CS 50 Web Design

APCSP Module 2: Internet

Unit 3: JavaScript

LECTURE 9: ARRAYS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- The basic definition for JavaScript array: declaration, instantiation, and initialization
- JavaScript array operations and methods
- JavaScript array as the only linear data structure for the language: array, vector, stack, queue, deque, node for a tree, and etc.
- Array projects
- Hangman game project with random number generators.

Overview of Arrays

SECTION 1

Array object

- An ***array*** is an ordered set of values that you refer to with a name and an index. For example, you could have an array called emp that contains employees' names indexed by their numerical employee number. So emp[1] would be employee number one, emp[2] employee number two, and so on.

- JavaScript does not have an explicit array data type. However, you can use the predefined Array object and its methods to work with arrays in your applications. The **Array** object has methods for manipulating arrays in various ways, such as joining, reversing, and sorting them. It has a property for determining the array length and other properties for use with regular expressions.

Why Array?

- Let’s look at dinosaurs again. Say you want to use a program to keep track of the many kinds of dinosaurs you know about. You could create a variable for each dinosaur, like this:

```
var dinosaur1 = "T-Rex";
var dinosaur2 = "Velociraptor";
var dinosaur3 = "Stegosaurus";
var dinosaur4 = "Triceratops";
var dinosaur5 = "Brachiosaurus";
var dinosaur6 = "Pteranodon";
var dinosaur7 = "Apatosaurus";
var dinosaur8 = "Diplodocus";
var dinosaur9 = "Compsognathus";
```

Why Array?

- This list is pretty awkward to use, though, because you have nine different variables when you could have just one. Imagine if you were keeping track of 1000 dinosaurs!
- You'd need to create 1000 separate variables, which would be almost impossible to work with.

Declaration

SECTION 1

Arrays

- Objects allow you to store **keyed** collections of values. That’s fine.
- But quite often we find that we need an ordered collection, where we have a 1st, a 2nd, a 3rd element and so on. For example, we need that to store a list of something: users, goods, HTML elements etc.
- It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can’t insert a new property “between” the existing ones. Objects are just not meant for such use.
- There exists a special data structure named Array, to store **ordered collections**.

Declaration

- There are two syntaxes for creating an empty array:

```
1.var arr = new Array(); // use construction
2.var arr = [];          // use list notation.
```

Creating an Array

- To create an array, you just use square brackets, []. In fact, an **empty array** is simply a pair of square brackets, like this:

  [];

- What is the difference between null and empty array?
- To create an array with values in it, enter the values, separated by commas, between the square brackets.

Creating an Array

- We can call the individual values in an array items or elements. In this example, our elements will be strings (the names of our favorite dinosaurs), so we’ll write them with quote marks. We’ll store the array in a variable called dinosaurs:

```
var dinosaurs = ["T-Rex",
"Velociraptor",
  "Stegosaurus", "Triceratops",
"Brachiosaurus",
  "Pteranodon", "Apatosaurus",
  "Diplodocus", "Compsognathus“
];
```

Array Literals

SECTION 1

JavaScript : Array literals

Description

- In Javascript, an array literal is a list of expressions, each of which represents an array element, enclosed in a pair of square brackets ' [ ] ' . When an array is created using an array literal, it is initialized with the specified values as its elements, and its length is set to the number of arguments specified. If no value is supplied it creates an empty array with zero length.

JavaScript : Array literals

Examples:

- Creating an empty array :

  var fruits = [ ];

- Creating an array with four elements.

  var fruits = ["Orange", "Apple", "Banana", "Mango"]

Comma in array literals

- There is no need to specify all elements in an array literal. If we put two commas in a row at any position in an array then an unspecified element will be created in that place.
- The following example creates the fruits array :

  fruits = ["Orange", , "Mango"]
- This array has one empty element in the middle and two elements with values. (fruits[0] is "Orange", fruits[1] is set to undefined, and fruits[2] is "Mango").

Comma in array literals

- If you include a single comma at the end of the elements, the comma is ignored. In the following example, the length of the array is three. There are no fruits[2].

  fruits = ["Orange", "Mango",]

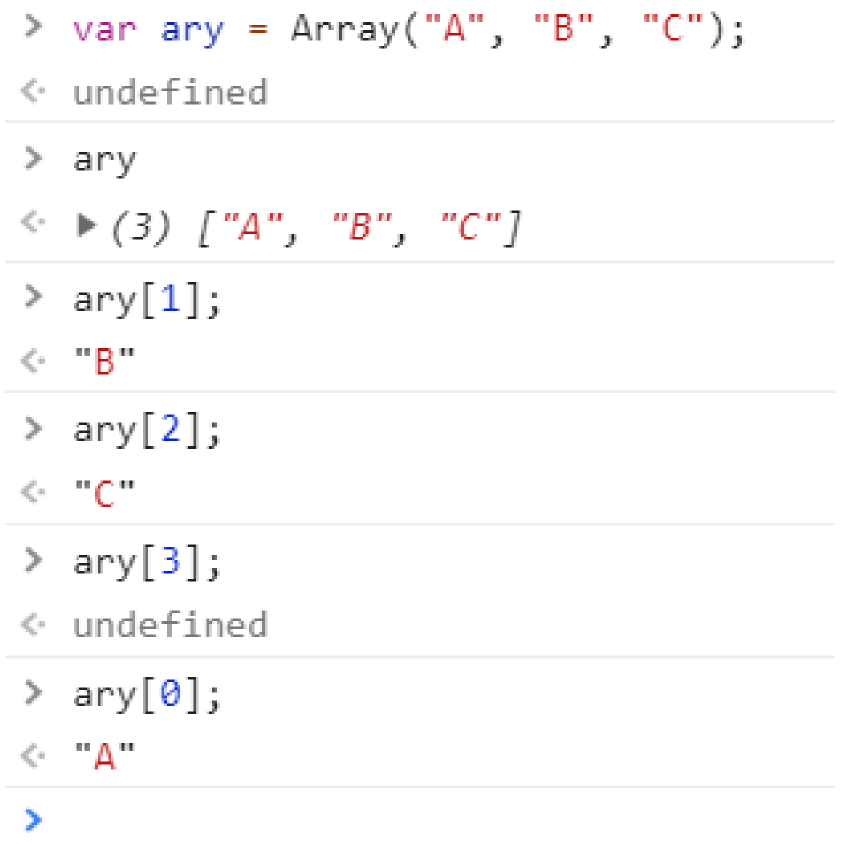
- In the following example, the length of the array is four, and fruits[0] and fruits[2] are undefined.

  fruits = [, 'Apple', , 'Orange'];

Creation of Array

Demo Using Chrome Console:

1. Using constructor Array to create an array.
2. ary[0]; // first element
3. ary[3]; // undefined (no such element)



```
> var ary = Array("A", "B", "C");
<· undefined
> ary
<· ▶ (3) ["A", "B", "C"]
> ary[1];
<· "B"
> ary[2];
<· "C"
> ary[3];
<· undefined
> ary[0];
<· "A"
>
```

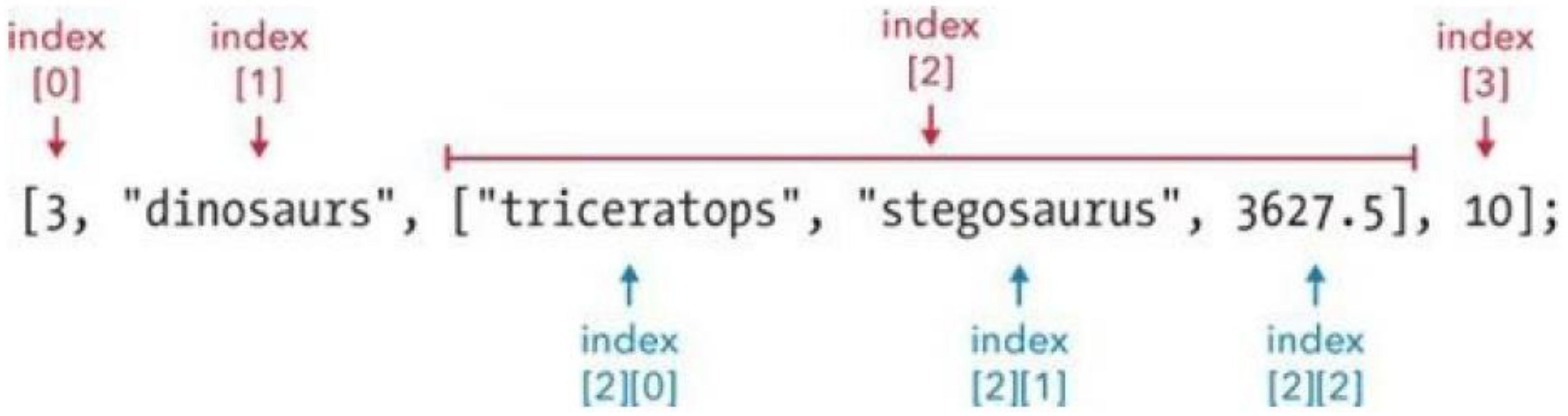
Heterogenous Array

SECTION 1

Mixing Data Types in an Array

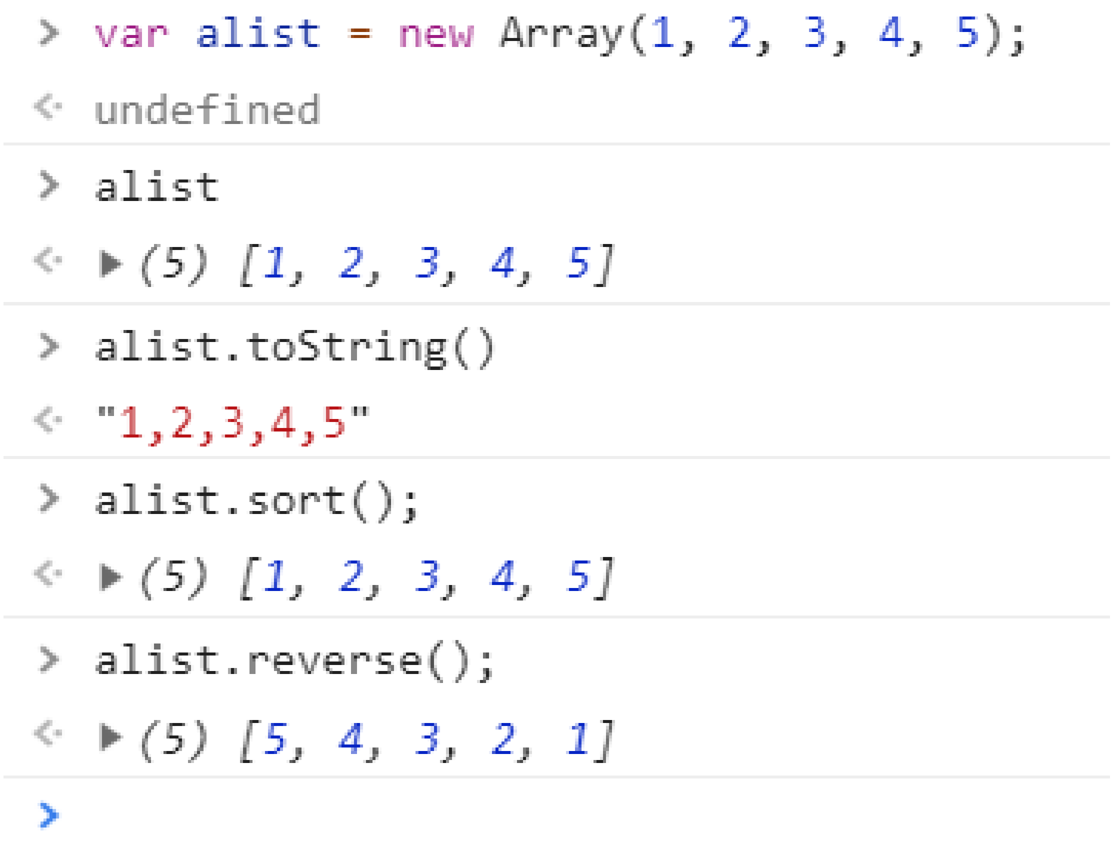
- Array elements don't all have to be the same type. For example, the next array contains a number (3), a string ("dinosaurs"), an array (["triceratops", "stegosaurus", 3627.5]), and another number (10):

```
var dinosaursAndNumbers =
[3, "dinosaurs", ["triceratops", "stegosaurus", 3627.5], 10];
```



Creating an Array Using new Operator

1. Use new operator to create an array.
2. toString()
3. sort();
4. reverse();

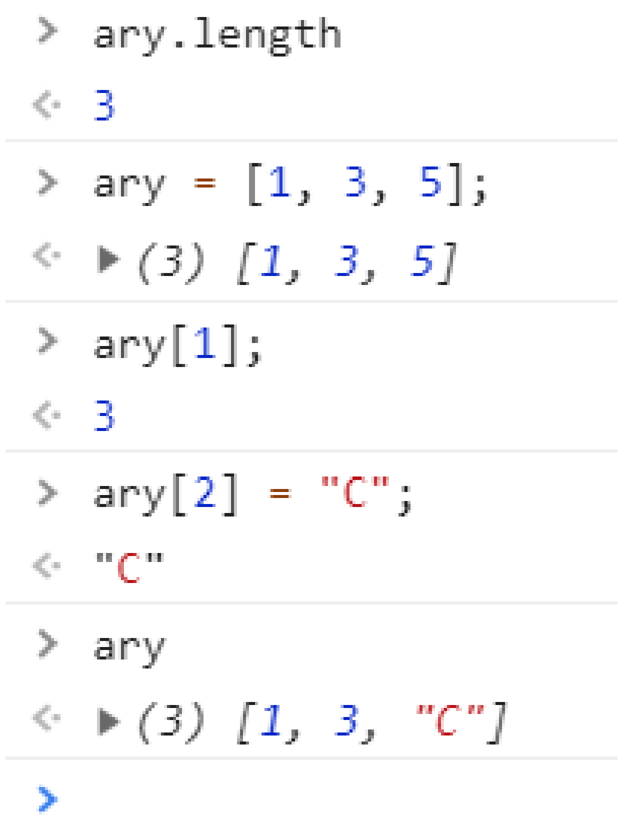


```
> var alist = new Array(1, 2, 3, 4, 5);
<· undefined
> alist
<· ▶ (5) [1, 2, 3, 4, 5]
> alist.toString()
<· "1,2,3,4,5"
> alist.sort();
<· ▶ (5) [1, 2, 3, 4, 5]
> alist.reverse();
<· ▶ (5) [5, 4, 3, 2, 1]
>
```

Setting or Changing Elements in an Array

Demo Using Chrome Console:

1. JavaScript Array can be homogenous/heterogenous
2. **ary.length** is a instance variable for the array length.



```
> ary.length
<· 3
> ary = [1, 3, 5];
<· ▶(3) [1, 3, 5]
> ary[1];
<· 3
> ary[2] = "C";
<· "C"
> ary
<· ▶(3) [1, 3, "C"]
>
```

JavaScript Array is an Object

SECTION 1

JavaScript Array is of Object Type

- The `Array` built-in object can be used to construct objects with special properties and that inherit various methods

```
var ary1 = new Array();
```

Array Constructor
No arguments

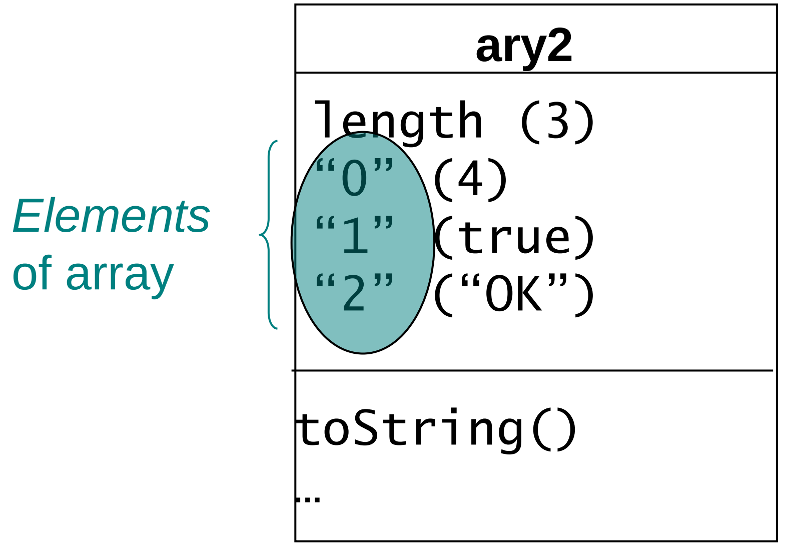
ary1	
`length (0)`	Properties

`toString()`
`sort()`
`shift()`
…
	Inherited methods

JavaScript Arrays

- The `Array` built-in object can be used to construct objects with special properties and that inherit various methods

```
var ary2 = new Array(4, true, "OK");
```



Accessing array elements:

- ✓ `ary2[1]`
- ✓ `ary2["1"]`
- ✗ `ary2.1`

Must follow identifier syntax rules

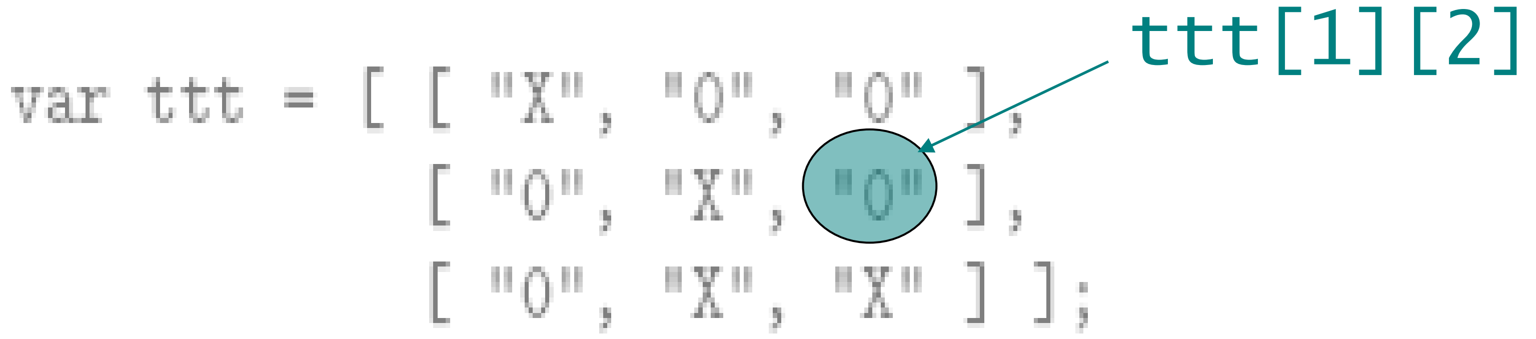
JavaScript Arrays

- The `Array` constructor is indirectly called if an array initializer is used

```
var ary2 = new Array(4, true, "OK");
```

```
var ary3 = [4, true, "OK"];
```

- Array initializiers can be used to create multidimensional arrays

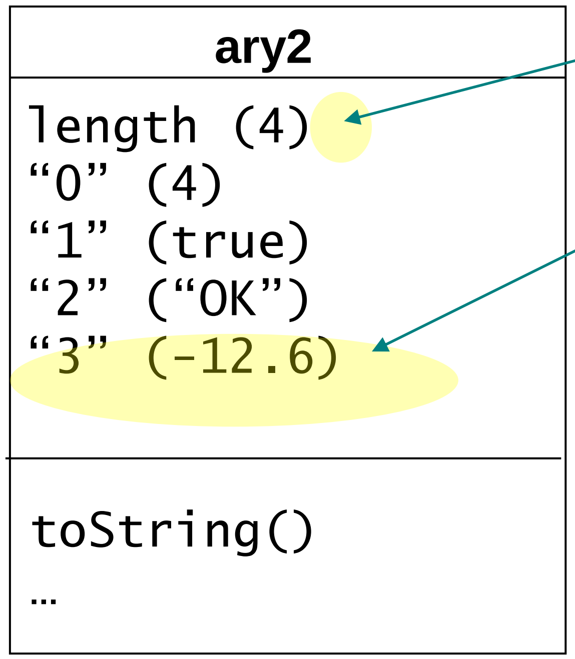


```
var ttt = [ [ "X", "O", "O" ],
            [ "O", "X", "O" ],
            [ "O", "X", "X" ] ];
```

JavaScript Arrays

```
var ary2 = new Array(4, true, "OK");
ary2[3] = -12.6;
```

Creates a new element dynamically, increases value of `length`

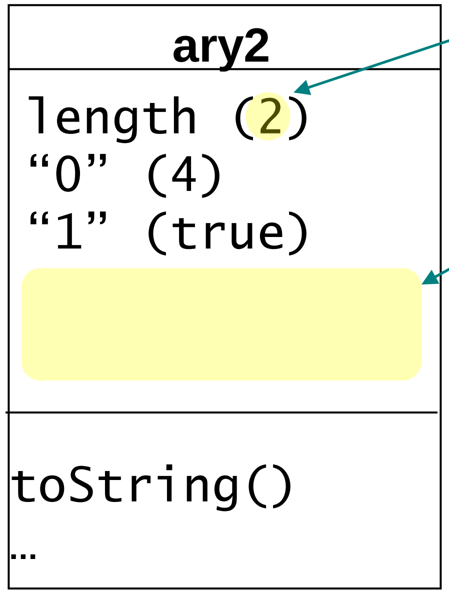


JavaScript Arrays

- Changing the number of elements:

```
var ary2 = new Array(4, true, "OK");
ary2[3] = -12.6;
ary2.length = 2;
```

Decreasing length can delete elements

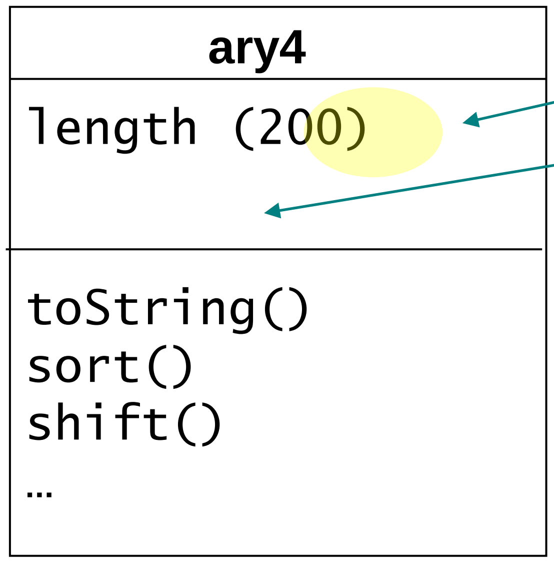


JavaScript Arrays

- Value of `length` is not necessarily the same as the actual number of elements

```
var ary4 = new Array(200);
```

Calling constructor with single argument sets `length`, does not create elements



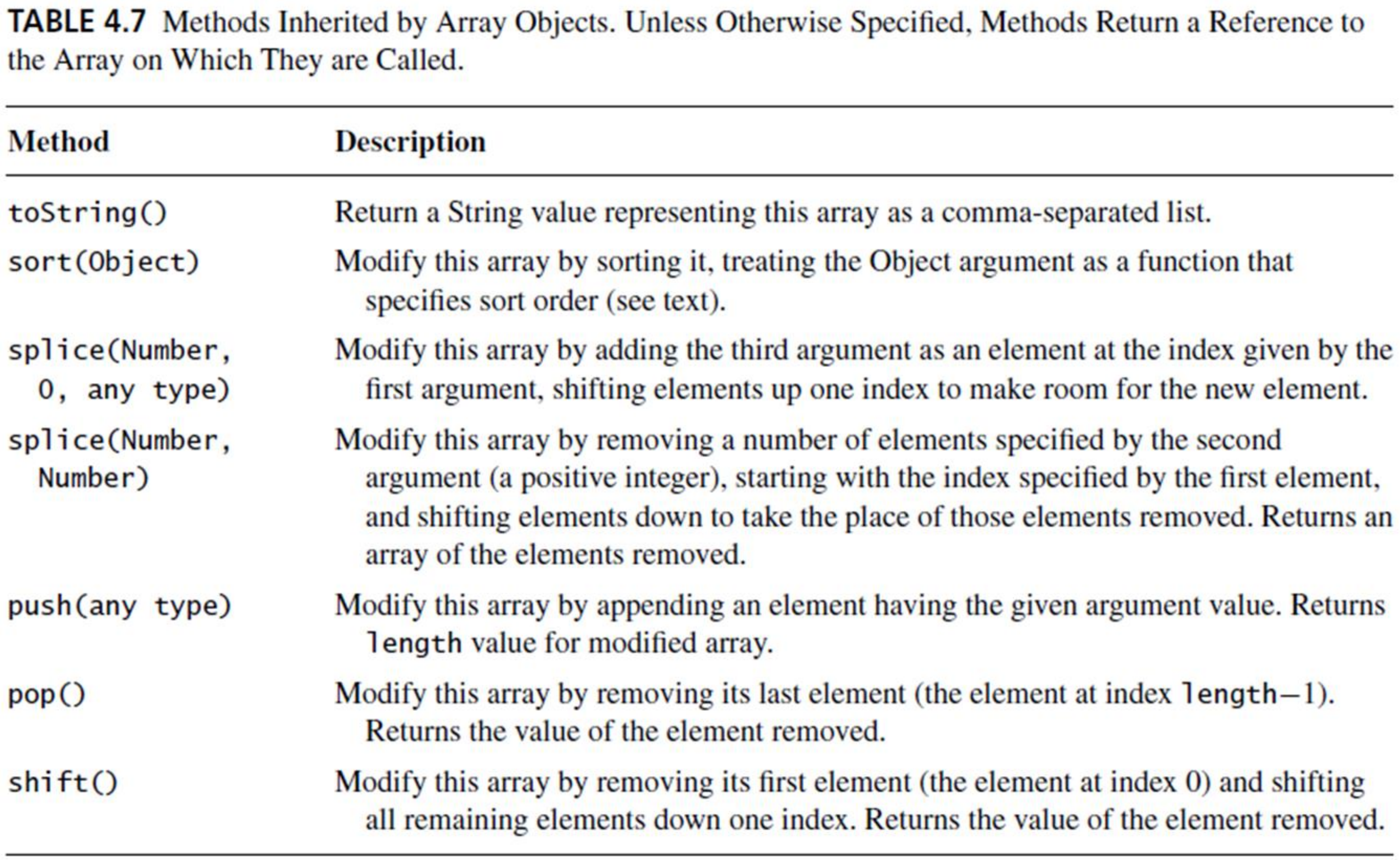
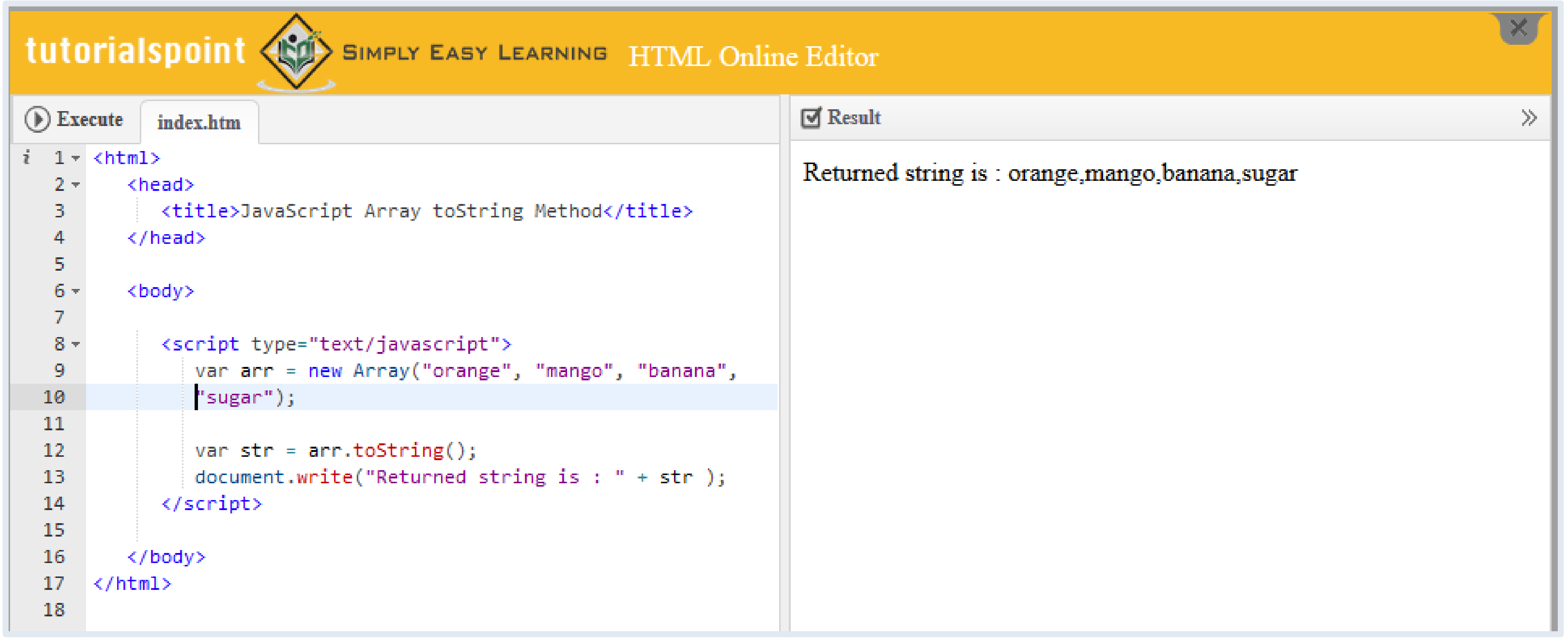


TABLE 4.7 Methods Inherited by Array Objects. Unless Otherwise Specified, Methods Return a Reference to the Array on Which They are Called.

Method	Description
`toString()`	Return a String value representing this array as a comma-separated list.
`sort(Object)`	Modify this array by sorting it, treating the Object argument as a function that specifies sort order (see text).
`splice(Number, 0, any type)`	Modify this array by adding the third argument as an element at the index given by the first argument, shifting elements up one index to make room for the new element.
`splice(Number, Number)`	Modify this array by removing a number of elements specified by the second argument (a positive integer), starting with the index specified by the first element, and shifting elements down to take the place of those elements removed. Returns an array of the elements removed.
`push(any type)`	Modify this array by appending an element having the given argument value. Returns `length` value for modified array.
`pop()`	Modify this array by removing its last element (the element at index `length`−1). Returns the value of the element removed.
`shift()`	Modify this array by removing its first element (the element at index 0) and shifting all remaining elements down one index. Returns the value of the element removed.

toString(): comma-separated format

HTML online Editor

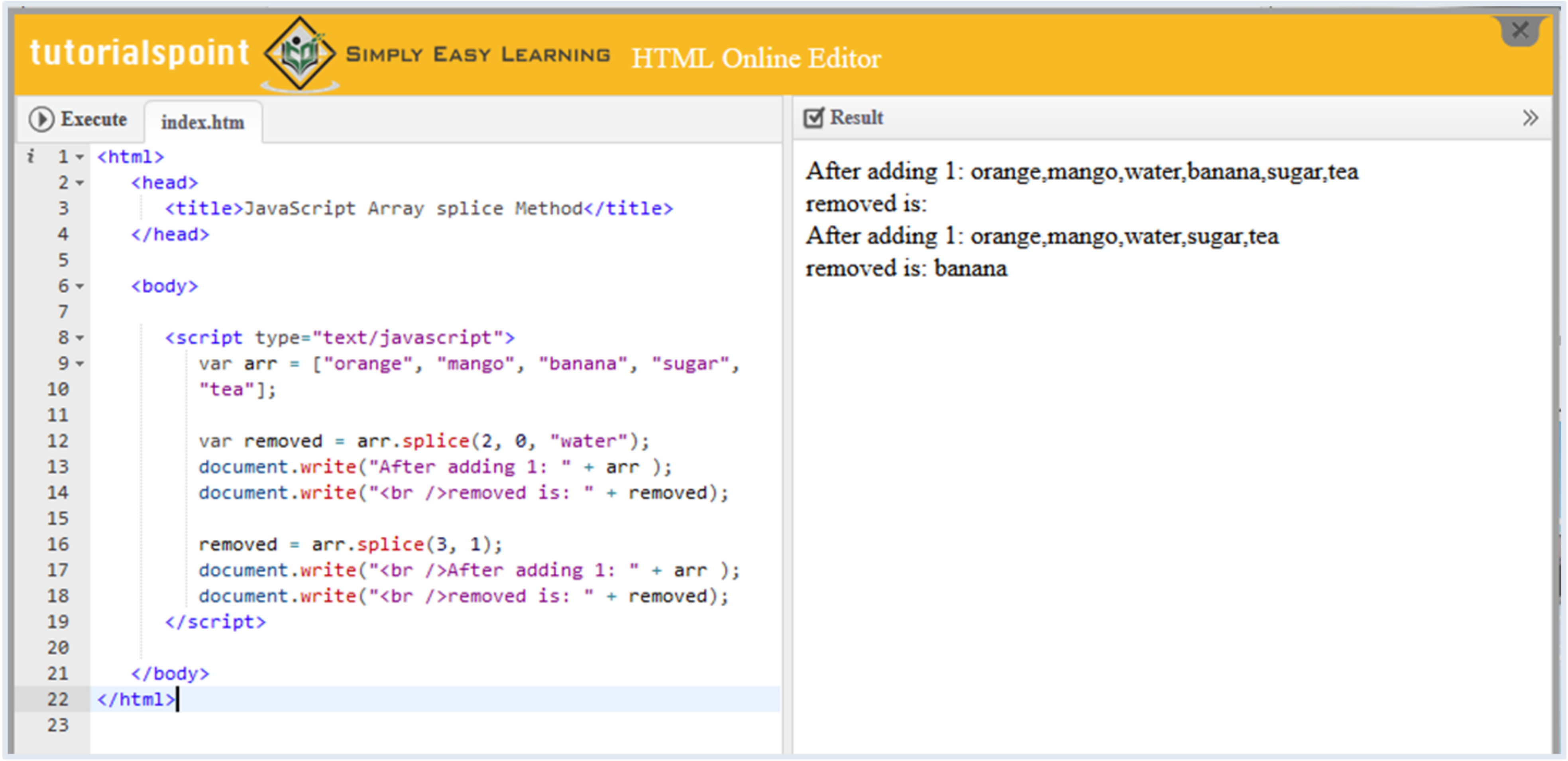


```
<html>
    <head>
        <title>JavaScript Array toString Method</title>
    </head>

    <body>

        <script type="text/javascript">
            var arr = new Array("orange", "mango", "banana",
            "sugar");

            var str = arr.toString();
            document.write("Returned string is : " + str );
        </script>

    </body>
</html>

```

```
Returned string is : orange,mango,banana,sugar
```

splice():

HTML online Editor

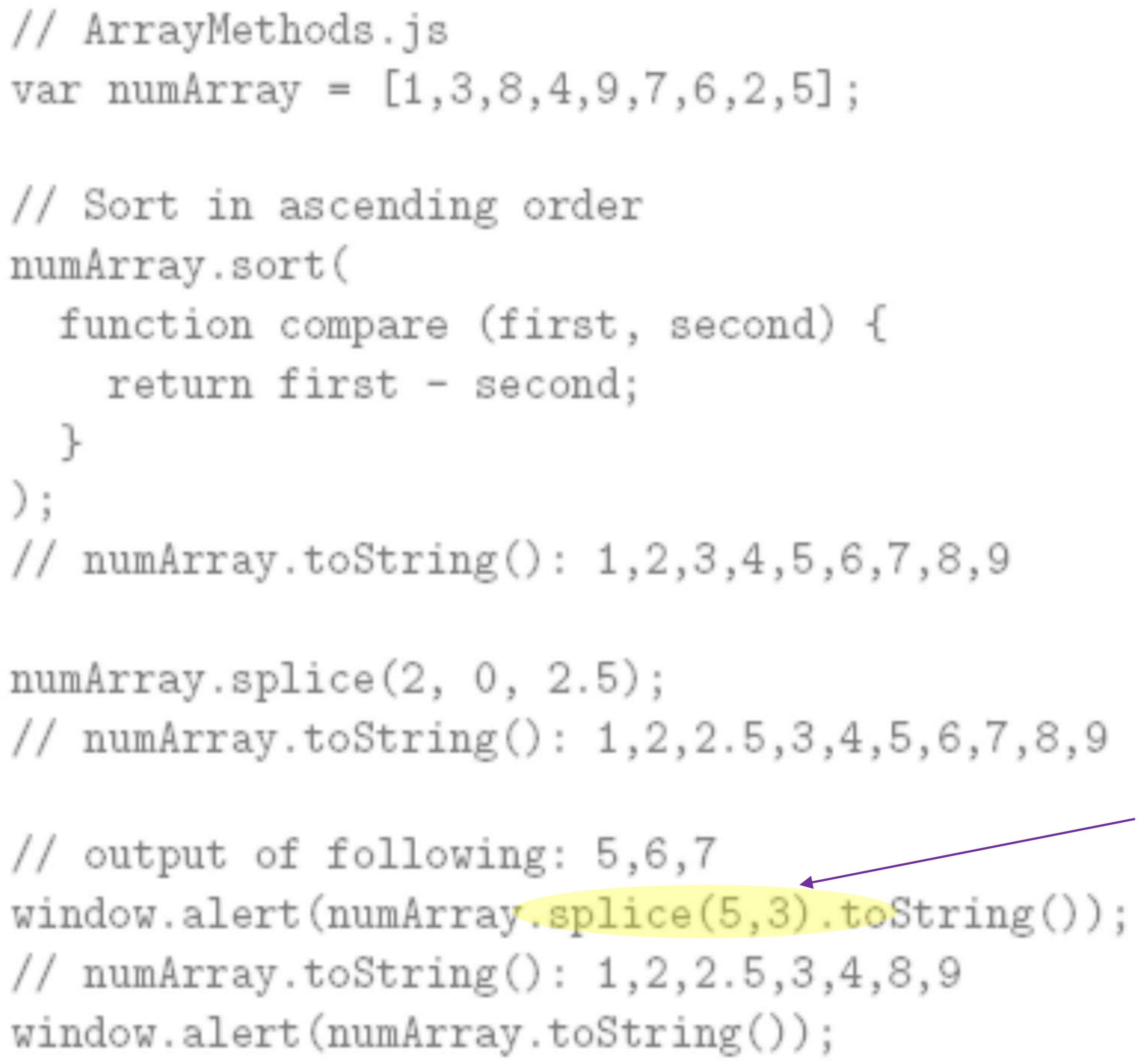


```
<html>
   <head>
      <title>JavaScript Array splice Method</title>
   </head>

   <body>

      <script type="text/javascript">
         var arr = ["orange", "mango", "banana", "sugar",
         "tea"];

         var removed = arr.splice(2, 0, "water");
         document.write("After adding 1: " + arr );
         document.write("<br />removed is: " + removed);

         removed = arr.splice(3, 1);
         document.write("<br />After adding 1: " + arr );
         document.write("<br />removed is: " + removed);
      </script>

   </body>
</html>
```

```
After adding 1: orange,mango,water,banana,sugar,tea
removed is:
After adding 1: orange,mango,water,sugar,tea
removed is: banana
```

```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Add element with value 2.5 at index 2, shift existing elements



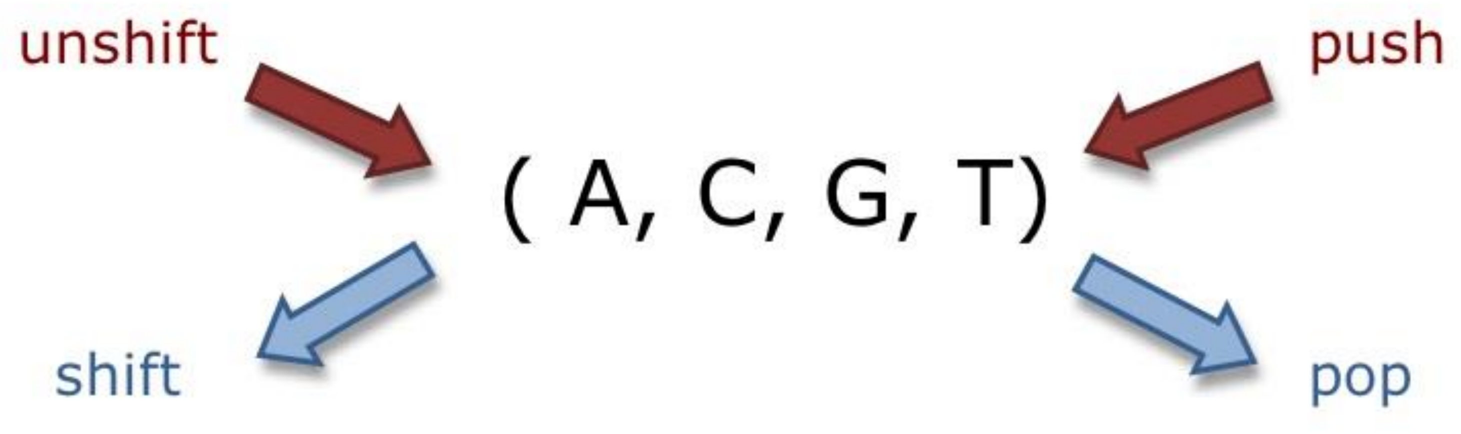
```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Remove 3 elements starting at index 5

