JavaScript Primer

Arrays, Functions, Closures

Arrays

- Arrays in JavaScript are a special type of object
- They work like regular object, except they have a length property
- Elements accessed using subscripting [index] syntax
- Length !== number of elements in array
- Length === highest index + 1

```
var array = ["dog", "cat", "fish"];
array[80] = "snake";
array.length; // 81
```

Iterating over an array

```
for (var i = 0; i < a.length; i++) { // approach 1
   // Do something with a[i];
}
```

```
for (var i = 0, len = a.length; i < len; i++) { // approach 2
   // Do something with a[i];
}
```

```
for (var i = 0, item; item = a[i++];) { // approach 3
   // Do something with item;
}
```

Arrays: forEach loop

```
["dog", "cat", "hen"].forEach(function(currentValue, index, array) {
    // Do something with currentValue or array[index]
});
```

array.forEach(*callback*[, *thisArg*]);

Parameters:

- callback: - Function to execute for each element, taking three arguments
- thisArg: - *Optional*. Value to use as this when executing callback.

Array Methods

Method name	Description
a.toString()	Returns a string with the toString() of each element separated by commas.
a.toLocaleString()	Returns a string with the toLocaleString() of each element separated by commas.
a.concat(item1[, item2[, ...[, itemN]]])	Returns a new array with the items added on to it.
a.join(sep)	Converts the array to a string — with values delimited by the sep param
a.pop()	Removes and returns the last item.
a.push(item1, ..., itemN)	Adds one or more items to the end.
a.reverse()	Reverses the array.
a.shift()	Removes and returns the first item.
a.slice(start[, end])	Returns a sub-array.
a.sort([cmpfn])	Takes an optional comparison function.
a.splice(start, delcount[, item1[, ...[, itemN]]])	Lets you modify an array by deleting a section and replacing it with more items.
a.unshift(item1[, item2[, ...[, itemN]]])	Prepends items to the start of the array.

JavaScript functions

- Note that JavaScript functions are themselves objects
- You can add or change properties on them just like you can on Objects
- They can be assigned to variables, array entries, & properties of other objects
- They can be passed as arguments to functions
- They can be returned as values from functions
- They are first-class objects

Functions

```
// Along with objects, a core component of JavaScript
  function add(x, y) {
    var total = x + y;
    return total;
  }
```

```
// Function using variable inside its body called arguments
  function add() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
      sum += arguments[i];
    }
    return sum;
  }
```

Calling function with apply

```
function avg() {
      var sum = 0;
      for (var i = 0, j = arguments.length; i < j; i++) {
         sum += arguments[i];
      }
      return sum / arguments.length;
   }
```

```
avg.apply(null, [2, 3, 4, 5]);
```

Anonymous functions

```
var avg = function() {
    var sum = 0;
    for (var i = 0, j = arguments.length; i < j; i++) {
        sum += arguments[i];
    }
    return sum / arguments.length;
};
```

```
(function() {
    var b = 3;
    a += b;
})();
```

Recursive functions

```
function countChars(elm) {
    if (elm.nodeType == 3) { // TEXT_NODE
        return elm.nodeValue.length;
    }
    var count = 0;
    for (var i = 0, child; child = elm.childNodes[i]; i++) {
        count += countChars(child);
    }
    return count;
}
```

Recursion with (IIFFs) anonymous functions

```
var charsInBody = (function counter(elm) {
  if (elm.nodeType == 3) { // TEXT_NODE
    return elm.nodeValue.length;
  }
  var count = 0;
  for (var i = 0, child; child = elm.childNodes[i]; i++) {
    count += counter(child);
  }
  return count;
})(document.body);
```

Scoping and functions

- In JavaScript scope is defined by functions, not by blocks
- Variable declarations are in scope from their point of declaration to the end of their enclosing function, regardless of block nesting
- Named function are in scope ***within the entire function*** within which they're declared

Function Invocation Patterns

- Method invocation
  - ***this*** is bound to the object upon invoking the method because the method belongs to the object.
- Function invocation
  - For functions that are not properties on objects, ***this*** is bound to the global object.
  - To bound ***this*** to the parent function, assign ***this*** to a variable in the parent function (var that = this;).
- Constructor invocation
  - Functions can be invoked with the new prefix similar to the way objects are constructed in other languages.
  - When this happens, this is bound to the new object.
- Apply invocation
  - Functions can have methods
  - The apply function is a method on the Function.prototype–the prototype for all JS functions

Apply and call

- Apply invocation
  - apply makes it possible to use one object's method in the context of another.
  - function.apply(null, [2, 3, 4, 5]);
  - We can do so by supplying the object to which ***this*** will be bound, also known as the context and an array with the correct number arguments.
- Call invocation
  - Works like apply, except that the arguments are passed directly in the arguments rather than as an array.
  - function.call(null, 2, 3, 4, 5);

Closures

- One of the most powerful abstractions that JavaScript has to offer

```
function makeAdder(a) {
   return function(b) {
      return a + b;
   };
}

var x = makeAdder(5); // closure x created
var y = makeAdder(20); // closure y created

console.log(x(6)); // ?
console.log(y(7)); // ?
```

Closure—what's happening?

- **Whenever** JavaScript executes a function, a 'scope' object is created to hold the local variables created within that function
- Normally JavaScript's garbage collector would clean up the scope object created for the outer function when it terminates, but the returned function maintains a reference ***back*** to that scope object
- As a result, the scope object will not be garbage collected until there are no more references to the ***function object*** that the ***outer function returned***
- A **closure** is the combination of a function and the scope object in which it was created

Resources

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

- Secrets of the JavaScript Ninja, by John Resig and Bear Bibeault

- http://blog.taylormcgann.com/2014/01/15/invocation-patterns-javascript/

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures