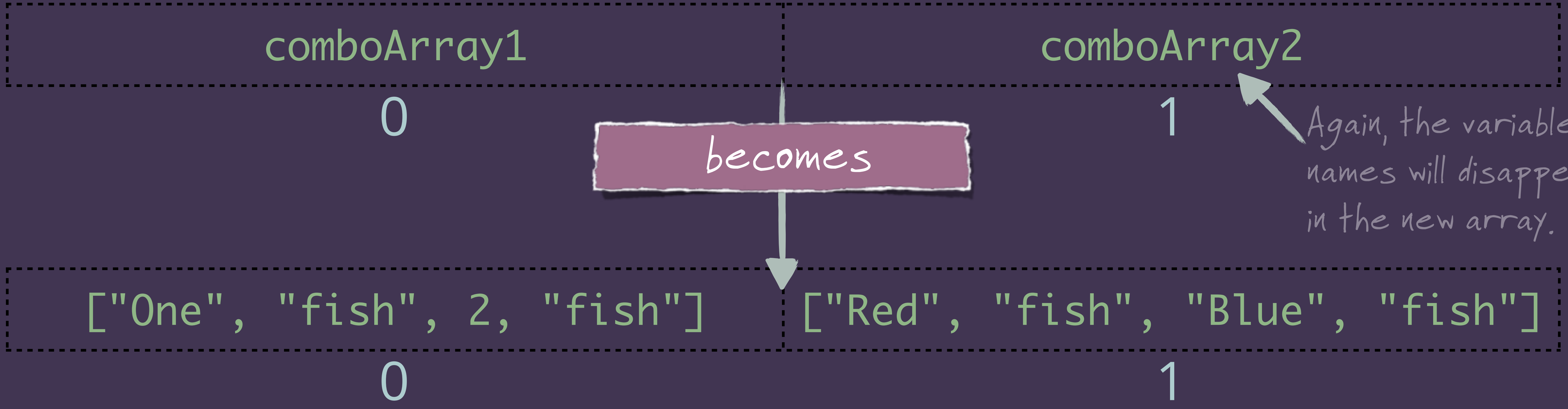
```
var comboArray2 = ["Red", poisson, "Blue", poisson];
```

"Red"	"fish"	"Blue"	"fish"
0	1	2	3

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var arrayOfArrays = [comboArray1, comboArray2];
```



Again, the variable names will disappear in the new array.

```
console.log( arrayOfArrays );
```

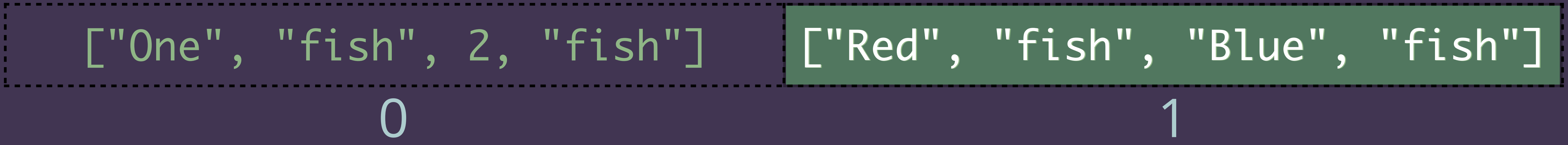
```
→ [ Array[4], Array[4] ]
```

Here, the [4] and [4] are providing the lengths of each of the arrays, which here happen to be the same.

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var arrayOfArrays = [comboArray1, comboArray2];
```



```
console.log( arrayOfArrays[1] );
```

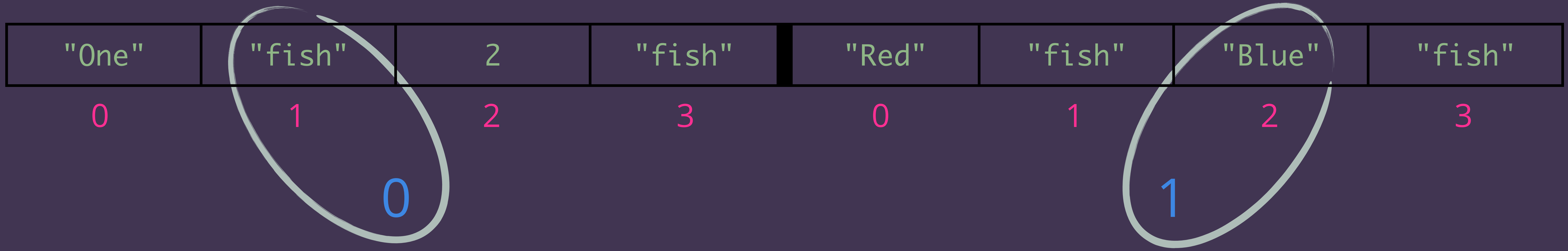
```
→ ["Red", "fish", "Blue", "fish"]
```

When we reference the [1] index of arrayOfArrays, we get another entire array because that's what the cell contains. Specifically, our earlier comboArray2.