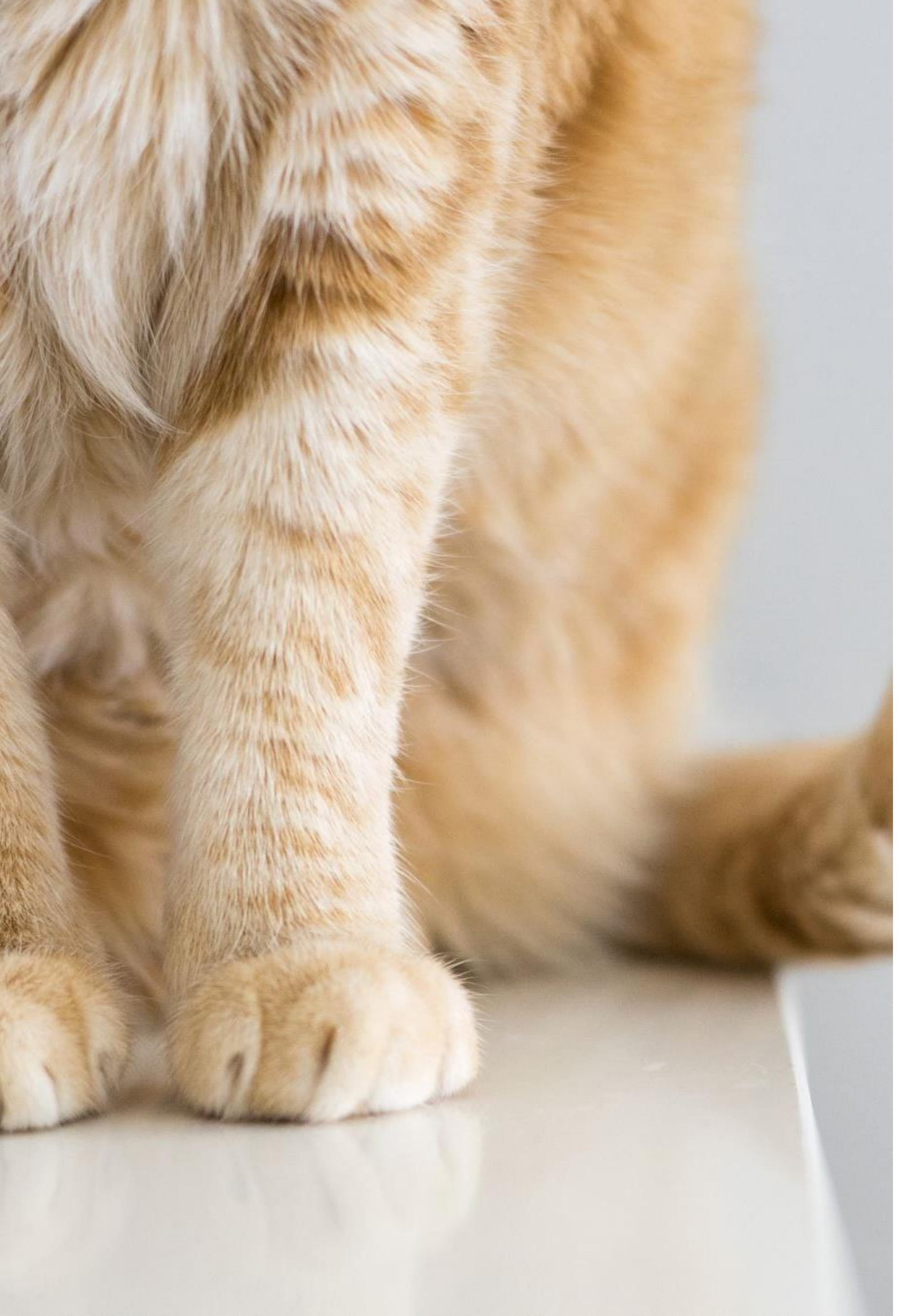
JavaScript Array is a List

SECTION 1

JavaScript Array has List Operations



- Use `unshift`. It's like `push`, except it adds elements to the beginning of the array instead of the end.
  - `unshift/push` - add an element to the beginning/end of an array
  - `shift/pop` - remove and return the first/last element of an array



push() as add()

- To add an element to the end of an array, you can use the push method. Add .push to the array name, followed by the element you want to add inside parentheses, like this:

```
> var animals = [];
> animals.push("Cat");
1
> animals.push("Dog");
2
> animals.push("Llama");
3
> animals;
["Cat", "Dog", "Llama"]
> animals.length;
3
```

unshift as add(0, data)

- The act of running a method in computer-speak is known as ***calling*** the method. When you call the push method, two things happen.
- First, the element in parentheses is added to the array.
- Second, the new length of the array is returned. That’s why you see those numbers printed out every time you call push.

unshift as add(0, data)

To add an element to the beginning of an array, you can use .unshift(*element*), like this:

```
> animals;
["Cat", "Dog", "Llama"]
> animals[0];  //❶
"Cat"
> animals.unshift("Monkey");
4
> animals;
["Monkey", "Cat", "Dog", "Llama"]
> animals.unshift("Polar Bear");
5
> animals;
["Polar Bear", "Monkey", "Cat", "Dog",
"Llama"]
>animals[0];
"Polar Bear"
> animals [2];          // ❷
"Cat"
```

pop() as remove()

- To remove the last element from an array, you can pop it off by **adding.pop()** to the end of the array name.
- The **pop** method can be particularly handy because it does two things: it removes the last element, *and* it returns that last element as a value.
- For example, let’s start with our animals array, ["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]. Then we’ll create a new variable called **lastAnimal** and save the last animal into it by calling **animals.pop()**.

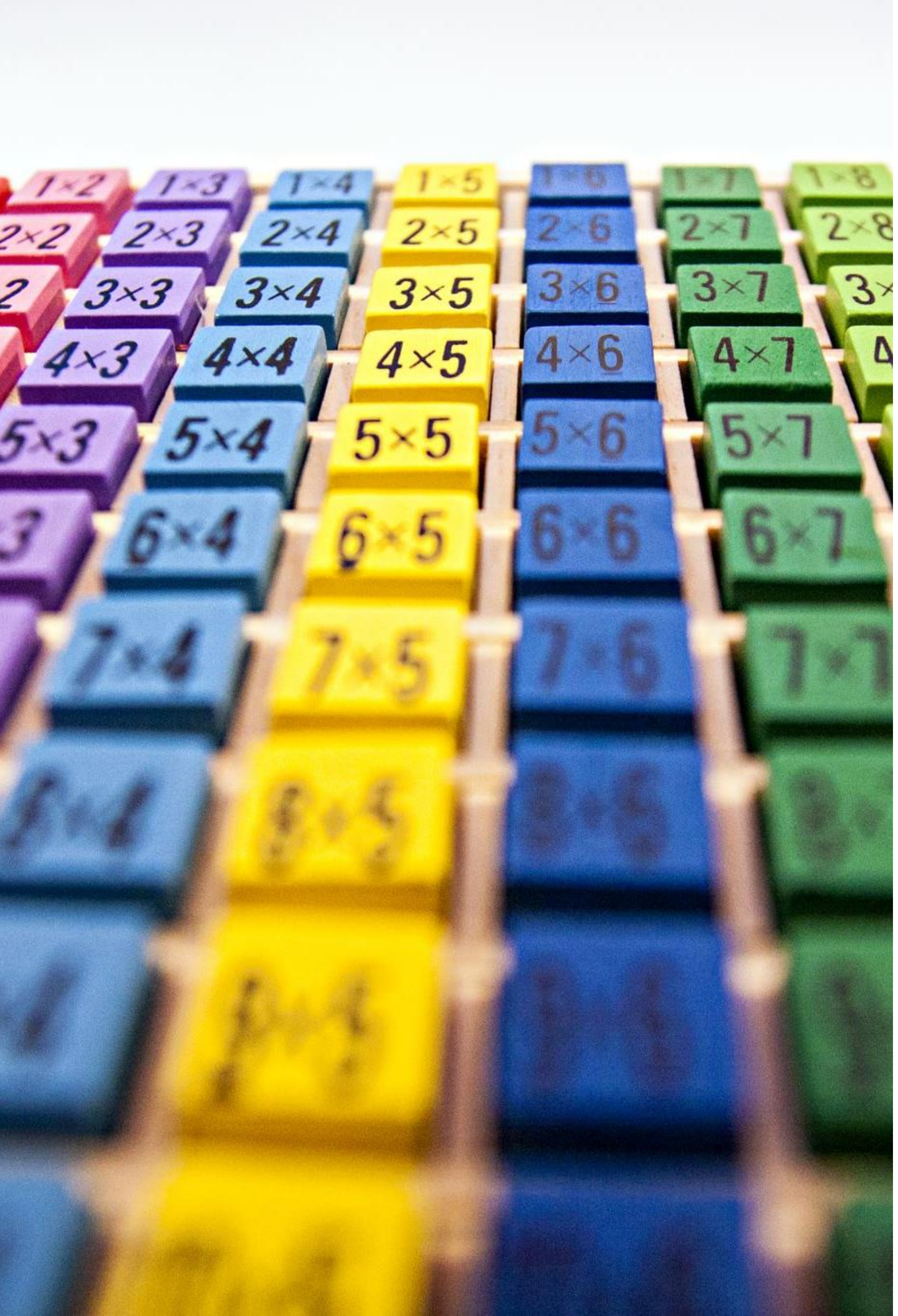
pop() as remove()

```
> animals;
["Polar Bear", "Monkey", "Cat", "Dog",
"Llama"]
> var lastAnimal = animals.pop();
lastAnimal;  // ❶
"Llama"
> animals;
["Polar Bear", "Monkey", "Cat", "Dog"]
> animals.pop();  // ❷
"Dog"
> animals;
["Polar Bear", "Monkey", "Cat"]
> animals.unshift(lastAnimal); // ❸
4
> animals;
["Llama", "Polar Bear", "Monkey", "Cat"]
```

shift() as remove(0)

To remove and return the first element of an array, use .shift():

```
> animals;
["Llama", "Polar Bear", "Monkey", "Cat"]
> var firstAnimal = animals.shift();
> firstAnimal;
"Llama"
> animals;
["Polar Bear", "Monkey", "Cat"]
```



JavaScript Array for all Data Collections

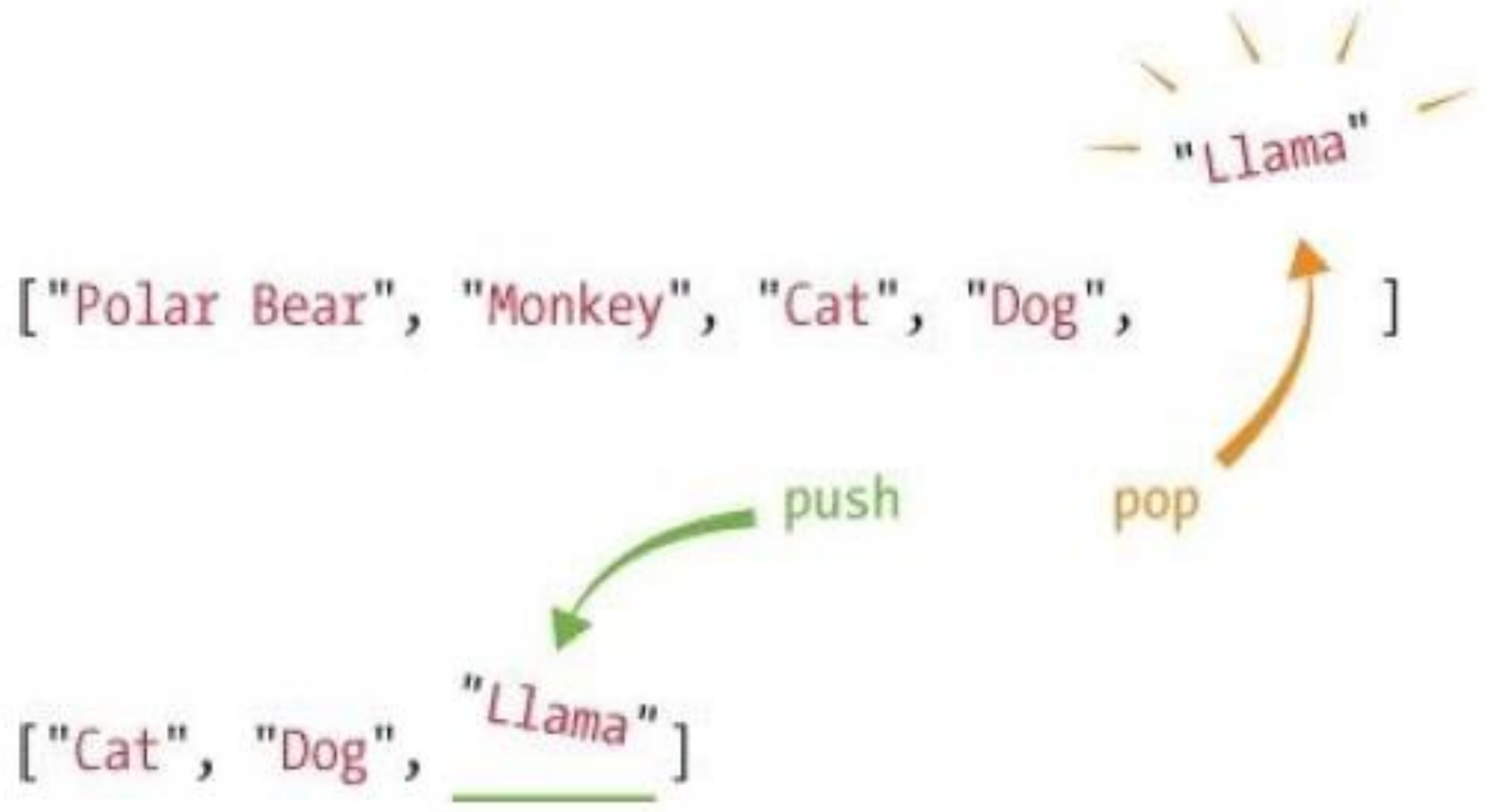
Stack: push() and pop()

Queue: unshift() and pop(), or shift() and push()

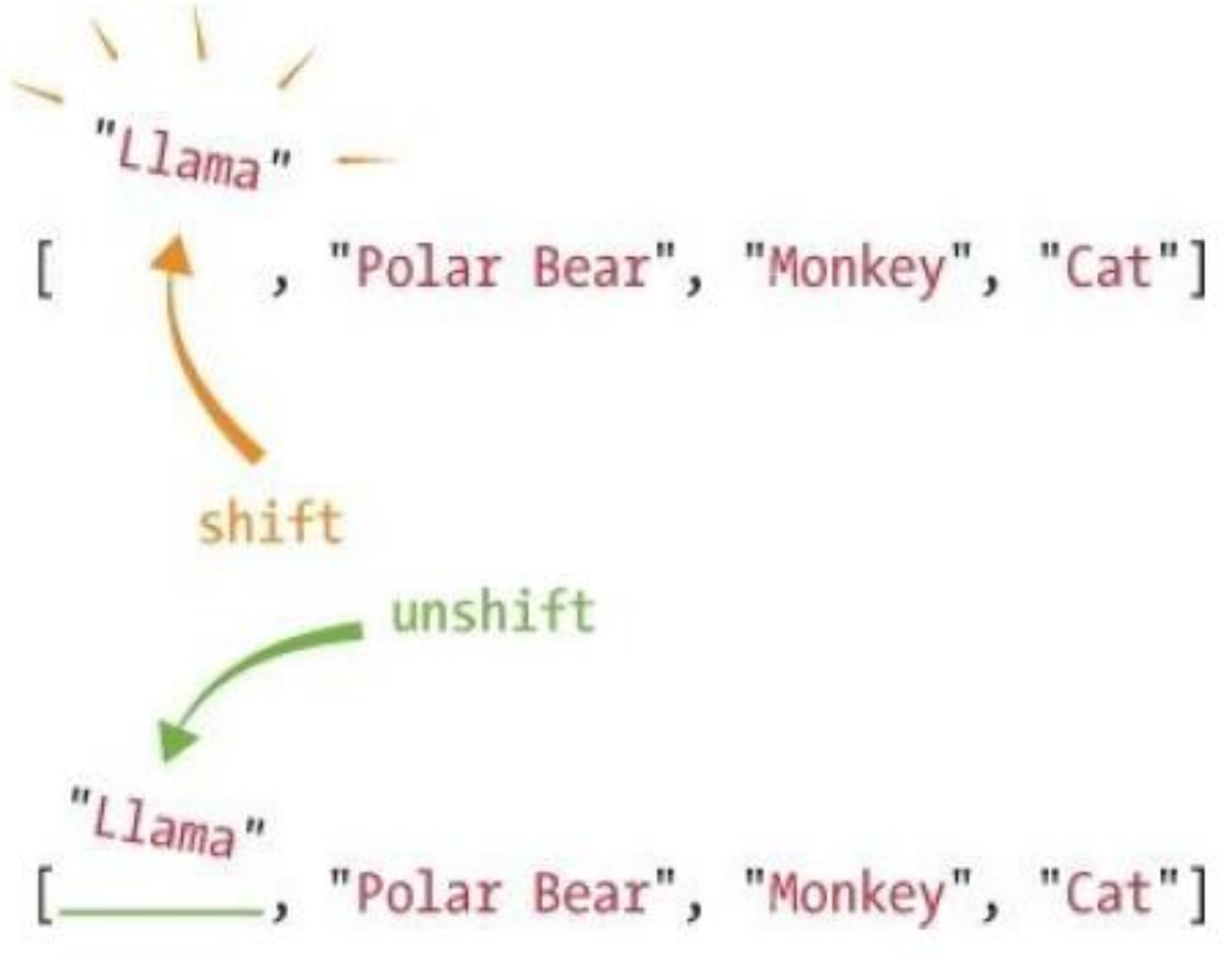
Deque: push()/unshift() and pop()/shift()

Stack Operations

Pushing and popping are a useful pair because sometimes you care about only the end of an array. You can push a new item onto the array and then pop it off when you’re ready to use it. We’ll look at some ways to use pushing and popping later in this chapter.



Stack Operations


"Llama"
[, "Polar Bear", "Monkey", "Cat"]
shift
unshift
"Llama"
[____, "Polar Bear", "Monkey", "Cat"]

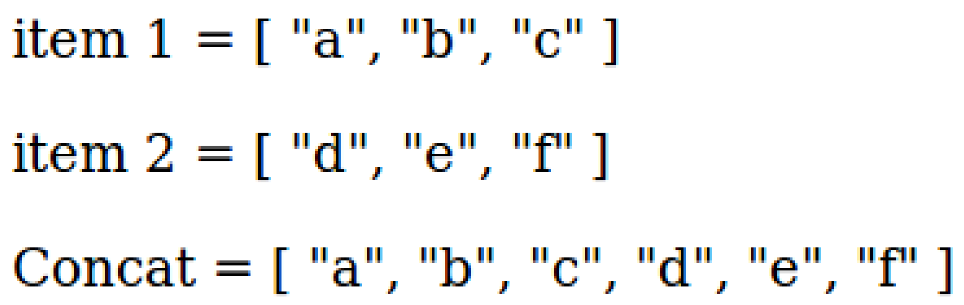
Stack Operations

Adding Arrays

SECTION 1

Adding Arrays

- ary1.concat(ary2): adding array ary2 to ary1.



item 1 = ["a", "b", "c"]

item 2 = ["d", "e", "f"]

Concat = ["a", "b", "c", "d", "e", "f"]

Joining Multiple Arrays

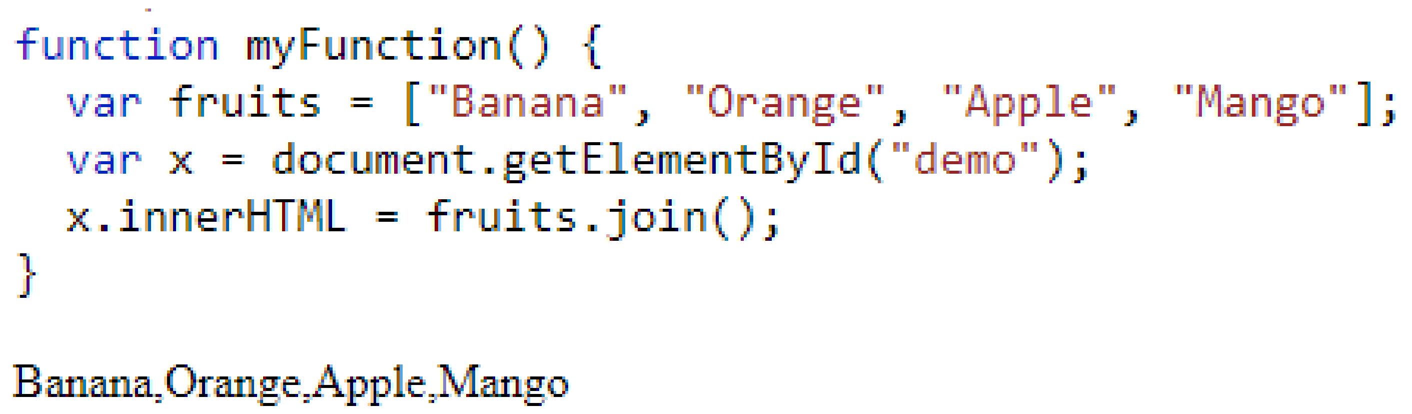
- You can use concat to join more than two arrays together. Just put the extra arrays inside the parentheses, separated by commas:

```
var furryAnimals = ["Alpaca", "Ring-tailed Lemur",
"Yeti"];
var scalyAnimals = ["Boa Constrictor", "Godzilla"];
var featheredAnimals = ["Macaw", "Dodo"];
var allAnimals = furryAnimals.concat(scalyAnimals,
featheredAnimals);

> allAnimals;
["Alpaca", "Ring-tailed Lemur", "Yeti", "Boa
Constrictor", "Godzilla", "Macaw", "Dodo"]
```

- Here the values from featheredAnimals get added to the very end of the new array, since they are listed last in the parentheses after the concat method.

