JavaScript OOP

IMY 220 ● Lecture 6

Functions as variables

Function expression:

```
var hello = function(who) {
 alert("Hello "+who);
}
hello("world");
```

Function declaration:

```
function hello(who) {
       alert("Hello "+who);
}
hello("world");
```

This construct means that the function will follow the same rules and abilities as any other variable in JavaScript.

It will have the same scope.

It can be passed to other functions just like any other variable.

Passing functions to functions

Declaring the function hello as a variable means we can pass it as a variable.

```
var say = function(what) {
      what('say function');
}

var hello = function(who) {
      alert('Hello ' + who); //outputs "Hello say function"
}
say(hello);
```

Anything in JavaScript that can hold a **value** can hold a **function**.

Passing functions to functions

There are two ways to use a **function as an argument**:

- You can pass a pointer to the function itself;
- You can execute the function in the call and pass the return value of the function.

The example above passes a pointer of the hello function to the variable what in the say function. what then becomes a pointer to hello.

Passing functions to functions

If you've ever created an event in JS, you've probably already used the "**pass as pointer**" concept --> all the event handlers want a pointer to a function to call when they're tripped.

```
function doSomething() {
        alert('you pushed my button!');
}
document.onclick = doSomething;
```

By using just doSomething instead of doSomething(), you pass a pointer to the onclick event handler.

Passing functions to functions

Using doSomething(), will still work as long as doSomething returns a pointer to a function.

```
function pushedMyButton() {
     alert('you pushed my button!');
}

function doSomething() {
     return pushedMyButton;
     // return a pointer to the pushedMyButton function
}

document.onclick = doSomething();
```

Passing functions to functions

The parenthesis tell JS that the function needs to be executed, and not just passed as a pointer.

doSomething is executed, which calls another function that passes back a pointer so the event will still work.

Adding the parenthesis means the function will be executed and the return value will be assigned.

Leaving out the parenthesis means you are passing the function as a pointer.

Passing functions to functions

```
var test = function() {
     return "This is a String";
}
var testCopy = test;
// testCopy is a pointer to the test function()

var testStr = test();
// testStr contains "This is a string"

var testing = testCopy();
// testing contains "This is a string"
```

Closures

Consider the following example:

```
function greet(name){
   var greetingText = "Hello there ";

   function makeGreeting(){
      return greetingText + name;
   }
   return makeGreeting();
}
console.log(greet("Diffie"));
```

Closures

At first glance, you might assume that once greet has finished executing, greetingText should no longer be accessible

However, when we call greet at the bottom, we're actually calling a reference to makeGreeting which, in JS, still has access to any variables that were in scope while we were declaring the inner function

In JS, this is known as creating a *closure*

Closures

“A closure is the combination of a function and the lexical environment within which that function was declared.”

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

Thus, when we call greet, we create a closure which includes makeGreeting and any variables makeGreeting has access to

This ensures that we are able to use any functions that a parent function may return, including even more nested functions (next slide)

Closures

```
function greet(name){
   var text1 = "Hello ";

   function makeGreeting(){
      var text2 = "there ";

      function makeGreeting2(){
          return text1 + text2 + name;
      }
      return makeGreeting2();
   }
   return makeGreeting();
}
console.log(greet("Diffie"));
```

Functions as Objects

All functions in JavaScript are objects

An Array, in JavaScript, exists on **two levels**.

At its **top level** it is an Array:

```
var myArray = [];
myArray[1] = 1;
myArray[2] = 2;
myArray[3] = 3;
```

Functions as Objects

Below that is the **object for that Array** which can be used to store data in its properties and create any methods in addition to built-in methods and properties all Arrays share (like .length, .sort, .concat, etc).

```
var myArray = [];
myArray[1] = 1;
myArray[2] = 2;
myArray[3] = 3;
myArray.one = 'one';
myArray.two = 'two';
myArray.three = 'three';

for(var i = 0; i < myArray.length; i++) {
       document.writeln(myArray[i]);
}
```

Functions as Objects

Functions are the same in JavaScript

- At the top level they are functions with the functionality you built for them.

- But underneath it is also an object that has common properties and methods which you can extend.

Function Objects and Methods

Every function has the following properties:

- arguments – an Array/object containing the arguments passed to the function:
    - arguments.length – the number of arguments in the array.
    - arguments.callee – pointer to the executing function (allows anonymous functions to recurse).
- length – the number of arguments the function was expecting.
- constructor – function pointer to the constructor function.
- prototype – allows the creation of prototypes.

Function Objects and Methods

Every function has the following methods:

- apply - A method that lets you more easily pass function arguments.
- call - Allows you to call a function within a different context.
- toString - Returns the source of the function as a string.

Understanding function arguments

The argument list in your function declaration can be considered a recommendation.

```
var myFunc = function(a, b, c) { }
```

The above function can be called in the following ways:

```
myFunc(1);
myFunc(1, 2);
myFunc(1, 2, 3);
myFunc(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Understanding function arguments

If myFunc(1) is called then b and c will be of type undefined.

```
var myFunc = function(a, b, c) {
       if(!a) {a = 1};
       b = b || 2;
       if(c == undefined) {c = 3;}
       console.log(a + ' ' + b + ' ' + c);
}
myFunc(1);
```

Undefined is a false condition so you can use a boolean to check if the argument was set and then correct it.
It is safer to check for “**undefined**” than for a **boolean** as you can send boolean values as well to the argument list.
Use myFunc.length and arguments.length to check how many arguments you have and how many you need

Mastering the Arguments Array

When you have more arguments in the argument list than the function is supposed to have, they are still available to the user.

```
var myFunc = function(){
     document.writeln(arguments.length + '<br>');
     // displays 10

     for(i = 0; i < arguments.length; i++){
          document.writeln(arguments[i] + ',');
          // displays: 1,2,3,4,5,6,7,8,9,10,
     }
}

myFunc(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Mastering the Arguments Array

You cannot work with the argument list like you work with a user defined array.

You do not have access to the same properties and methods (sort, splice, slice like in a user defined array).

You can use the following solution:

```
var args = Array.prototype.slice.apply(arguments);
```

With this code, args becomes a proper array copy of arguments.

How to pass arguments to another function

If you want to send the arguments of one function, to another function:

```
var otherFunc = function(){
        alert(arguments.length); // Outputs: 1 -- an array
}

var myFunc = function(){
        alert(arguments.length); // Outputs: 10
        otherFunc(arguments);
}
myFunc(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

The above will send the arguments array, not the arguments list.

How to pass arguments to another function

```
var otherFunc = function(){
        alert(arguments.length); // Outputs: 10
}

var myFunc = function(){
        alert(arguments.length); // Outputs: 10
        otherFunc.apply(this, arguments);
}

myFunc(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Above we use a function called apply that will apply the arguments array as an arguments list in the new function.

Recursion in anonymous functions

Anonymous factorial function

How do we call a function without a name?

arguments.callee contains a pointer to the current executing function which means an anonymous function can call itself.

```
alert((function(n){
        if(n <= 1){
                return 1;
        } else {
                return n * arguments.callee(n - 1);
        }
})(10));
```

Hoisting

JavaScript puts variable and function declarations into memory during the compile phase. This is called hoisting.

This can lead to some unexpected behaviour with regards to how variables are declared and used, for example:

```
console.log(num);
var num;
```

This “should” give an error, since the variable is only declared after it is used, but instead it gives undefined

Hoisting

This is because the variable declaration is hoisted, i.e. it is put into memory before executing any code that may use that declaration. For that reason, the following piece of code…

```
console.log(num);  // Output: undefined
var num = 5;
```

…is implicitly understood as

```
var num;
console.log(num);
num = 5;
```

This example also illustrates that hoisting only applies to declarations, not initialisations

Hoisting

The same applies to function *declarations*, which allows us to use functions before they are declared

```
console.log(sayHello("Diffie"));

function sayHello(name){
     return "Hello " + name;
}
```

Function declarations are thus hoisted before the code executes

Hoisting

The same, however, does not apply to function *expressions*, i.e. functions assigned to variables using the var keyword

```
console.log(sayHello("Diffie"));

var sayHello = function(name){
     return "Hello " + name;
}
// This code gives an error
```

Function expressions only load when the line of code which defines the function is reached

Arrays

Create new Array

```
    var names = [];
```

Access the array elements

```
    document.write(names[0]);
```

- Prints the first element in the array

Add the names to the array

```
    names[0] = "Jake";
    names[1] = "Amy";
    names[2] = "Rosa";
    names[3] = "Gina";
    names[4] = "Terry";
```

Arrays

Output the array

```
for(var x = 0; x < 5; x++) {
   alert(names[x]);
}
```

Assign values to array

```
for(var y = 0; y < 5; y++) {
   names[y] = prompt('Enter a Name!',' ');
}
```

The Window.prompt() displays a dialog with an optional message prompting the user to input some text.

Arrays

Associative arrays

```
names["one"] = "Raymond";
names["two"] = "Jake";
names["three"] = "Charles";
names["four"] = "Gina";
names["five"] = "Terry";
```

Arrays

JavaScript has a bunch of built-in functions that simplify dealing with arrays:

- concat()
- forEach()
- map()
- filter()
- every()
- splice()
- slice()
- reduce()
- etc.

Arrays

"The concat() method is used to merge two or more arrays. This method does not change the existing arrays, but instead returns a new array."

(The code below creates a new array from two existing arrays)

```
var nums1 = [1, 2, 3];
var nums2 = [4, 5, 6];
var arr = nums1.concat(nums2);

console.log(arr.toString());
// Output: 1, 2, 3, 4, 5, 6
```

Arrays

“The forEach() method executes a provided function once for each array element.”

(The code below adds all elements from arr to string, separated by a space)

```
var arr = [1, 2, 3];
var str = "";

arr.forEach(function(element){
   str += element + " ";
});

console.log(str);
// Output: 1 2 3
```

Arrays

Most array functions also provide the index as a second function parameter

```
var names = ["Jake", "Amy", "Rosa"];
var str = "";

names.forEach(function(element, index){
   str += index + ": " + element + ", ";
});

console.log(str);
// Output: 0: Jake, 1: Amy, 2:Rosa,
```

Arrays

“The filter() method creates a new array with all elements that pass the test implemented by the provided function.”

(The code below creates a new array from all elements from arr that are larger than 3)

```
var arr = [1, 2, 3, 4, 5];
var arr2;

arr2 = arr.filter(function(element){
   return element > 3;
});

console.log(arr2.toString());
// Output: 4, 5
```

Arrays

"The every() method tests whether all elements in the array pass the test implemented by the provided function" (and returns true/false)

(The code below tests whether all elements in an array are less than 4)

```
var arr = [1, 2, 3];
var smaller;

smaller = arr.every(function(element){
   return element < 4;
});

console.log(smaller);
// Output: true
```

```
var arr = [1, 2, 3, 6];
var smaller;

smaller = arr.every(function(element){
   return element < 4;
});

console.log(smaller);
// Output: false
```

Arrays

“The splice() method changes the contents of an array by removing existing elements and/or adding new elements.”

The function is called on the array to be modified and does not return a new array

The function parameters are as follows:

```
arrayName.splice(start, deleteCount, item1, item2, …);
```

Arrays

The following code starts at index 2, removes 1 item and inserts another element at the start index

```
var names = ["Gina", "Jake", "Amy", "Rosa"];
names.splice(2, 1, "Charles");
console.log(names.toString());
// Output: Gina,Jake,Charles,Rosa
```

However, you can provide any number of elements to add at the start index

```
var names = ["Gina", "Jake", "Amy", "Rosa"];
names.splice(2, 1, "Charles", "Terry", "Raymond");
console.log(names.toString());
// Output: Gina,Jake,Charles,Terry,Raymond,Rosa
```

Arrays

“The slice() method returns a shallow copy of a portion of an array into a new array object selected from begin to end (end not included). The original array will not be modified.”