ARRAYS CAN HOLD LOTS OF STUFF

Strings, values, variables, other arrays, and combinations of them all!

```
var arrayOfArrays = [comboArray1, comboArray2];
```



The first bracket selects a cell in the master array.

The second bracket then selects a cell in the lower level array

```
console.log( arrayOfArrays[1][2] );
```

→ Blue

```
console.log( arrayOfArrays[0][1] );
```

→ fish

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
for (var i = 0; i < numberList.length; i++){

}
```

You'll often see the variable `i` used as a loop counter by convention and for simplicity.

To look through our entire array, we continue only until we have reached the last index of the zero-based array. Since our array has a `length` of `10`, we want to stop checking at index `9`.

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
for (var i = 0; i < numberList.length; i++){

  console.log("The value in cell " + i + " is " + numberList[i]);

}
```

Our loop counter can also serve as a current index position, helping us "iterate" over the entire contents of the array in order.

Don't confuse the index number (the *position*) with the contents of the cell (the *value*)!

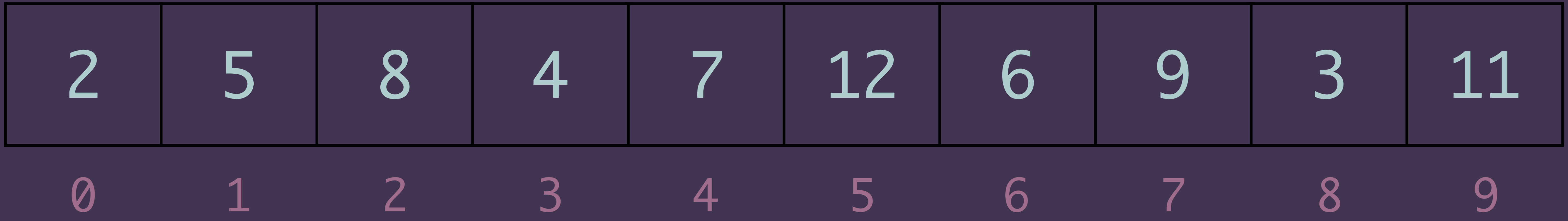
USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

i	i < numberList.length ?	numberList[i]	printout
0	TRUE	2	The value in cell 0 is 2
1	TRUE	5	The value in cell 1 is 5
2	TRUE	8	The value in cell 2 is 8
3	TRUE	4	The value in cell 3 is 4
4	TRUE	7	The value in cell 4 is 7
5	TRUE	12	The value in cell 5 is 12
6	TRUE	6	The value in cell 6 is 6
7	TRUE	9	The value in cell 7 is 9
8	TRUE	3	The value in cell 8 is 3
9	TRUE	11	The value in cell 9 is 11
10	FALSE	NA	STOP!

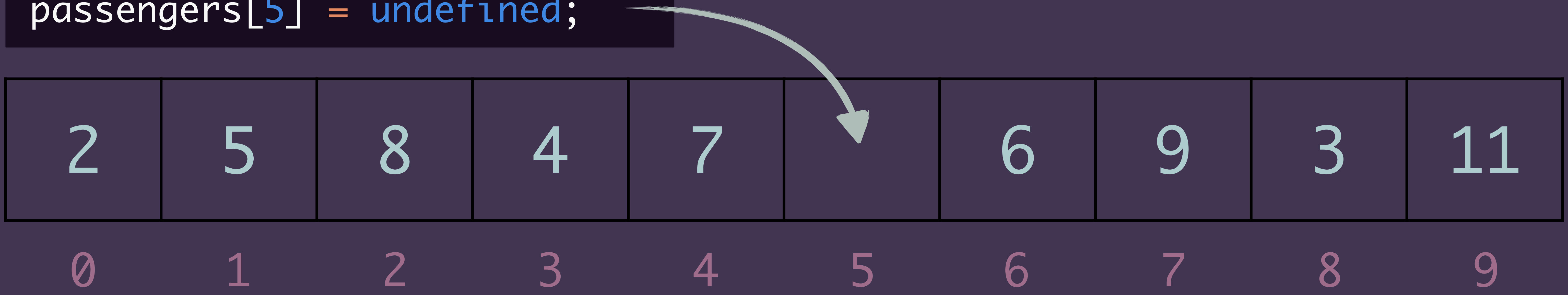
EMPTY CELLS IN ARRAYS?

Using the undefined value to create “empty” cells.



To make a cell empty, we’ll use the special `undefined` value, which means “no contents.”

```
passengers[5] = undefined;
```



A NEW FUNCTION WITH ARRAYS

Let's count even numbers AND erase odds.

```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
var evenCount = 0;
for (var i = 0; i < numberList.length; i++) {
     if (numberList[i] % 2 == 0) {
          evenCount++;
     }


}
```

We'll set up a counter before the loop.

Even numbers will have a zero remainder when divided by 2!

A NEW FUNCTION WITH ARRAYS

Let's count even numbers AND erase odds.

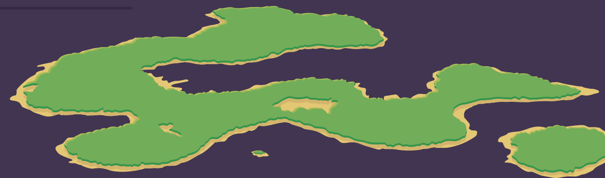
```
var numberList = [ 2, 5, 8, 4, 7, 12, 6, 9, 3, 11 ];
```

```
var evenCount = 0;
for (var i = 0; i < numberList.length; i++) {
     if (numberList[i] % 2 == 0) {
            evenCount++;
     } else {
            numberList[i] = undefined;
     }
}
```

Otherwise, if not's even, we know it's odd! Here's where we will use undefined.

```
console.log(evenCount);
```

→ 5



USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

i	i < numberList.length ?	numberList[i]	numberList[i] % 2 == 0 ?	evenCount
0	TRUE	2	**TRUE**	**1**
1	TRUE	5	FALSE	1
2	TRUE	8	**TRUE**	**2**
3	TRUE	4	**TRUE**	**3**
4	TRUE	7	FALSE	3
5	TRUE	12	**TRUE**	**4**
6	TRUE	6	**TRUE**	**5**
7	TRUE	9	FALSE	5
8	TRUE	3	FALSE	5
9	TRUE	11	FALSE	5
10	FALSE	NA	STOP!	

USING LOOPS WITH ARRAYS

Loops help us move through all indices of an array

```
console.log(numberList);
```

```
→ [2, undefined, 8, 4, undefined, 12, 6, undefined, undefined, undefined]
```

All of our empty spaces
are saved inside the array!

```
console.log(numberList.length);
```

```
→ 10
```

The length of the array stayed unchanged.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( *passenger's name*, *array of passengers*) {
        *if list is empty* {
            *add passenger to list*
        } *else* {
            *for all spots in the list*{
                *if the current spot is empty* {
                    *add passenger to that spot*
                    *return the list and exit the function*
                } *else, if the end of the list is reached* {
                    *add passenger to end of list*
                    *return the list and exit the function*
                }
            }
        }
}
```

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {
      if (list.length == 0) {
          *add passenger to list*
      } *else* {
           *for all spots in the list*{
                 *if the current spot is empty* {
                       *add passenger to that spot*
                       *return the list and exit the function*
                 } *else, if the end of the list is reached* {
                       *add passenger to end of list*
                       *return the list and exit the function*
                 }
           }
      }
}
```

A length of 0 means the array is empty.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name, list ) {

        if (list.length == 0) {
             list.push(name);
        } else {
              *for all spots in the list*{
                      *if the current spot is empty* {
                              *add passenger to that spot*
                              *return the list and exit the function*
                      } *else, if the end of the list is reached* {
                              *add passenger to end of list*
                              *return the list and exit the function*
                      }
              }
        }
}
```

We start the list by pushing a passenger into the empty array.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {

      if (list.length == 0) {
          list.push(name);
      } else {
           for (var i = 0; i < list.length; i++) {
                *if the current spot is empty*  {
                      *add passenger to that spot*
                      *return the list and exit the function*
                }  *else, if the end of the list is reached*  {
                      *add passenger to end of list*
                      *return the list and exit the function*
                }
           }
      }
}
```

We want to check all spots in the list, which will include all indices through `list.length - 1`

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
           for (var i = 0; i < list.length; i++) {
                 if(list[i] == undefined){
                      list[i] = name;
                       *return the list and exit the function*
                 }  *else, if the end of the list is reached*  {
                       *add passenger to end of list*
                       *return the list and exit the function*
                 }
           }
      }
}
```

If a passenger spot has been emptied, it will be undefined. We want to fill that empty spot before adding more spots to the list.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {
      if (list.length == 0) {
          list.push(name);
      } else {
           for (var i = 0; i < list.length; i++) {
                 if(list[i] == undefined){
                      list[i] = name;
                      return list;
                 } *else, if the end of the list is reached* {
                      *add passenger to end of list*
                      *return the list and exit the function*
                 }
           }
      }
}
```

If we've placed the passenger name, then we're done! No need to keep looping. We can now return the updated list and exit the function.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {

      if (list.length == 0) {
          list.push(name);
      } else {
           for (var i = 0; i < list.length; i++) {
                 if(list[i] == undefined){
                      list[i] = name;
                      return list;
                 } else if (i == list.length - 1) {
                      list.push(name);
                       *return the list and exit the function*
                 }
           }
      }
}
```

If we have reached the final index of list without finding an empty spot, then push the name onto the end of list.

BUILDING A PASSENGER LIST

Using an array and functions to keep track of train passengers

```
function addPassenger ( name,  list ) {

       if (list.length == 0) {
            list.push(name);
       } else {
             for (var i = 0; i < list.length; i++) {
                    if(list[i] == undefined){
                          list[i] = name;
                          return list;
                    } else if (i == list.length - 1) {
                          list.push(name);
                          return list;
                    }
             }
       }
}
```

If the list was initially empty, we can return the updated list and exit.

CREATING A NEW PASSENGER LIST

Let’s make a new list and add a few passengers to it.

```
var passengerList = [ ];
```

An empty set of brackets will create an array with no cells.

```
passengerList = addPassenger("Gregg Pollack", passengerList );
```

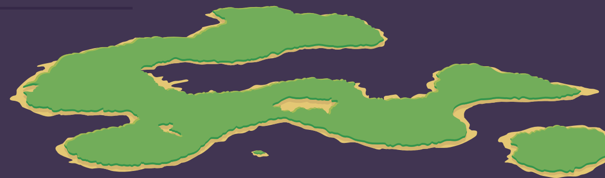
```
["Gregg Pollack"]
```

```
passengerList = addPassenger("Ashley Smith", passengerList );
```

```
["Gregg Pollack", "Ashley Smith"]
```

```
passengerList = addPassenger("Jon Friskics", passengerList );
```

```
["Gregg Pollack", "Ashley Smith", "Jon Friskics"]
```



REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        }









}
```

If the list is empty, log it to the user.

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {






            }
        }
}
```

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
            console.log("List is empty!");
        } else  {
            for (var i = 0; i < list.length; i++) {
                  if(list[i] == name){
                        list[i] = undefined;



            }
        }
}
```

If the contents of the index match the name exactly, delete it by setting the index to `undefined`.

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
             console.log("List is empty!");
        } else  {
             for (var i = 0; i < list.length; i++) {
                    if(list[i] == name){
                          list[i] = undefined;
                          return list;



             }
       }
}
```

Once we've deleted the passenger, we don't need any more loop cycles, so return will exit the entire function with the updated list.

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {
        if (list.length == 0){
             console.log("List is empty!");
        } else  {
             for (var i = 0; i < list.length; i++) {
                    if(list[i] == name){
                          list[i] = undefined;
                          return list;
                    } else if (i == list.length - 1) {
                           console.log("Passenger not found!");
                    }
             }
        }
}
```

If we get to the end, and we haven't deleted a name, then we know the passenger wasn't present!

REMOVING PASSENGERS

Using an array and functions to keep track of train passengers

```
function deletePassenger ( name, list ) {

        if (list.length == 0){
             console.log("List is empty!");
        } else  {
             for (var i = 0; i < list.length; i++) {
                    if(list[i] == name){
                          list[i] = undefined;
                          return list;
                    } else if (i == list.length - 1) {
                           console.log("Passenger not found!");
                    }
             }
        }
        return list;
}
```

If the list was empty, or if we never found the passenger, we just return the same list.

MODIFYING OUR PASSENGER LIST

Let's take some passengers out, and put some back in.

```
passengerList = ["Gregg Pollack", "Ashley Smith", "Jon Friskics"];
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
["Gregg Pollack", undefined, "Jon Friskics" ]
```

```
passengerList = addPassenger( "Adam Rensel", passengerList );
```

```
["Gregg Pollack", "Adam Rensel", "Jon Friskics" ]
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
→ Passenger not found!
```

MODIFYING OUR PASSENGER LIST

Let's take some passengers out, and put some back in.

```
passengerList = ["Gregg Pollack", "Adam Rensel", "Jon Friskics"];
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
→ Passenger not found!
```

MODIFYING OUR PASSENGER LIST

Let's take some passengers out, and put some back in.

```
passengerList = ["Gregg Pollack", "Adam Rensel", "Jon Friskics"];
```

```
passengerList = deletePassenger( "Ashley Smith", passengerList );
```

```
→ Passenger not found!
```

```
passengerList = deletePassenger("Gregg Pollack", passengerList );
```

```
[undefined, "Adam Rensel", "Jon Friskics" ]
```

```
passengerList = addPassenger("Jennifer Borders", passengerList );
```

```
["Jennifer Borders", "Adam Rensel", "Jon Friskics" ]
```