Join Arrays into String

- ary.join("Separators"): adding all arrays into one string.



```
function myFunction() {
  var fruits = ["Banana", "Orange", "Apple", "Mango"];
  var x = document.getElementById("demo");
  x.innerHTML = fruits.join();
}
```

```
Banana,Orange,Apple,Mango
```

Turning an Array into a String

- You can use .join() to join all the elements in an array together into one big string.

```
var boringAnimals = ["Monkey", "Cat", "Fish", "Lizard"];
> boringAnimals.join();
"Monkey,Cat,Fish,Lizard"
```

Turning an Array into a String with Separator

- You can use .join(*separator*) to do the same thing, but with your own chosen separator between each value.
- The separator is whatever string you put inside the parentheses. For example, we can use three different separators: a hyphen with spaces on either side, an asterisk, and the word *sees* with spaces on either side.
- Notice that you need quote marks around the separator, because the separator is a string.

```
var boringAnimals = ["Monkey", "Cat", "Fish", "Lizard"];
> boringAnimals.join(" - ");
"Monkey - Cat - Fish - Lizard"
> boringAnimals.join("*")
"Monkey*Cat*Fish*Lizard"
> boringAnimals.join(" sees ")
"Monkey sees Cat sees Fish sees Lizard"
```

Turning an Array into a String

- This is useful if you have an array that you want to turn into a string. Say you have lots of middle names and you've got them stored in an array, along with your first and last name. You might be asked to give your full name as a string. Using join, with a single space as the separator, will join all your names together into a single string:

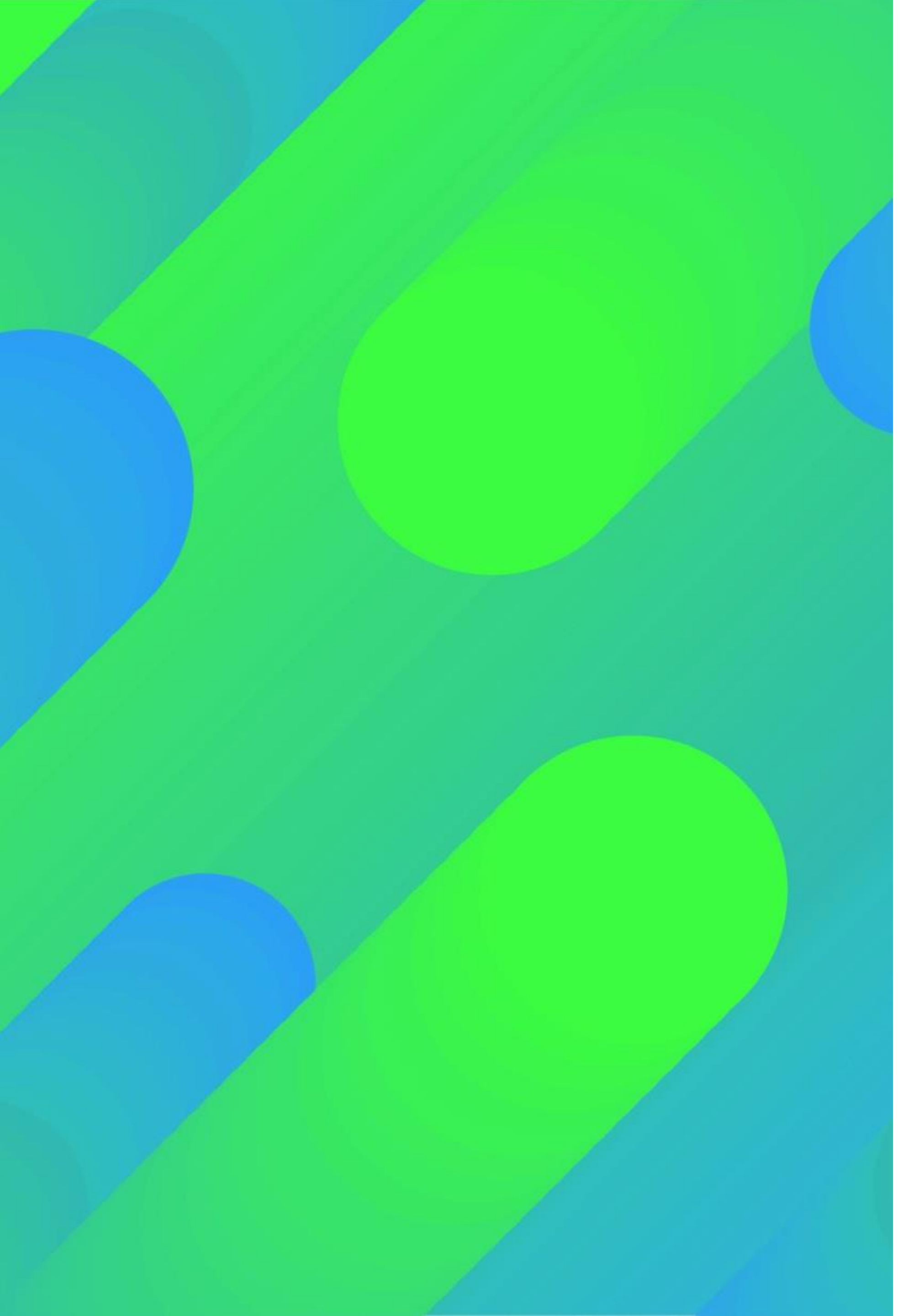
```
var myNames = ["Nicholas", "Andrew", "Maxwell", "Morgan"];
myNames.join(" ");
"Nicholas Andrew Maxwell Morgan"
```

- If you didn't have join, you'd have to do something like this, which would be really annoying to type out:

```
myNames[0] + " " + myNames[1] + " " + myNames[2] + " " + myNames[3];
"Nicholas Andrew Maxwell Morgan"
```

Searching Arrays

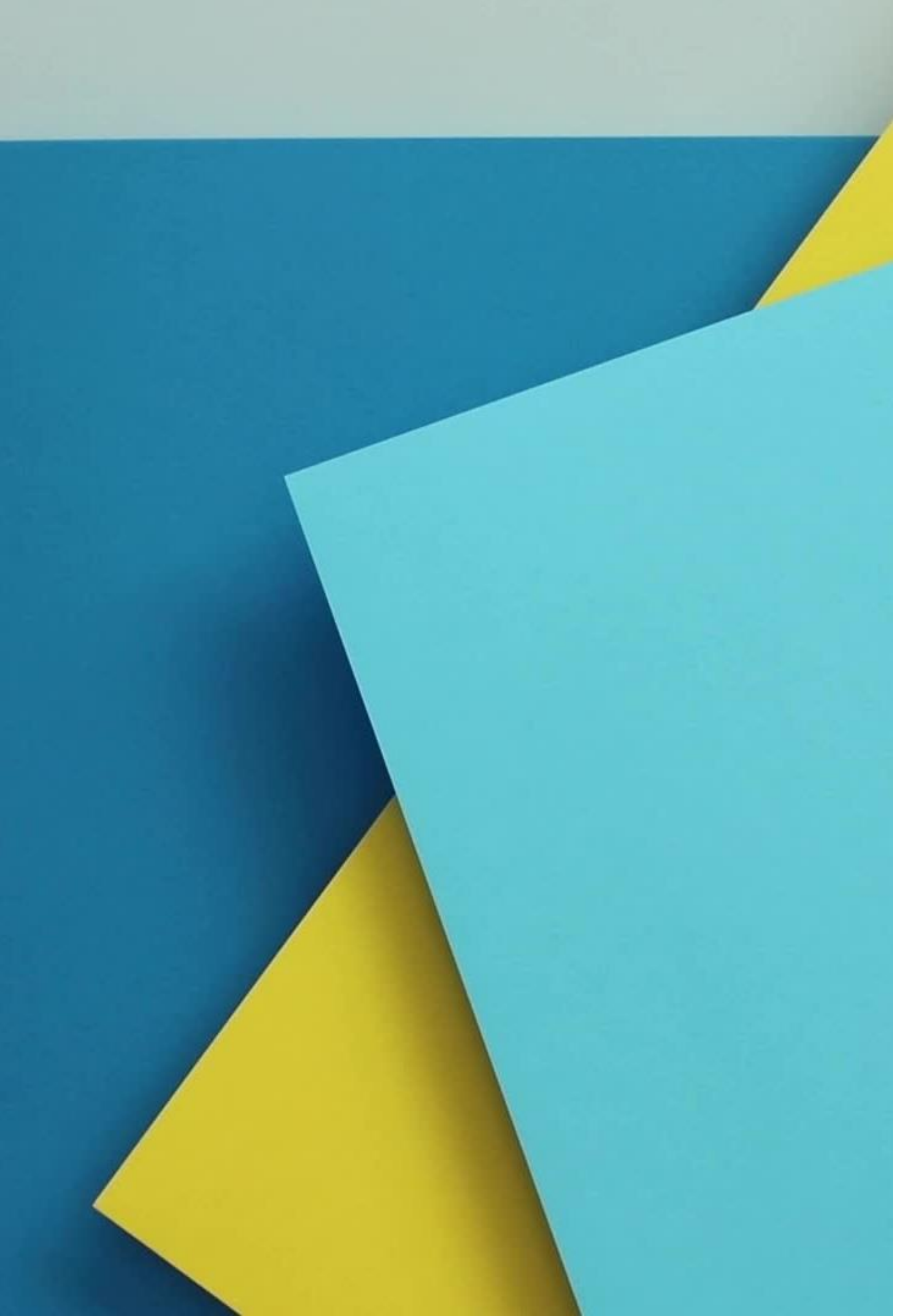
SECTION 1



indexOf()
Finding the Index of an Element in an Array

- To find the index of an element in an array, use .indexOf(*element*). Here we define the array colors and then ask for the index positions of "blue" and "green" with colors.indexOf("blue") and colors.indexOf("green").
- Because the index of "blue" in the array is 2, colors.indexOf("blue") returns 2. The index of "green" in the array is 1, so colors.indexOf("green") returns 1.

```
var colors = ["red", "green",
"blue"];
> colors.indexOf("blue");
2
> colors.indexOf("green");
1
```



indexOf()
Finding the Index of an Element in an Array

- indexOf is like the reverse of using square brackets to get a value at a particular index; colors[2] is "blue", so colors.indexOf("blue") is 2:\

```
> colors[2];
"blue"
> colors.indexOf("blue");
2
```

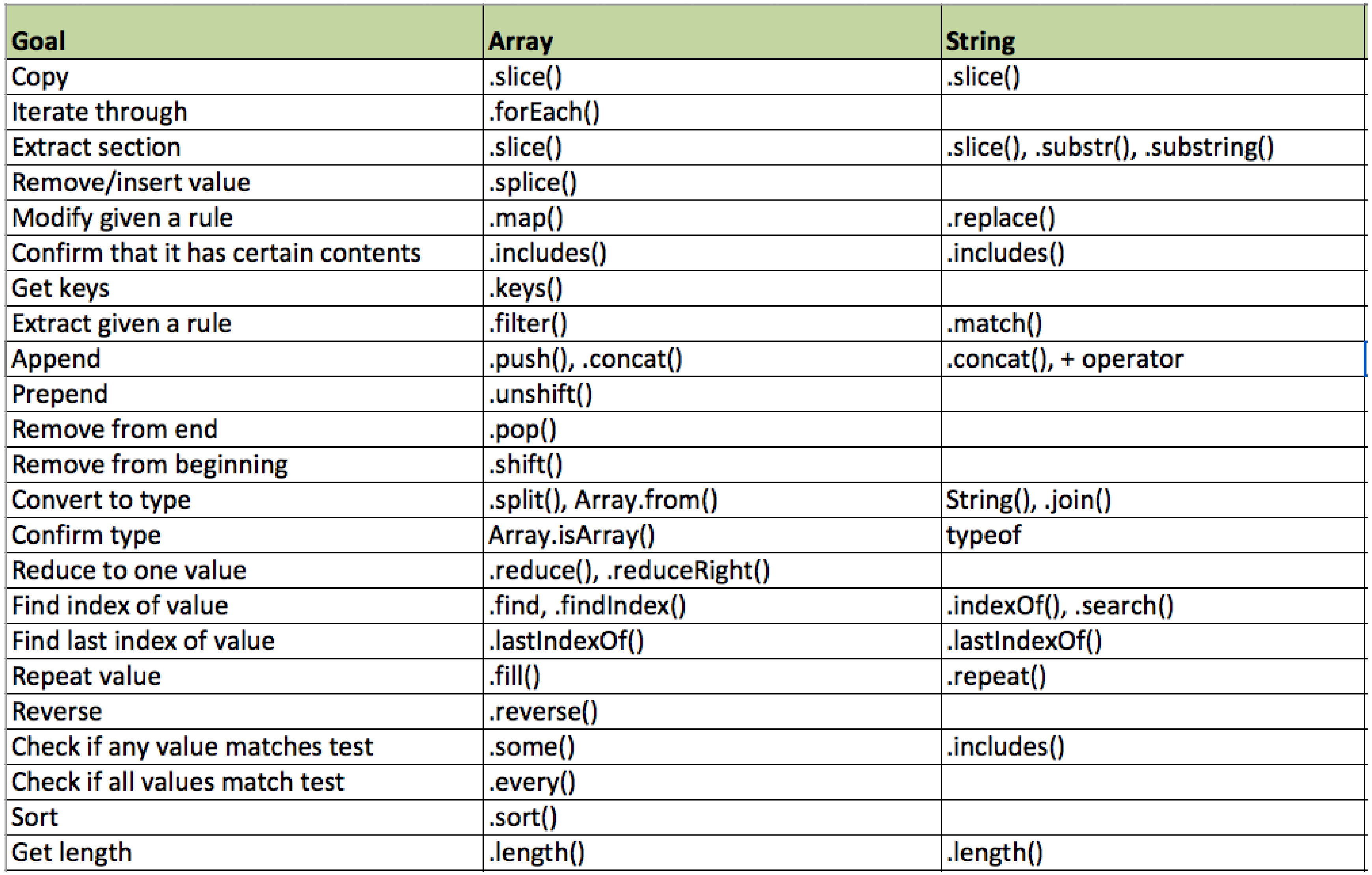
indexOf() Finding the Index of an Element in an Array

- Even though "blue" appears third in the array, its index position is 2 because we always start counting from 0. And the same goes for "green", of course, at index 1.
- If the element whose position you ask for is not in the array, JavaScript returns -1.

```
> colors.indexOf("purple");
-1
```

- This is JavaScript’s way of saying “That doesn’t exist here,” while still returning a number.
- If the element appears more than once in the array, the indexOf method will return the first index of that element in the array.

```
var insects = ["Bee", "Ant", "Bee", "Bee",
"Ant"];
> insects.indexOf("Bee");
0
```



Goal	Array	String
Copy	.slice()	.slice()
Iterate through	.forEach()	
Extract section	.slice()	.slice(), .substr(), .substring()
Remove/insert value	.splice()	
Modify given a rule	.map()	.replace()
Confirm that it has certain contents	.includes()	.includes()
Get keys	.keys()	
Extract given a rule	.filter()	.match()
Append	.push(), .concat()	.concat(), + operator
Prepend	.unshift()	
Remove from end	.pop()	
Remove from beginning	.shift()	
Convert to type	.split(), Array.from()	String(), .join()
Confirm type	Array.isArray()	typeof
Reduce to one value	.reduce(), .reduceRight()	
Find index of value	.find, .findIndex()	.indexOf(), .search()
Find last index of value	.lastIndexOf()	.lastIndexOf()
Repeat value	.fill()	.repeat()
Reverse	.reverse()	
Check if any value matches test	.some()	.includes()
Check if all values match test	.every()	
Sort	.sort()	
Get length	.length()	.length()

Project 1: Backtracking

SECTION 1

Building the Array with Push

- Here we create an empty array named landmarks and then use push to store all the landmarks you pass on the way to your friend’s house.

```
var landmarks = [];
landmarks.push("My house");
landmarks.push("Front path");
landmarks.push("Flickering streetlamp");
landmarks.push("Leaky fire hydrant");
landmarks.push("Fire station");
landmarks.push("Cat rescue center");
landmarks.push("My old school");
landmarks.push("My friend's house");
```

Going in Reverse with pop

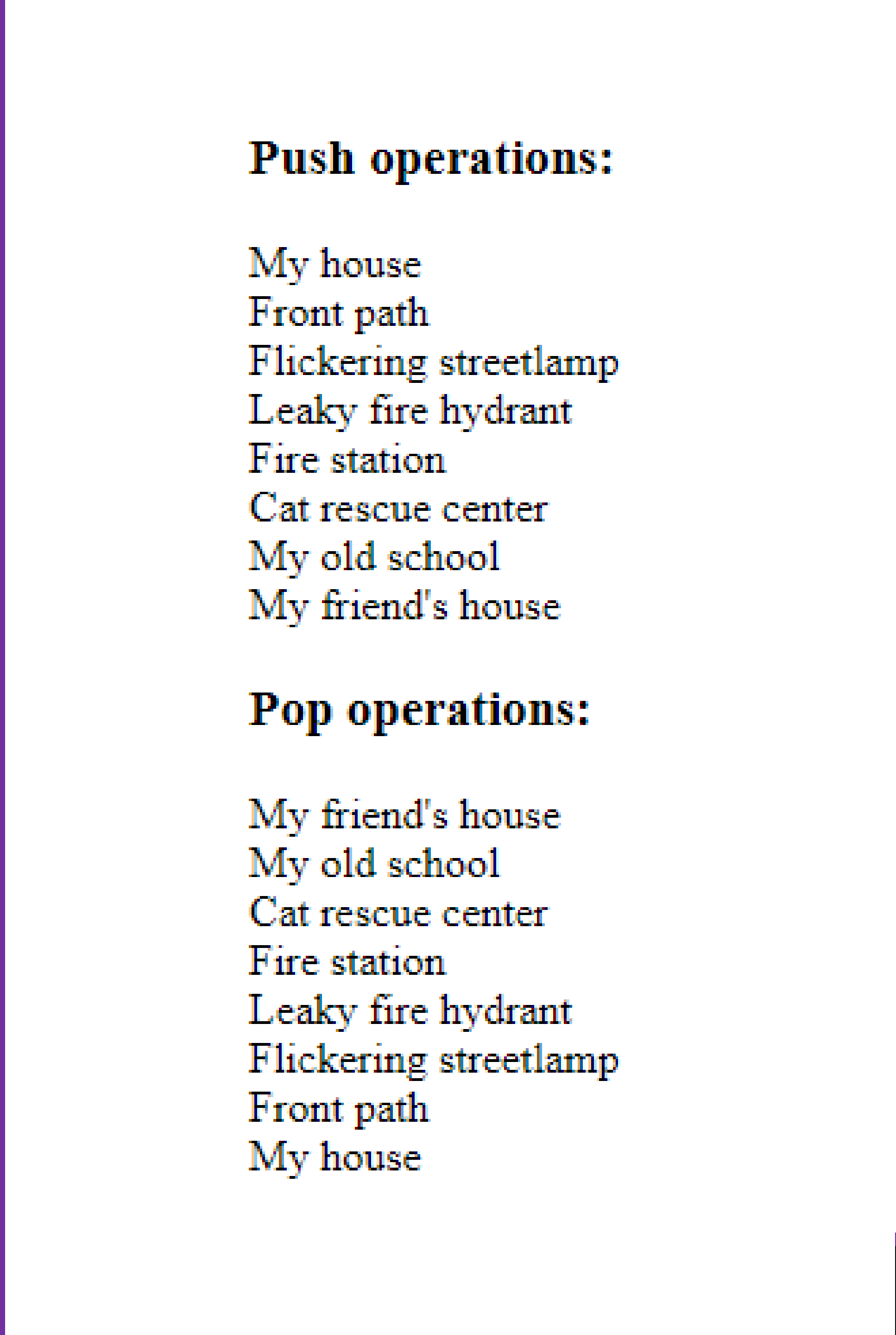
- Once you arrive at your friend’s house, you can inspect your array of landmarks.
- Sure enough, the first item is "My house", followed by "Front path", and so on through the end of the array, with the final item "My friend's house".
- When it’s time to go home, all you need to do is pop off the items one by one, and you’ll know where to go next.

Pop Operations

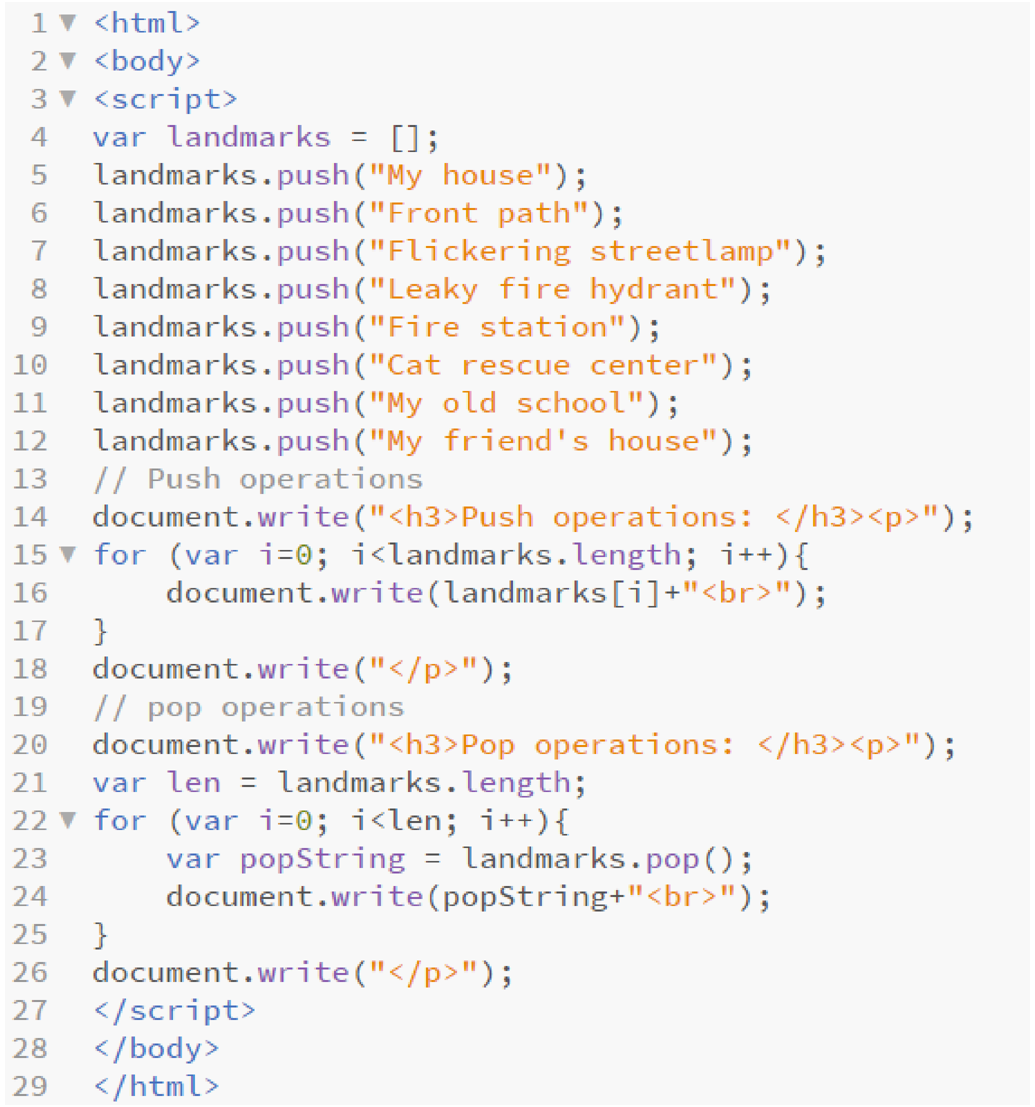
```
> landmarks.pop();
"My friend's house"
> landmarks.pop();
"My old school"
> landmarks.pop();
"Cat rescue center"
> landmarks.pop();
"Fire station"
> landmarks.pop();
"Leaky fire hydrant"
> landmarks.pop();
"Flickering streetlamp"
> landmarks.pop();
"Front path"
> landmarks.pop();
"My house"
```

Back to Home
Demo Program:
Push_Pop.html

Go Brackets!!!



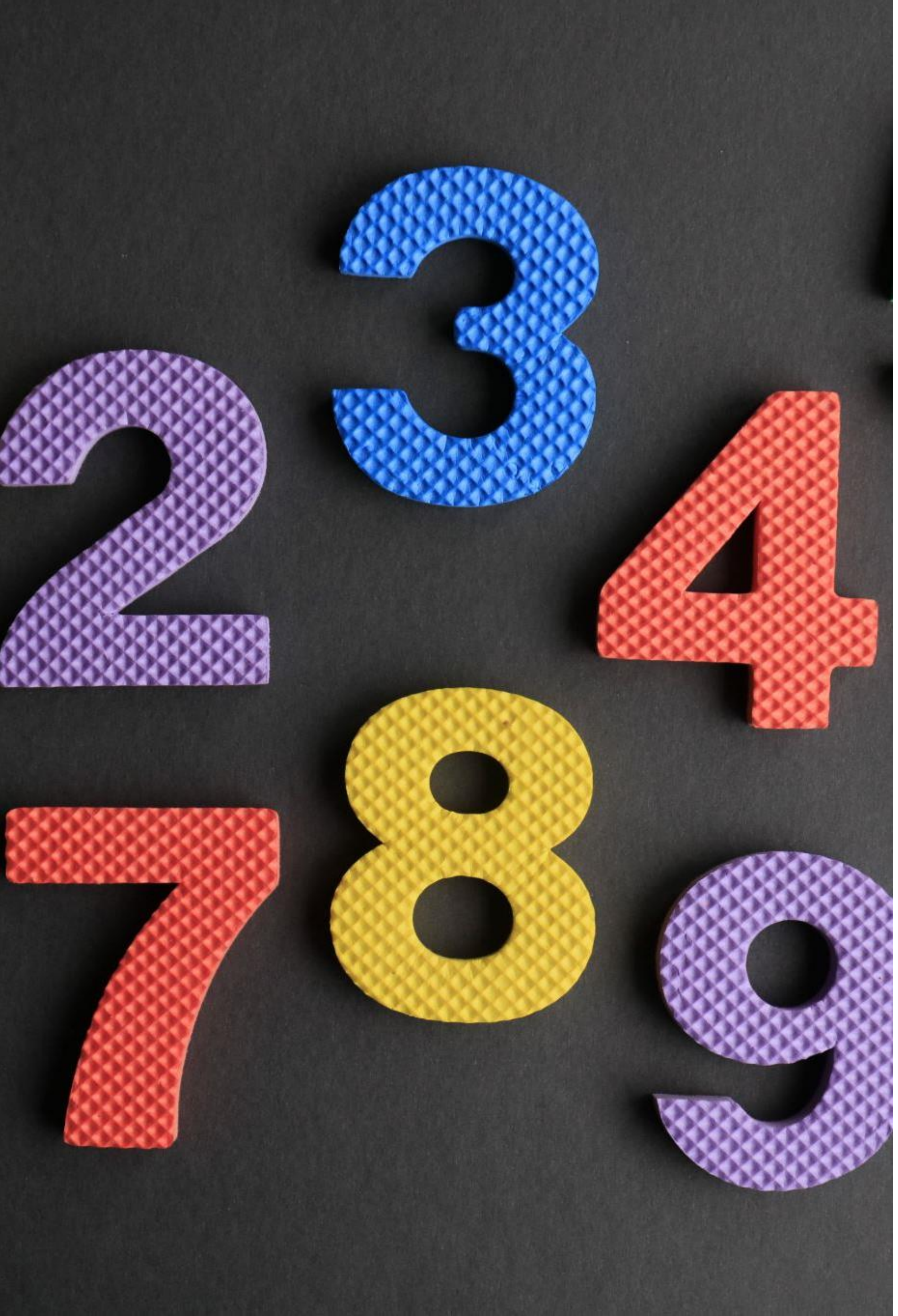
```
Push operations:

My house
Front path
Flickering streetlamp
Leaky fire hydrant
Fire station
Cat rescue center
My old school
My friend's house

Pop operations:

My friend's house
My old school
Cat rescue center
Fire station
Leaky fire hydrant
Flickering streetlamp
Front path
My house
```



```
<html>
<body>
<script>
var landmarks = [];
landmarks.push("My house");
landmarks.push("Front path");
landmarks.push("Flickering streetlamp");
landmarks.push("Leaky fire hydrant");
landmarks.push("Fire station");
landmarks.push("Cat rescue center");
landmarks.push("My old school");
landmarks.push("My friend's house");
// Push operations
document.write("<h3>Push operations: </h3><p>");
for (var i=0; i<landmarks.length; i++){
    document.write(landmarks[i]+"<br>");
}
document.write("</p>");
// pop operations
document.write("<h3>Pop operations: </h3><p>");
var len = landmarks.length;
for (var i=0; i<len; i++){
    var popString = landmarks.pop();
    document.write(popString+"<br>");
}
document.write("</p>");
</script>
</body>
</html>
```

Project 2: Decision Maker

SECTION 1



Using Math.random()

- We can produce random numbers using a special method called Math.random(), which returns a random number between 0 and 1 each time it's called. Here's an example:

```
> Math.random();
0.8945409457664937
> Math.random();
0.3697543195448816
> Math.random();
0.48314980138093233
```

Using Math.random()
Count parameter

- If you want a bigger number, just multiply the result of calling Math.random(). For example, if you wanted numbers between 0 and 10, you would multiply Math.random() by 10:

```
> Math.random() * 10;
7.648027329705656
> Math.random() * 10;
9.756590453442186l
> Math.random() * 10;
0.2148344297893345
```

Rounding Down with Math.floor()

Quantization

- We can’t use these numbers as array indexes, though, because indexes have to be whole numbers with nothing after the decimal point. To fix that, we need another method called Math.floor().
- This takes a number and rounds it down to the whole number below it (basically getting rid of everything after the decimal point).

```
> Math.floor(3.7463463);
3
> Math.floor(9.9999);
9
> Math.floor(0.79342345196342 6);
0
```

Rounding Down with Math.floor()
Similar to Java (int)(Math.random()* count)

- We can combine these two techniques to create a random index. All we need to do is multiply Math.random() by the length of the array and then call Math.floor() on that value. For example, if the length of the array were 4, we would do this:

```
> Math.floor(Math.random() * 4);
2 // could be 0, 1, 2, or 3
```

- Every time you call the code above, it returns a random number from 0 to 3 (including 0 and 3). Because Math.random() always returns a value less than 1, Math.random() * 4 will never return 4 or anything higher than 4. Now, if we use that random number as an index, we can select a random element from an array:

```
var randomWords = ["Explosion", "Cave", "Princess", "Pen"];
var randomIndex = Math.floor(Math.random() * 4);
> randomWords[randomIndex];
"Cave"
```

Rounding Down with Math.floor()

- Here we use Math.floor(Math.random() * 4); to pick a random number from 0 to 3. Once that random number is saved to the variable randomIndex, we use it as an index to ask for a string from the array randomWords.
- In fact, we could shorten this by doing away with the randomIndex variable altogether and just say:

```
> randomWords[Math.floor(Math.random() * 4)];
"Princess"
```

The Complete Decision Maker

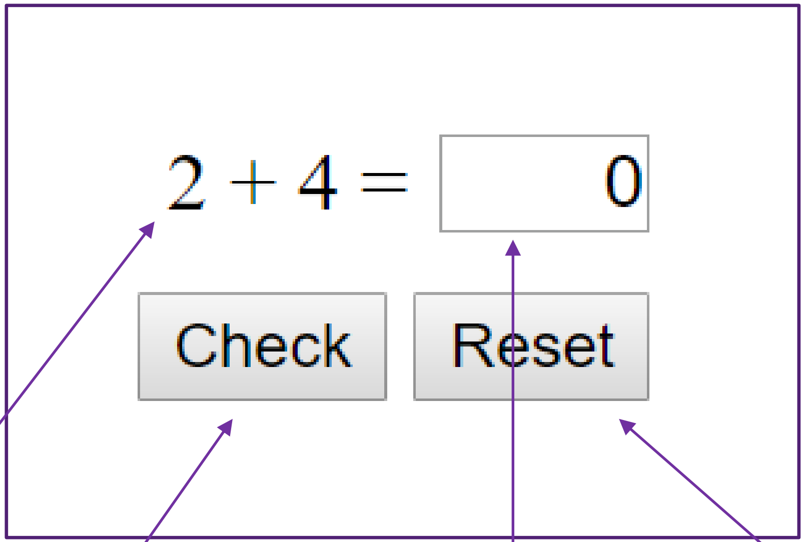
Practice with HTML online editor

```
var phrases = [
"That sounds good",
"Yes, you should definitely do that",
"I'm not sure that's a great idea",
"Maybe not today?",
"Computer says no."
];
// Should I have another milkshake?
> phrases[Math.floor(Math.random() * 5)];
"I'm not sure that's a great idea"
// Should I do my homework?
> phrases[Math.floor(Math.random() * 5)];
"Maybe not today?"
```

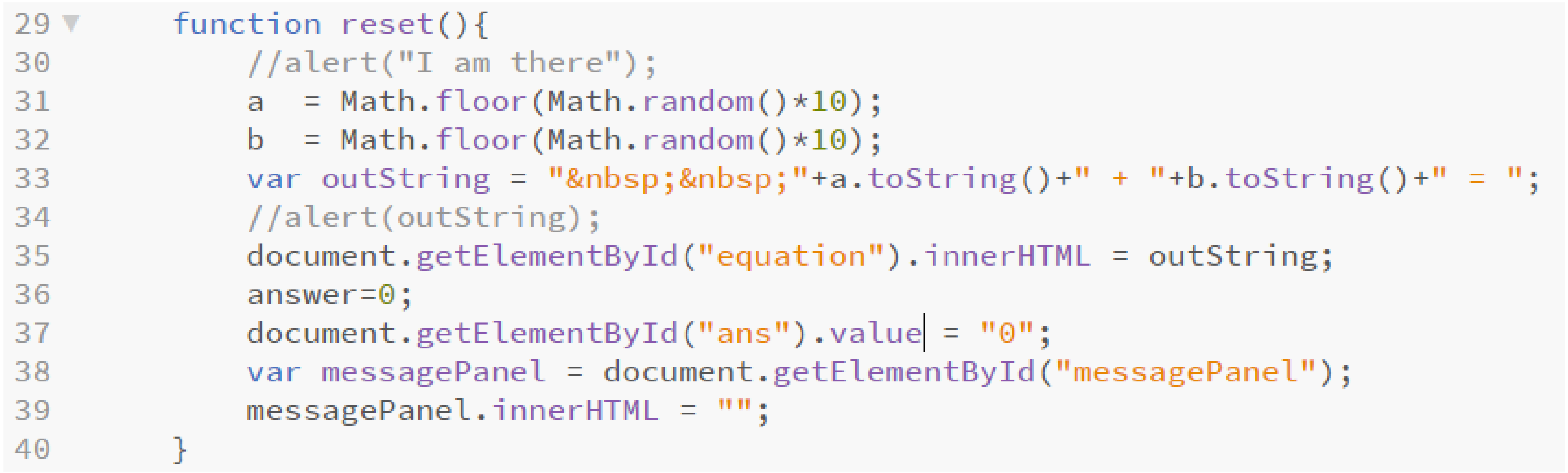
Decision Maker

Demo Program: decisionmaker.html

Go Brackets!!!

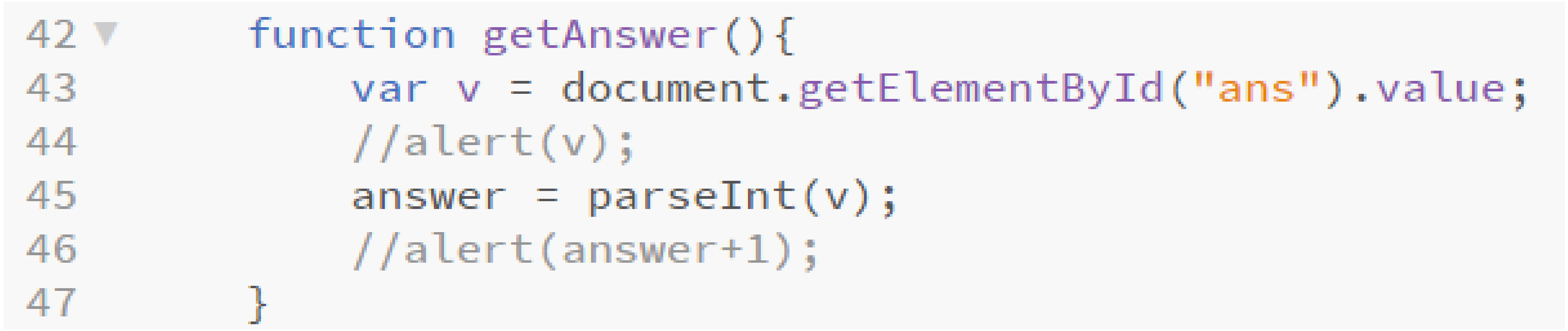


```
<body onload="reset()">
    <div>
    <label id="equation">
    </label>&nbsp;&nbsp;
    <input id="ans" onchange="getAnswer()" placeholder="0">
    </div>
    <br>
    <button onclick="checkAnswer()">&nbsp;Check&nbsp;</button><button
    onclick="reset()">&nbsp;Reset&nbsp;</button>
    <br>
    <p id="messagePanel"></p>
</body>
```



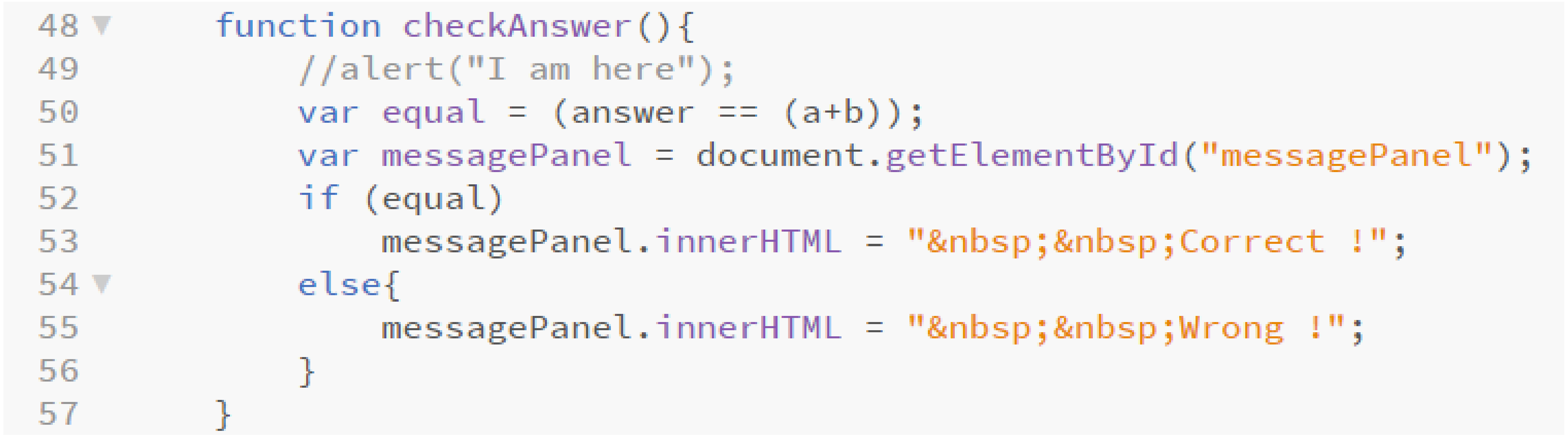
```
    function reset(){
        //alert("I am there");
        a  = Math.floor(Math.random()*10);
        b  = Math.floor(Math.random()*10);
        var outString = "&nbsp;&nbsp;"+a.toString()+" + "+b.toString()+" = ";
        //alert(outString);
        document.getElementById("equation").innerHTML = outString;
        answer=0;
        document.getElementById("ans").value = "0";
        var messagePanel = document.getElementById("messagePanel");
        messagePanel.innerHTML = "";
    }
```

Label (Randomized)



```
function getAnswer(){
    var v = document.getElementById("ans").value;
    //alert(v);
    answer = parseInt(v);
    //alert(answer+1);
}
```

Input Form onchange="getAnswer()"



```
    function checkAnswer(){
        //alert("I am here");
        var equal = (answer == (a+b));
        var messagePanel = document.getElementById("messagePanel");
        if (equal)
            messagePanel.innerHTML = "&nbsp;&nbsp;Correct !";
        else{
            messagePanel.innerHTML = "&nbsp;&nbsp;Wrong !";
        }
    }
```

Check Answer

Project 3: Random Insult Generator

SECTION 1

Creating a Random Insult Generator

- We can extend the decision maker example to create a program that generates a random insult every time you run it!

```
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
// Pick a random body part from the randomBodyParts array:
❶ var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Pick a random adjective from the randomAdjectives array:
❷ var randomAdjective = randomAdjectives[Math.floor(Math.random()* 3)];
// Pick a random word from the randomWords array:
❸ var randomWord = randomWords[Math.floor(Math.random() * 5)];
// Join all the random strings into a sentence:
var randomInsult = "Your " + randomBodyPart + " is like a " +
randomAdjective + " " + randomWord + "!!!";
randomInsult;
"Your Nose is like a Stupid Marmot!!!"
```

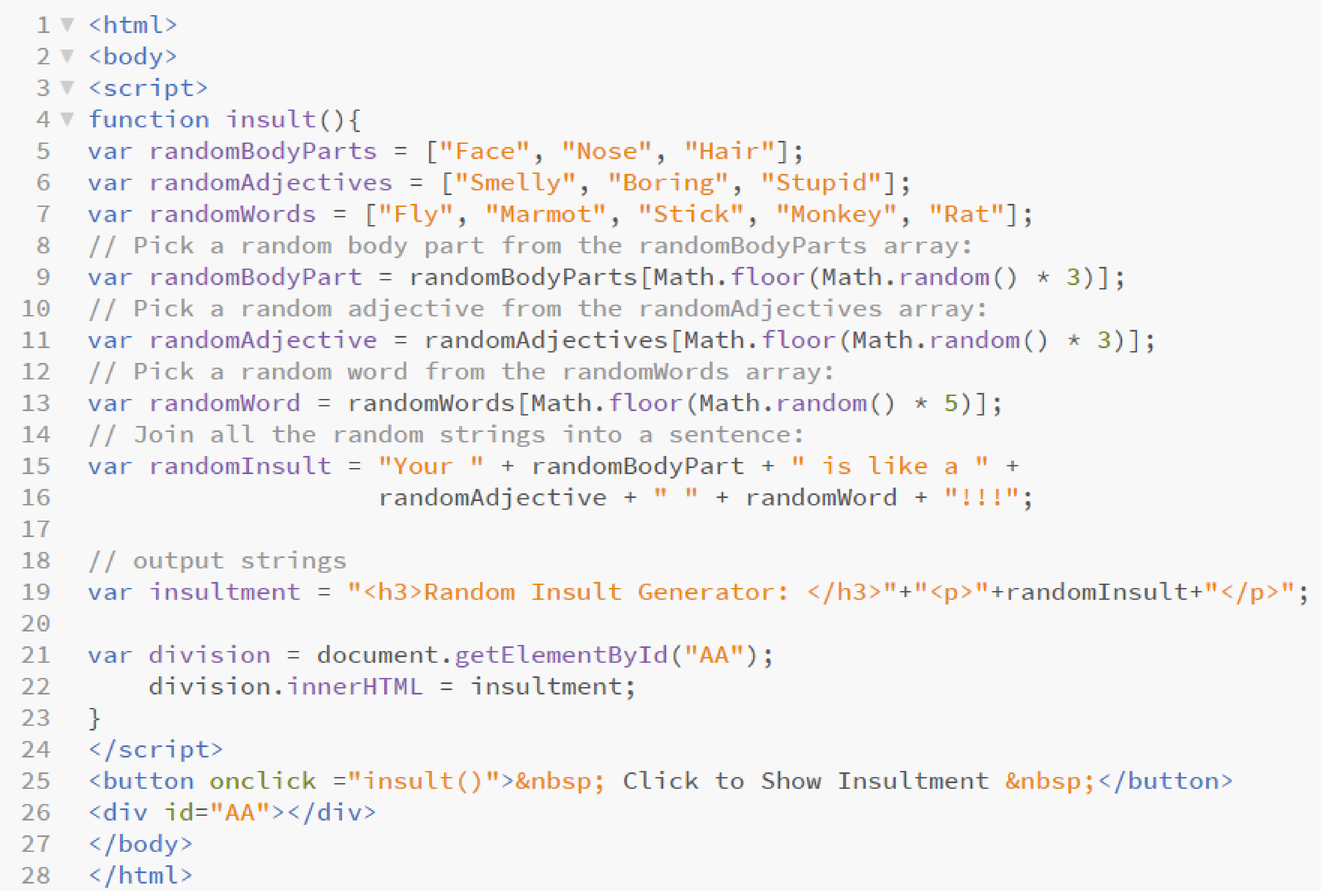
Creating a Random Insult Generator

- Here we have three arrays, and in lines ❶, ❷, and ❸, we use three indexes to pull a random word from each array. Then, we combine them all in the variable randomInsult to create a complete insult.
- At ❶ and ❷ we’re multiplying by 3 because randomAdjectives and randomBodyParts both contain three elements. Likewise, we’re multiplying by 5 at ❸ because randomWords is five elements long.
- Notice that we add a string with a single space between randomAdjective and randomWord. Try running this code a few times — you should get a different random insult each time!

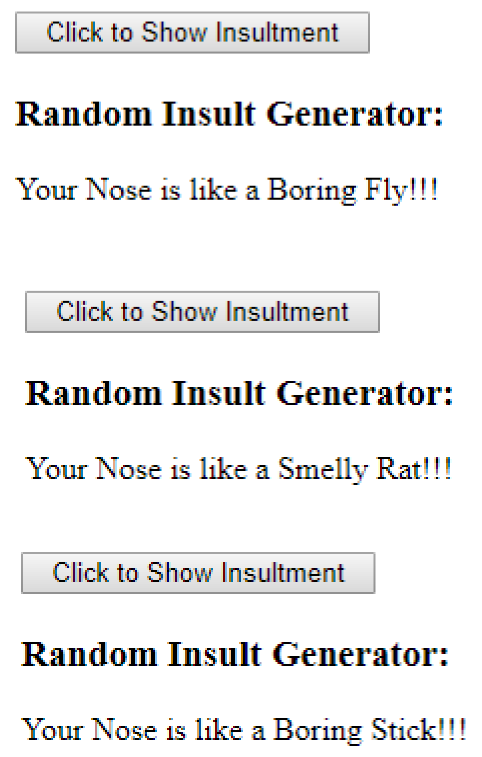
Random Insult Generator

Demo Program: insult.html

Go Brackets!!!



```
<html>
<body>
<script>
function insult(){
var randomBodyParts = ["Face", "Nose", "Hair"];
var randomAdjectives = ["Smelly", "Boring", "Stupid"];
var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];
// Pick a random body part from the randomBodyParts array:
var randomBodyPart = randomBodyParts[Math.floor(Math.random() * 3)];
// Pick a random adjective from the randomAdjectives array:
var randomAdjective = randomAdjectives[Math.floor(Math.random() * 3)];
// Pick a random word from the randomWords array:
var randomWord = randomWords[Math.floor(Math.random() * 5)];
// Join all the random strings into a sentence:
var randomInsult = "Your " + randomBodyPart + " is like a " +
                   randomAdjective + " " + randomWord + "!!!";

// output strings
var insultment = "<h3>Random Insult Generator: </h3>"+"<p>"+randomInsult+"</p>";

var division = document.getElementById("AA");
    division.innerHTML = insultment;
}
</script>
<button onclick ="insult()">&nbsp; Click to Show Insultment &nbsp;</button>
<div id="AA"></div>
</body>
</html>
```



Random Insult Generator:

Your Nose is like a Boring Fly!!!

Random Insult Generator:

Your Nose is like a Smelly Rat!!!

Random Insult Generator:

Your Nose is like a Boring Stick!!!

Hangman Game

SECTION 7

Hangman Game

- In this chapter we’ll build a **Hangman** game! We’ll learn how to use dialogs to make the game interactive and take input from someone playing the game.
- Hangman is a word-guessing game. One player picks a secret word, and the other player tries to guess it.
- For example, if the word were TEACHER, the first player would write:

_ _ _ _ _ _ _

Hangman Game

- The guessing player tries to guess the letters in the word. Each time they guess a letter correctly, the first player fills in the blanks for each occurrence of that letter.
- For example, if the guessing player guessed the letter E, the first player would fill in the Es in the word TEACHER like so:

_ E _ _ _ E _

Hangman Game

- When the guessing player guesses a letter that isn’t in the word, they lose a point and the first player draws part of a stick-man for each wrong guess. If the first player completes the stickman before the guessing player guesses the word, the guessing player loses.
- In our version of Hangman, the **JavaScript** program will choose the word and the human player will guess letters. We won’t be drawing the stickman, because we haven’t yet learned how to draw in JavaScript (we’ll learn how to do that in Chapter 13).

Interacting with a Player

SECTION 8

Interacting with a Player

- To create this game, we have to have some way for the guessing player (human) to enter their choices.
- One way is to open a pop-up window (which JavaScript calls a prompt) that the player can type into.

Creating a Prompt

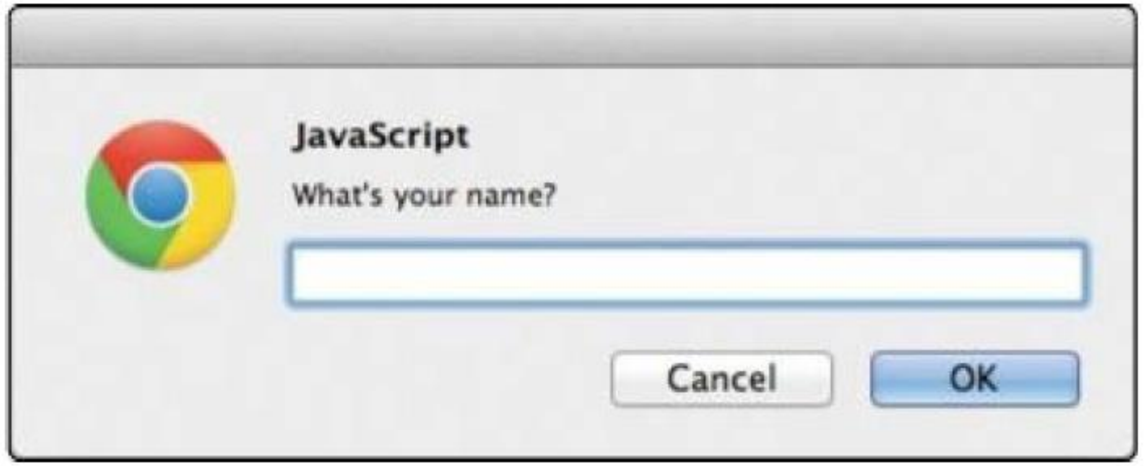
- First, let’s create a new HTML document. Using **File ▸ Save As**, save your *page.html* file from Chapter 5 as *prompt.html*. To create a prompt, enter this code between the <script> tags of *prompt.html* and refresh the browser:

```
var name = prompt("What's your name?");
console.log("Hello " + name);
```

- Here we create a new variable, called name, and assign to it the value returned from calling prompt("What's your name?"). When prompt is called, a small window (or *dialog*) is opened, which should look like Figure 7-1.

Creating a Prompt

Demo Program: prompt.html


JavaScript

What's your name?

Figure 7-1. A prompt dialog

Creating a Prompt

- Calling `prompt("What's your name?")` pops up a window with the text "What's your name?" along with a text box for input. At the bottom of the dialog are two buttons, Cancel and OK. In Chrome, the dialog has the heading *JavaScript*, to inform you that JavaScript opened the prompt.
- When you enter text in the box and click OK, that text becomes the value that is returned by `prompt`. For example, if I were to enter my name into the text box and click OK, JavaScript would print this in the console:

```
Hello Nick
```

- Because I entered *Nick* in the text box and clicked OK, the string `"Nick"` is saved in the variable `name` and `console.log` prints `"Hello " + "Nick"`, which gives us `"Hello Nick"`.

Using confirm to Ask a Yes or No Question

- The `confirm` function is a way to take user input without a text box by asking for a yes or no (Boolean) answer. For example, here we use `confirm` to ask the user if they like cats (see Figure 7-2).
- If so, the variable `likesCats` is set to true, and we respond with "You're a cool cat!"
- If they don't like cats, `likesCats` is set to false, so we respond with "Yeah, that's fine. You're still cool!"

Using confirm to Ask a Yes or No Question

Demo Program: confirm.html

```
var likesCats = confirm("Do you like cats?");
if (likesCats) {
  console.log("You're a cool cat!");
} else {
  console.log("Yeah, that's fine. You're still cool!");
}
```

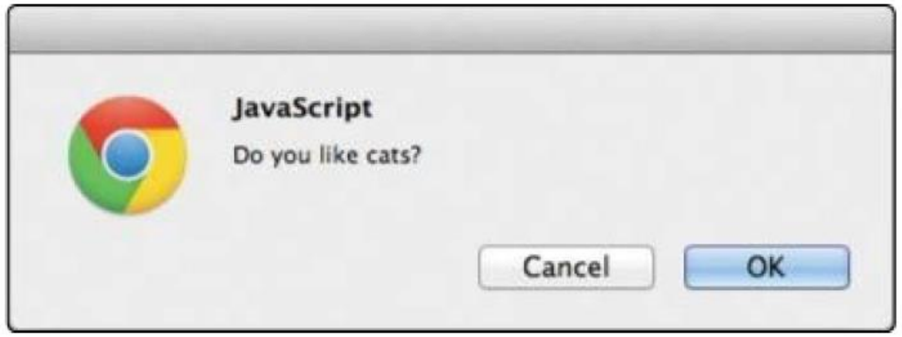


Figure 7-2. A confirm dialog

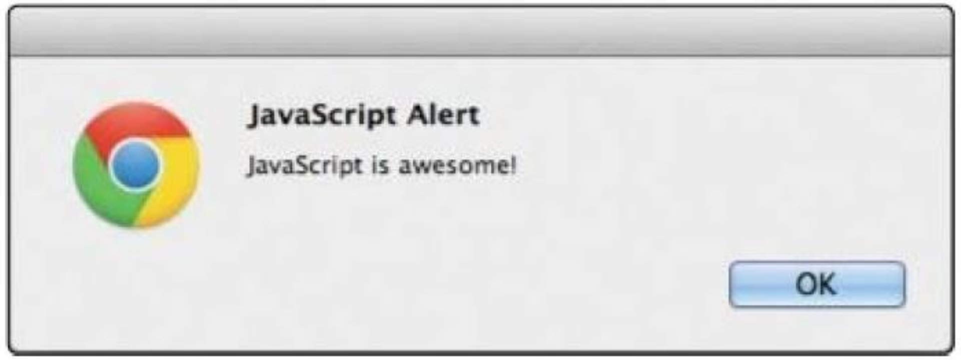
The answer to the **confirm** prompt is returned as a Boolean value. If the user clicks OK in the **confirm** dialog shown in Figure 7-2, true is returned. If they click Cancel, false is returned.

Using Alerts to Give a Player Information

- If you want to just give the player some information, you can use an alert dialog to display a message with an OK button. For example, if you think that JavaScript is awesome, you might use this `alert` function:

```
alert("JavaScript is awesome!");
```

- Figure 7-3 shows what this simple alert dialog would look like.
- Alert dialogs just display a message and wait until the user clicks OK.


JavaScript Alert

JavaScript is awesome!

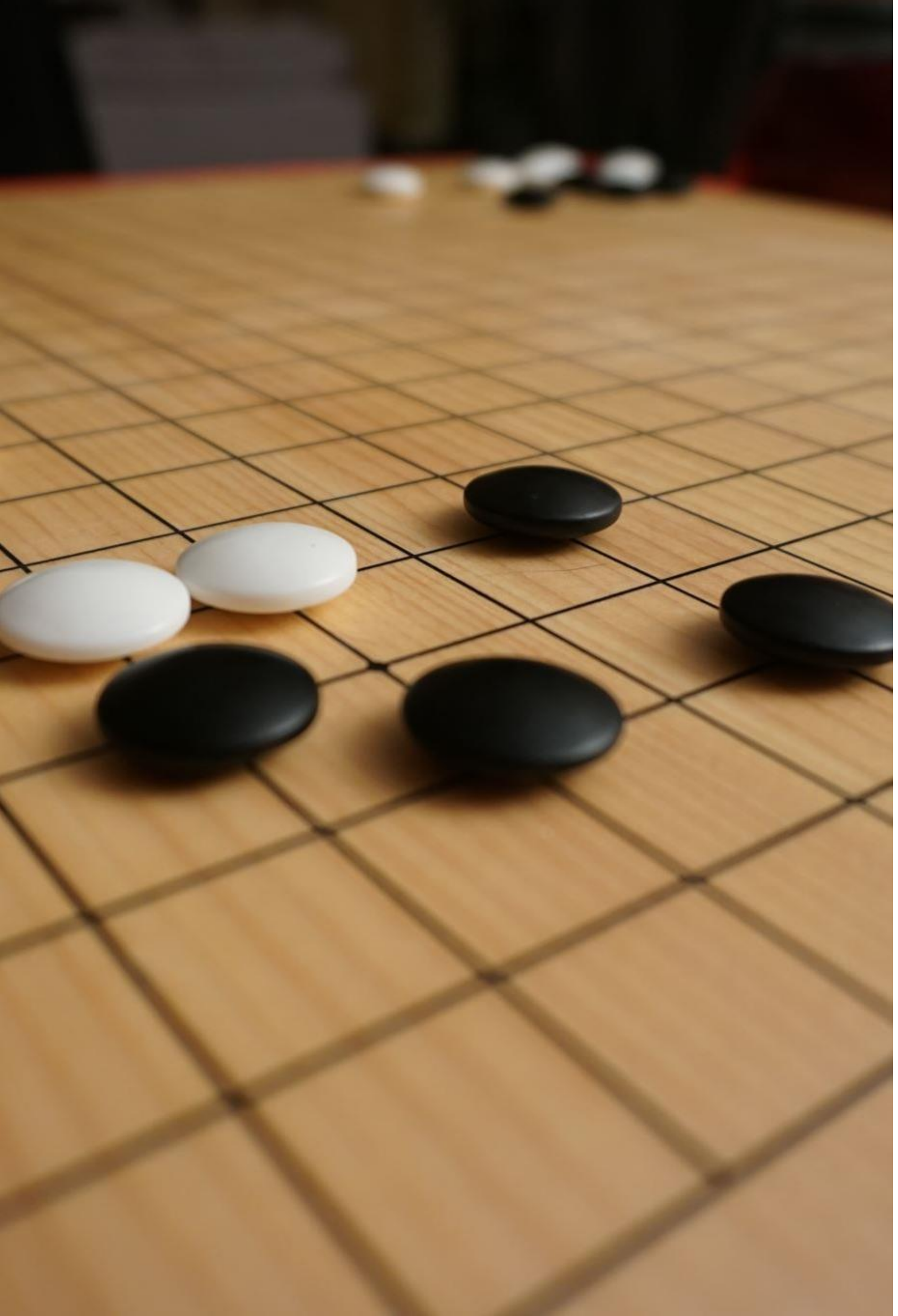
Figure 7-3. An alert dialog

Why Use alert Instead of console.log?

- Why use an alert dialog in a game instead of using console.log?
  1. First, because if all you want to do is tell the player something, using alert means the player doesn't have to interrupt game play to open the console to see a status message.
  2. Second, calling alert (as well as prompt and confirm) pauses the JavaScript interpreter until the user clicks OK (or Cancel, in the case of prompt and confirm).
- That means the player has time to read the alert. On the other hand, when you use console.log, the text is displayed immediately and the interpreter moves on to the next line in your program.

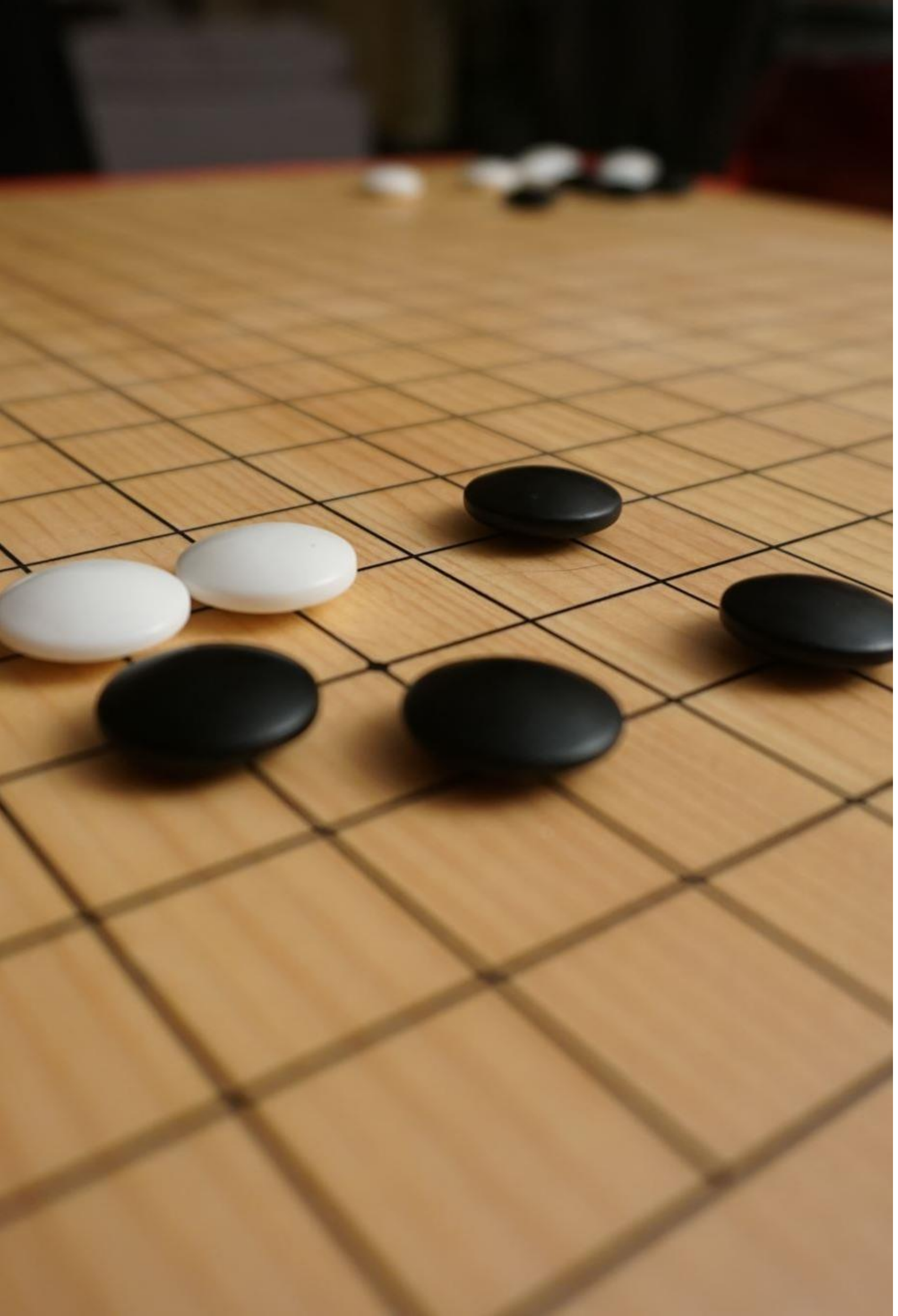
Designing Your Game

SECTION 9



Designing Your Game

- Before we start writing the Hangman game, let’s think about its structure. There are a few things we need our program to do:
  1. Pick a random word.
  2. Take the player’s guess.
  3. Quit the game if the player wants to.
  4. Check that the player’s guess is a valid letter.
  5. Keep track of letters the player has guessed.
  6. Show the player their progress.
  7. Finish when the player has guessed the word.



Pseudo Code

- Apart from the first and last tasks (picking a word for the player to guess and finishing the game), these steps all need to happen multiple times, and we don’t know how many times (it depends on how well the player guesses). When you need to do the same thing multiple times, you know you’ll need a loop.
- But this simple list of tasks doesn’t really give us any idea of what needs to happen when. To get a better idea of the structure of the code, we can use ***pseudocode***.

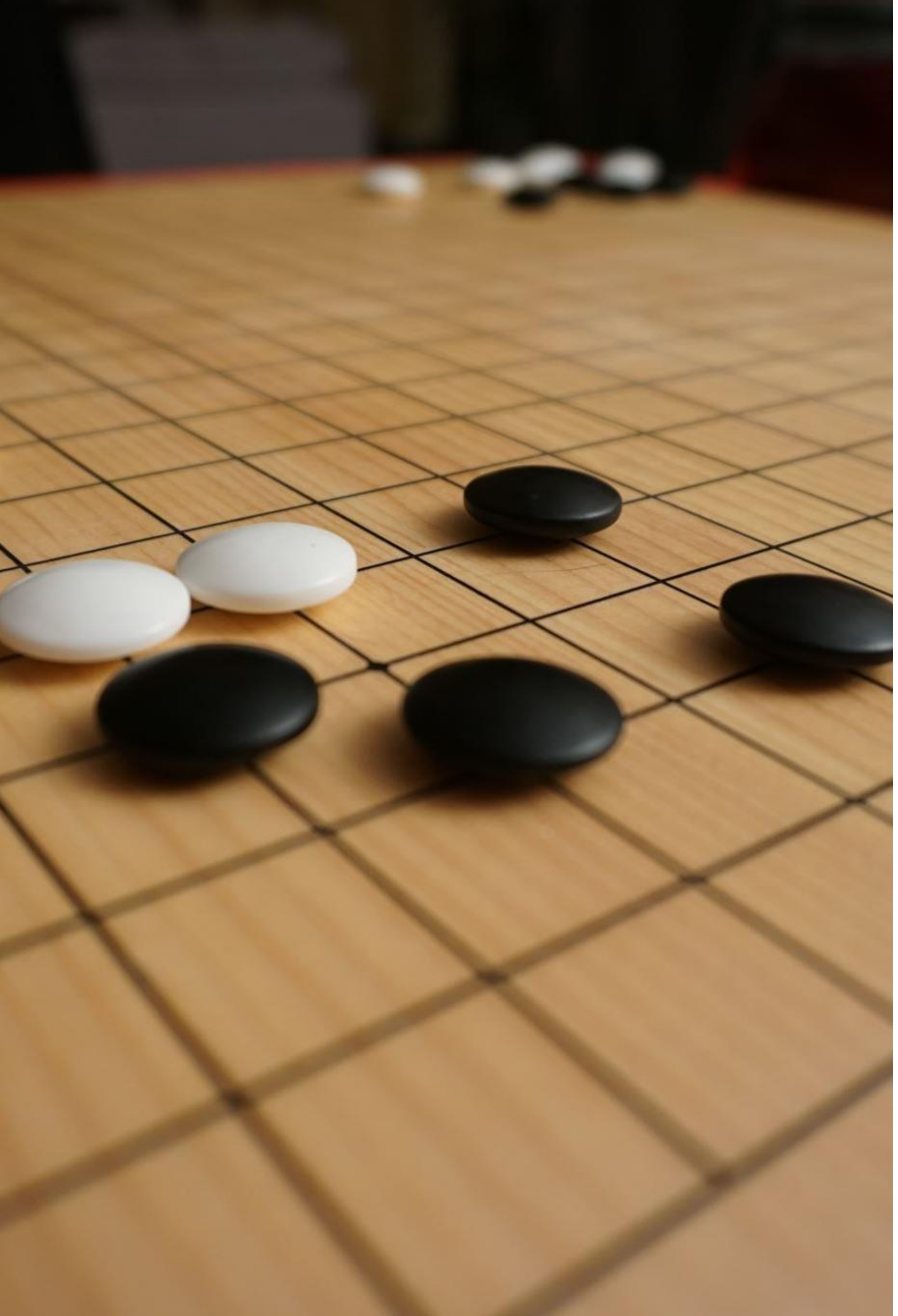
Pseudo Code

- Pseudocode is a handy tool that programmers often use to design programs. It means “fake code,” and it’s a way of describing how a program will work that looks like a cross between written English and code.
- Pseudocode has loops and conditionals, but other than that, everything is just plain English. Let’s look at a pseudocode version of our game to get an idea:

```
Pick a random word

While the word has not been guessed {
  Show the player their current progress
  Get a guess from the player

  If the player wants to quit the game {
    Quit the game
  }
  Else If the guess is not a single letter {
    Tell the player to pick a single letter
  }
  Else {
    If the guess is in the word {
      Update the player's progress with the guess
    }
  }
}

Congratulate the player on guessing the word
```

Pseudo Code

- As you can see, none of this is real code, and no computer could understand it. But it gives us an idea of how our program will be structured, before we get to actually writing the code and having to deal with the messy details, like how we're going to pick a random word.



Tracking the State of the Word

- In the previous pseudocode, one of the first lines says, “Show the player their current progress.” For the Hangman game, this means filling in the letters that the player has guessed correctly and showing which letters in the secret word are still blank. How are we going to do this? We can actually keep track of the player’s progress in a similar way to how traditional Hangman works: by keeping a collection of blank spaces and filling them in as the player guesses correct letters.
- In our game, we’ll do this using an array of blanks for each letter in the word. We’ll call this the answer array, and we’ll fill it with the player’s correct guesses as they’re made. We’ll represent each blank with the string "_".

Tracking the State of the Word

- The answer array will start out as a group of these empty entries equal in number to the letters in the secret word. For example, if the secret word is *fish*, the array would look like this:

  ["_", "_", "_", "_"]

- If the player correctly guessed the letter *i*, we'd change the second blank to an *i*:

  ["_", "i", "_", "_"]

- Once the player guesses all the correct letters, the completed array would look like this:

  ["f", "i", "s", "h"]

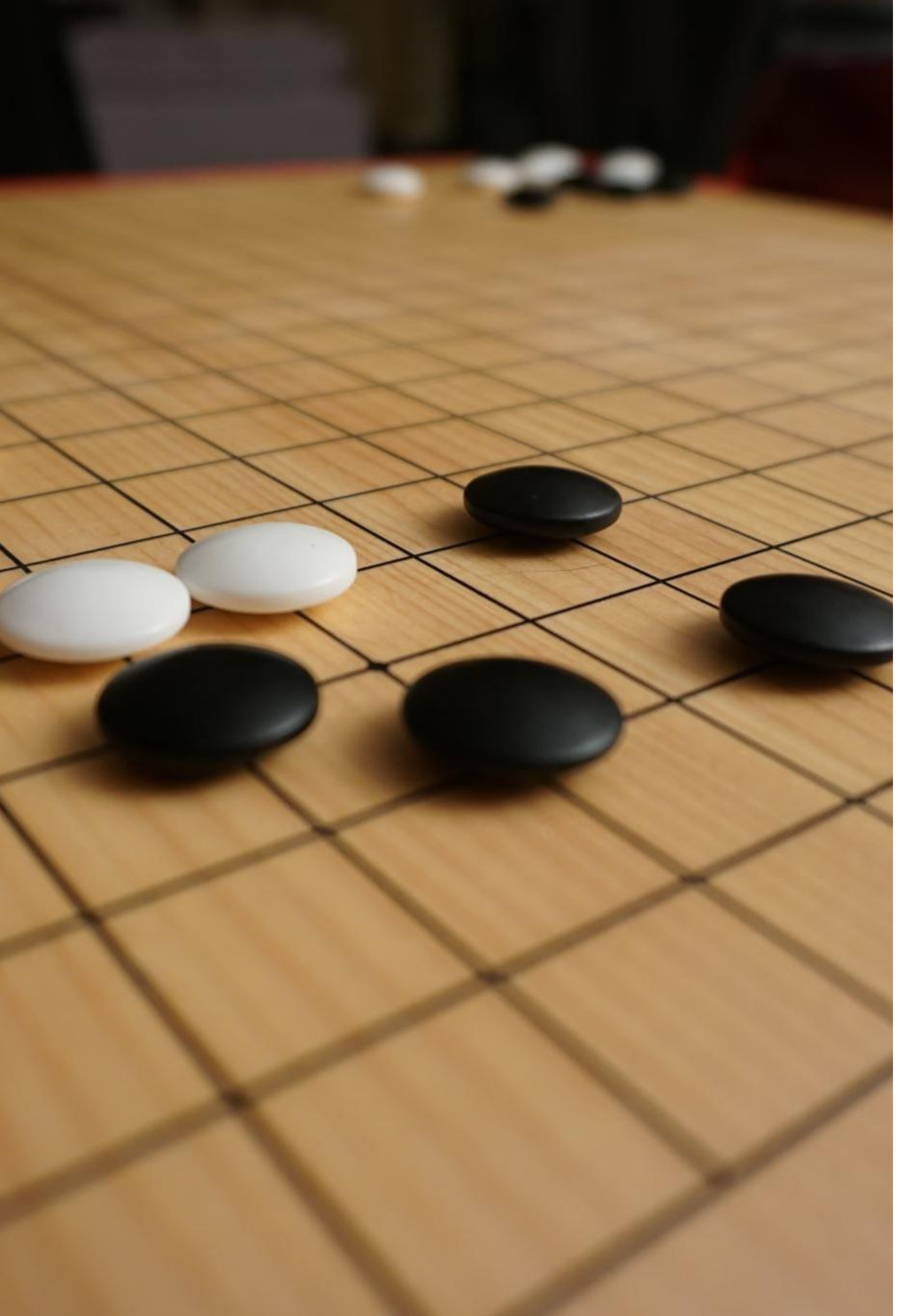
- We'll also use a variable to keep track of the number of remaining letters the player has to guess. For every occurrence of a correctly guessed letter, this variable will decrease by 1. Once it hits 0, we know the player has won.

Designing the Game Loop

- The main game takes place inside a **while** loop (in our pseudocode, this loop begins with the line
- "While the word has not been guessed"). In this loop we display the current state of the word being guessed (beginning with all blanks); ask the player for a guess (and make sure it's a valid, single-letter guess); and update the answer array with the chosen letter, if that letter appears in the word.

Designing the Game Loop

- Almost all computer games are built around a loop of some kind, often with the same basic structure as the loop in our Hangman game. A game loop generally does the following:
    1. Takes input from the player
    2. Updates the game state
    3. Displays the current state of the game to the player



Designing the Game Loop

- Even games that are constantly changing follow this same kind of loop — they just do it really fast. In the case of our Hangman game, the program takes a guess from the player, updates the answer array if the guess is correct, and displays the new state of the answer array.
- Once the player guesses all letters in the word, we show the completed word and a congratulatory message telling them that they won.

Coding the Game

SECTION 10

Coding the Game

- Now that we know the general structure of our game, we can start to go over how the code will look.
- The following sections will walk you through all the code in the game. After that, you'll see the whole game code in one listing so you can type it up and play it yourself.

Choosing a Random Word

- The first thing we have to do is to choose a random word. Here’s how that will look:

```
❶ var words = [
   "javascript",
   "monkey",
   "amazing",
   "pancake"
];
❷ var word = words[Math.floor(Math.random() * words.length)];
```

Choosing a Random Word

- We begin our game at ❶ by creating an array of words (*javascript*, *monkey*, *amazing*, and *pancake*) to be used as the source of our secret word, and we save the array in the words variable. The words should be all lowercase.
- At ❷ we use Math.random and Math.floor to pick a random word from the array, as we did with the random insult generator in Chapter 3.

Creating the Answer Array

- Next we create an empty array called answerArray and fill it with underscores (_) to match the number of letters in the word.

```
var answerArray = [];
❶ for (var i = 0; i < word.length; i++) {
answerArray[i] = "_";
}
var remainingLetters = word.length;
```

Creating the Answer Array

- The for loop at ❶ creates a looping variable i that starts at 0 and goes up to (but does not include) word.length. Each time around the loop, we add a new element to answerArray, at answerArray[i].
- When the loop finishes, answerArray will be the same length as word. For example, if word is "monkey" (which has six letters), answerArray will be ["_", "_", "_", "_", "_", "_"] (six underscores).
- Finally, we create the variable remainingLetters and set it to the length of the secret word. We'll use this variable to keep track of how many letters are left to be guessed. Every time the player guesses acorrect letter, this value will be *decremented* (reduced) by 1 for each instance of that letter in the word.

Coding the Game Loop

- The skeleton of the game loop looks like this:

```
while (remainingLetters > 0) {
  // Game code goes here
  // Show the player their progress
  // Take input from the player
  // Update answerArray and remainingLetters for every correct guess
}
```

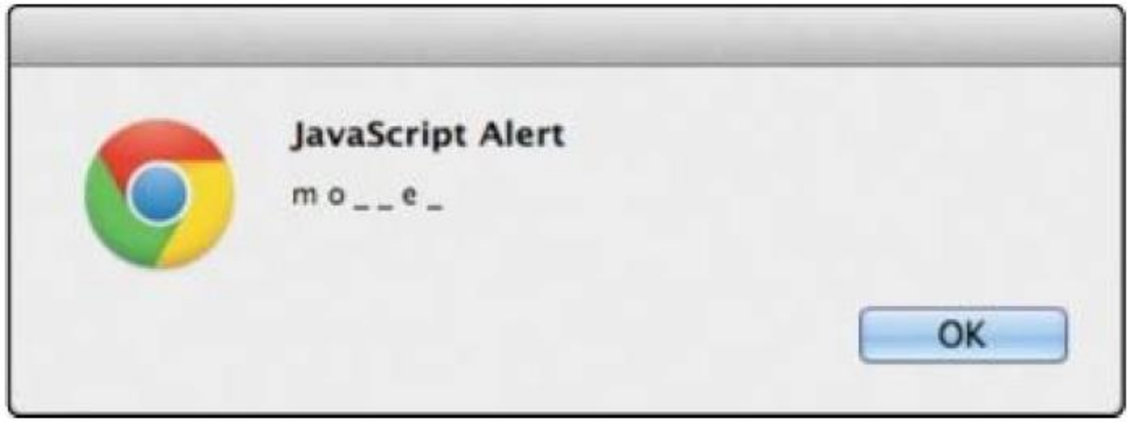
- We use a while loop, which will keep looping as long as remainingLetters > 0 remains true. The body of the loop will have to update remainingLetters for every correct guess the player makes. Once the player has guessed all the letters, remainingLetters will be 0 and the loop will end.
- The following sections explain the code that will make up the body of the game loop.

Showing the Player’s Progress

- The first thing we need to do inside the game loop is to show the player their current progress:

```
alert(answerArray.join(" "));
```

- We do that by joining the elements of answerArray into a string, using the space character as the separator, and then using alert to show that string to the player. For example, let’s say the word is monkey and the player has guessed m, o, and e so far. The answer array would look like this ["m", "o", "_", "_", "e", "_"], and answerArray.join(" ") would be "m o _ _ e _". The alert dialog would then look like Figure 7-4.



```
m o _ _ e _
```

Figure 7-4. Showing the player’s progress using `alert`

Handling the Player’s Input

- Now we have to get a guess from the player and ensure that it’s a single character.

```
❶ var guess = prompt("Guess a letter, or click Cancel to stop playing.");
❷ if (guess === null) {
          break;
❸ } else if (guess.length != 1) {
          alert("Please enter a single letter.");
  } else {
❹         // Update the game state with the guess
  }
```

Handling the Player's Input

- At ➊, `prompt` takes a guess from the player and saves it to the variable `guess`. One of four things will happen at this point.
- First, if the player clicks the Cancel button, then `guess` will be `null`.
- We check for this condition at ➋ with `if (guess === null)`. If this condition is true, we use break to exit the loop.

Handling the Player’s Input

- The second and third possibilities are that the player enters either nothing or too many letters. If they enter nothing but click OK, guess will be the empty string "". In this case, guess.length will be 0. If they enter anything more than one letter, guess.length will be greater than 1.
- At ❸, we use else if (guess.length !== 1) to check for these conditions, ensuring that guess is exactly one letter. If it’s not, we display an alert saying, “Please enter a single letter.”
- The fourth possibility is that the player enters a valid guess of one letter. Then we have to update the game state with their guess using the else statement at ❹, which we’ll do in the next section.

Updating the Game State

- Once the player has entered a valid guess, we must update the game's answerArray according to the guess. To do that, we add the following code to the else statement:

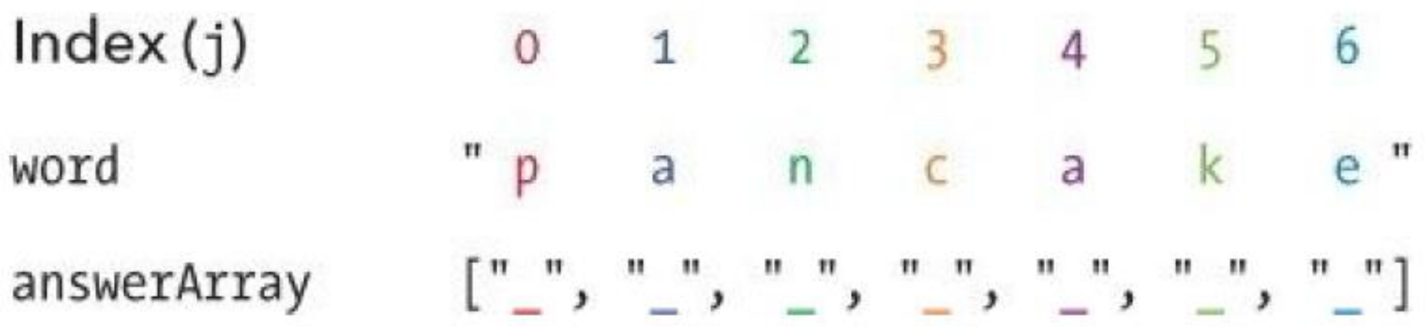
```
❶ for (var j = 0; j < word.length; j++) {
❷     if (word[j] === guess) {
              answerArray[j] = guess;
❸             remainingLetters--;
       }
   }
```

Updating the Game State

- At ❶, we create a for loop with a new looping variable called j, which runs from 0 up to word.length. (We’re using j as the variable in this loop because we already used i in the previous for loop.)
- We use this loop to step through each letter of word. For example, let’s say word is pancake. The first time around this loop, when j is 0, word[j] will be "p". The next time, word[j] will be "a", then "n", "c", "a", "k", and finally "e".

Updating the Game State

- At ❷, we use **`if (word[j]===guess)`** to check whether the current letter we're looking at matches the player's guess. If it does, we use **`answerArray[j]=guess`** to update the answer array with the current guess. For each letter in the word that matches guess, we update the answer array at the corresponding point.
- This works because the looping variable j can be used as an index for **`answerArray`** just as it can be used as an index for word, as you can see in Figure 7-5.



Index (j)	0	1	2	3	4	5	6
word	"p	a	n	c	a	k	e"
answerArray	["_",	"_",	"_",	"_",	"_",	"_",	"_"]

Figure 7-5. The same index can be used for both `word` *and* `answerArray`.

Updating the Game State

- For example, imagine we've just started playing the game and we reach the for loop at ❶. Let's say `word` is `"pancake",` guess is `"a"`, and `answerArray` currently looks like this:

  ["_", "_", "_", "_", "_", "_", "_"]

- The first time around the `for` loop at ❶, `j` is 0, so `word[j]` is `"p".` Our guess is `"a",` so we skip the if statement at ❷ (because `"p" === "a"` is `false`). The second time around, j is 1, so `word[j]` is `"a"`.

- This *is* equal to `guess`, so we enter the if part of the statement. The line `answerArray[j] = guess;` sets the element at index 1 (the second element) of `answerArray` to guess, so `answerArray` now looks like this:

  ["_", "a", "_", "_", "_", "_", "_"]

Updating the Game State

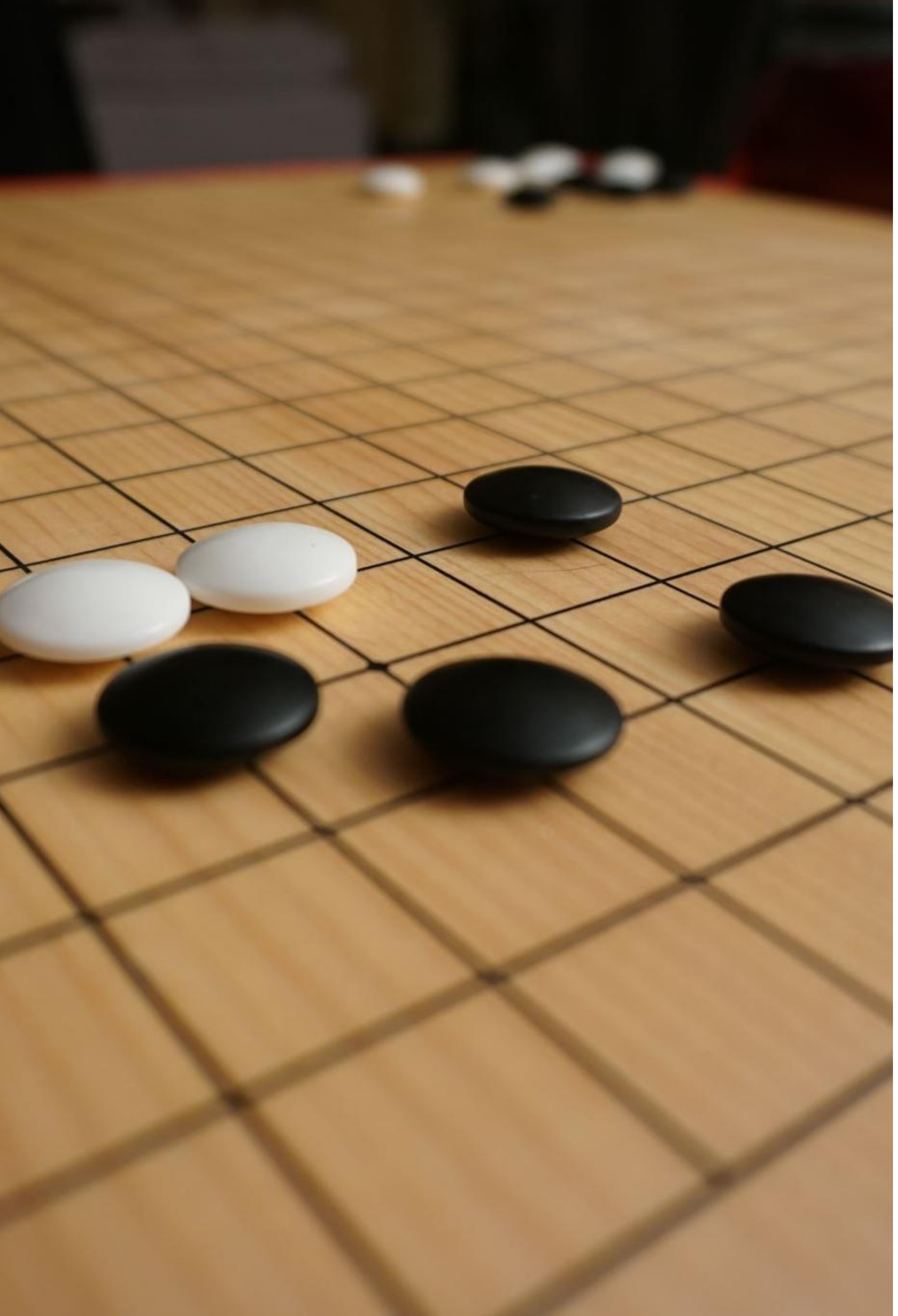
- The next two times around the loop, word[j] is "n" and then "c", which don't match guess. However, when j reaches 4, word[j] is "a" again. We update answerArray again, this time setting the element at index 4 (the fifth element) to guess. Now answerArray looks like this:

  ["_", "a", "_", "_", "a", "_", "_"]

- The remaining letters don't match "a", so nothing happens the last two times around the loop. At the end of this loop, answerArray will be updated with all the occurrences of guess in word.
- For every correct guess, in addition to updating answerArray, we also need to decrement remainingLetters by 1. We do this at ❸ using remainingLetters--;. Every time guess matches a letter in word, remainingLetters decreases by 1. Once the player has guessed all the letters correctly, remainingLetters will be 0.

Ending the Game

SECTION 11



Ending the Game

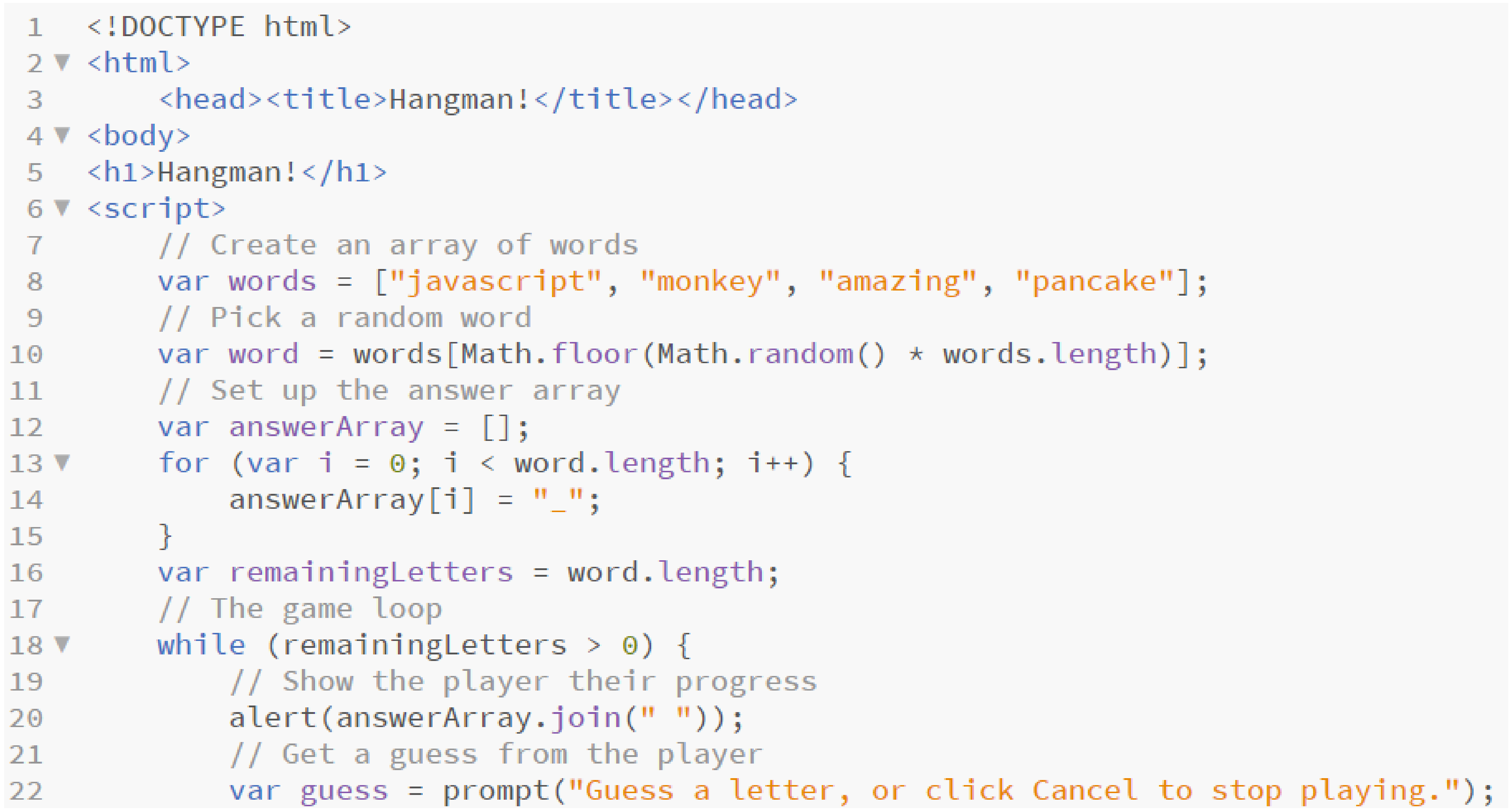
- As we've already seen, the main game loop condition is **`remainingLetters > 0`**, so as long as there are still letters to guess, the loop will keep looping. Once **`remainingLetters`** reaches 0, we leave the loop. We end with the following code:

```
alert(answerArray.join(" "));
alert("Good job! The answer was " + word);
```

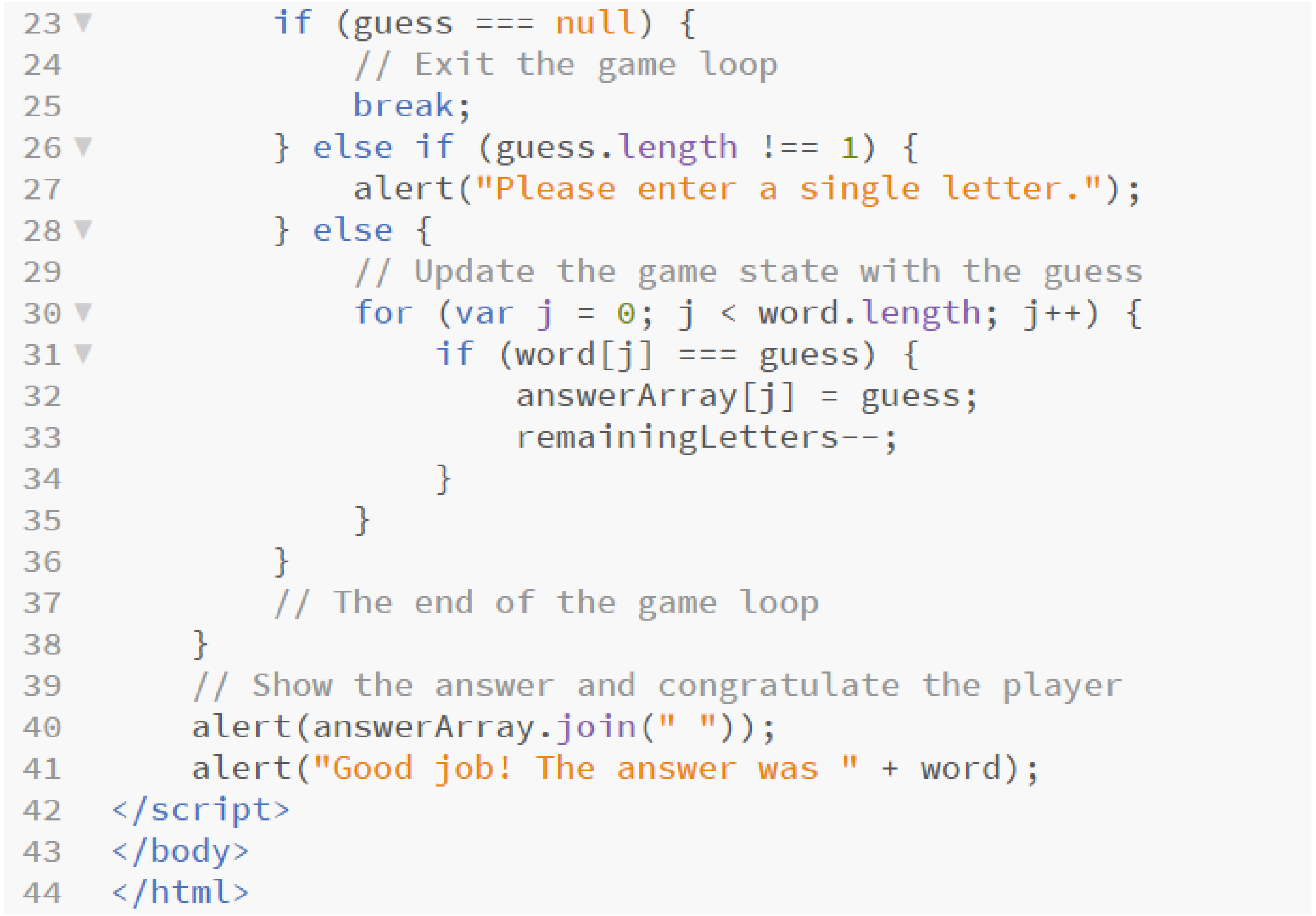
- The first line uses alert to show the answer array one last time. The second line uses alert again to congratulate the winning player.

The Game Code Demo Program: HangMan.html

- Now we’ve seen all the code for the game, and we just need to put it together. What follows is the full listing for our Hangman game. I’ve added comments throughout to make it easier for you to see what’s happening at each point. It’s quite a bit longer than any of the code we’ve written so far, but typing it out will help you to become more familiar with writing JavaScript.
- Create a new HTML file called *hangman.html* and type the following into it:



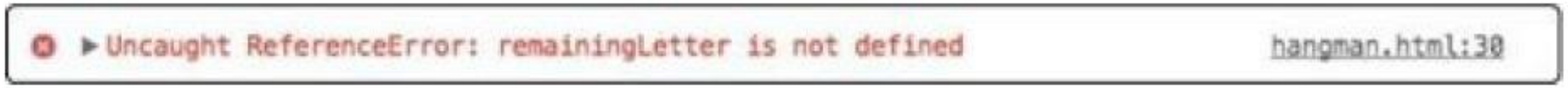
```
<!DOCTYPE html>
<html>
    <head><title>Hangman!</title></head>
<body>
<h1>Hangman!</h1>
<script>
    // Create an array of words
    var words = ["javascript", "monkey", "amazing", "pancake"];
    // Pick a random word
    var word = words[Math.floor(Math.random() * words.length)];
    // Set up the answer array
    var answerArray = [];
    for (var i = 0; i < word.length; i++) {
        answerArray[i] = "_";
    }
    var remainingLetters = word.length;
    // The game loop
    while (remainingLetters > 0) {
        // Show the player their progress
        alert(answerArray.join(" "));
        // Get a guess from the player
        var guess = prompt("Guess a letter, or click Cancel to stop playing.");
```



```
        if (guess === null) {
            // Exit the game loop
            break;
        } else if (guess.length !== 1) {
            alert("Please enter a single letter.");
        } else {
            // Update the game state with the guess
            for (var j = 0; j < word.length; j++) {
                if (word[j] === guess) {
                    answerArray[j] = guess;
                    remainingLetters--;
                }
            }
        }
        // The end of the game loop
    }
    // Show the answer and congratulate the player
    alert(answerArray.join(" "));
    alert("Good job! The answer was " + word);
</script>
</body>
</html>
```

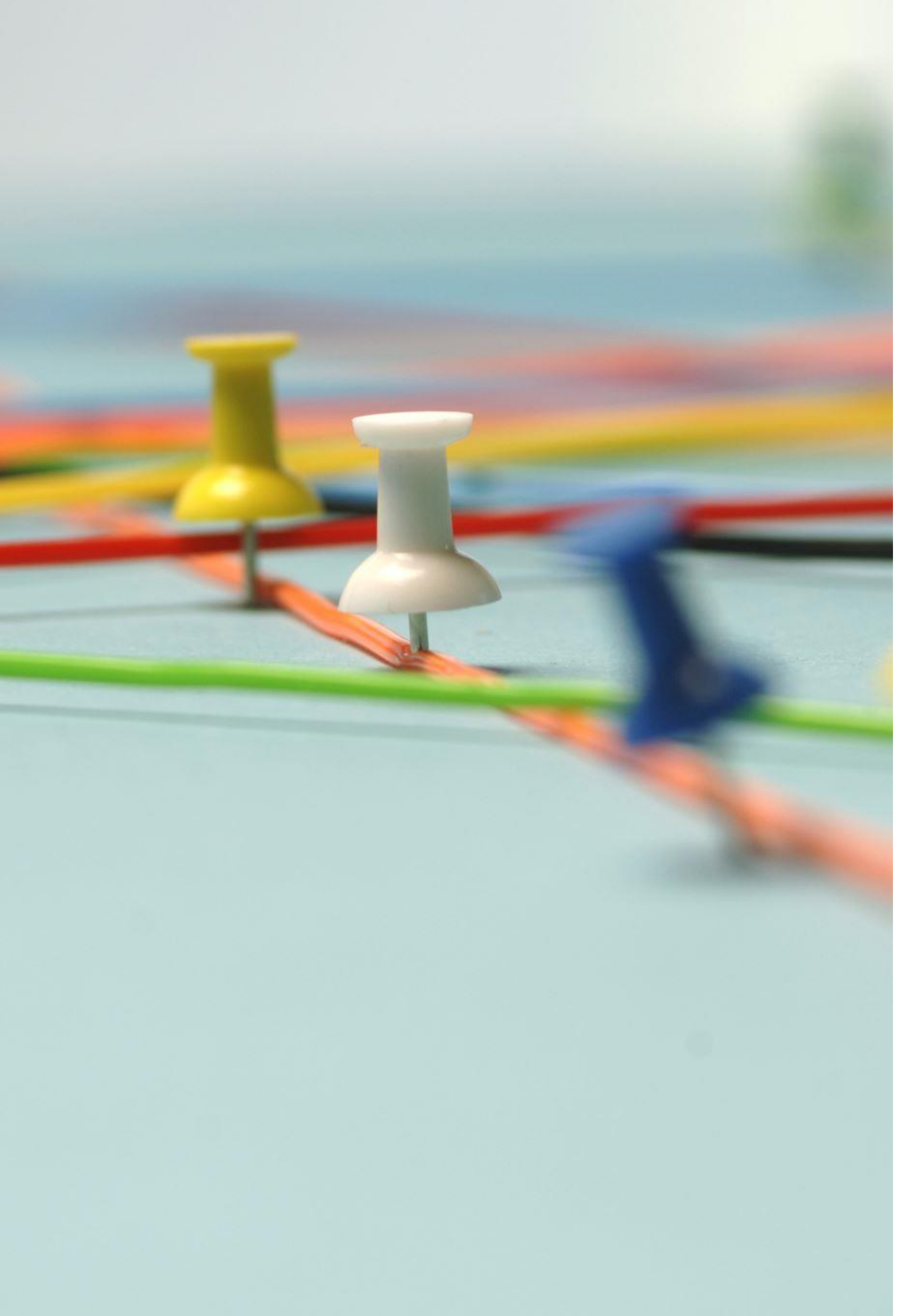
The Game Code

- If the game doesn't run, make sure that you typed in everything correctly. If you make a mistake, the JavaScript console can help you find it. For example, if you misspell a variable name, you'll see something like Figure 7-6 with a pointer to where you made your mistake.



```
▶ Uncaught ReferenceError: remainingLetter is not defined          hangman.html:30
```

Figure 7-6. A JavaScript error in the Chrome console



The Game Code

- If you click **`hangman.html:30`**, you'll see the exact line where the error is. In this case, it's showing us that we misspelled **`remainingLetters`** as **`remainingLetter`** at the start of the **`while`** loop.
- Try playing the game a few times. Does it work the way you expected it to work? Can you imagine the code you wrote running in the background as you play it?