The function parameters are as follows:

```
slicedArrayName = arrayName.slice(start, end);
```

If end is omitted, the array is copied from start to the last element of the array

Arrays

The following code copies the values from names, starting at index 1 up until before index 3, into a new array

```
var names = ["Gina", "Jake", "Amy", "Rosa", "Charles"];
var names2 = names.slice(1, 3);
console.log(names2.toString());
// Output: Jake,Amy
```

If the second parameter is omitted, the rest of the array, starting at the start index, is copied

```
var names = ["Gina", "Jake", "Amy", "Rosa", "Charles"];
var names2 = names.slice(1);
console.log(names2.toString());
// Output: Jake,Amy,Rosa,Charles
```

Arrays

While the slice() function copies values, like strings, numbers, etc., directly, it only copies references to objects (i.e. shallow copy)

```
var person1 = {name: "Jake"};
var person2 = {name: "Amy"};
var names = [person1, person2];
var names2 = names.slice(1);
console.log(names2[0].name);
// Output: Amy

person2.name = "Charles";
console.log(names2[0].name);
// Output: Charles
```

Arrays

The map() function loops through each array element, performs some change on it, and returns a new array with the altered elements

```
var arr = [1, 2, 3];
var arr2;

arr2 = arr.map(function(element, index){
     return element + 1;
});

console.log(arr2.toString());
// Output: 2, 3, 4
```

Arrays

“The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.”

(The code below returns the sum of all values in an array)

```
var nums = [1, 2, 3, 4];
var sum;

sum = nums.reduce(function(accumulator, currElement){
   return accumulator + currElement;
});
console.log(sum);
// Output: 10
```

Arrays

In the first iteration, accumulator refers to the first element of the array and currElement to the second. So, using the previous example:

Iteration: 1	accumulator: arr[0] -> 1	currElement: arr[1] -> 2	return-value: 3

After that accumulator holds on to the value returned from the value returned from the previous iteration while the value of currElement always refers to the next element of the array:

Iteration: 2	accumulator: 3	currElement: arr[2] -> 3	return-value: 6
Iteration: 3	accumulator: 6	currElement: arr[3] -> 4	return-value: 10
End of array reached. return-value: 10			

Arrays

Apart from the function parameter, you can also provide an initial value as a second parameter to the function.

```
var nums = [1, 2, 3, 4];
var sum;

sum = nums.reduce(function(accumulator, currElement){
   return accumulator + currElement;
}, 10);
console.log(sum);
// Output: 20
```

Arrays

When an initial value is provided, the function executes the same way, except that an extra iteration is added in which accumulator refers to the initial value and currValue refers to arr[0]

Iteration: 1	accumulator: initial value -> 10	currElement: arr[0] -> 1	return value: 11
Iteration: 2	accumulator: 11	currElement: arr[1] -> 2	return value: 13
Iteration: 3	accumulator: 13	currElement: arr[2] -> 3	return-value: 16
Iteration: 4	accumulator: 16	currElement: arr[3] -> 4	return-value: 20
End of array reached. return-value: 20			

Arrays

Many more useful Array functions

You can read more about JS Arrays and find details about all the built-in Array functions here: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Arrays

Since many functions return arrays, you can chain Array functions to perform more complex functions

```
var names1 = ["Jake", "Amy", "Charles"];
var names2 = ["Scully", "Hitchcock", "Gina"];

var arr = names1
          .concat(names2)
          .filter(function(element){
               return element.length < 5;
          }).map(function(element){
               return "Yass " + element + "!";
          });

console.log(arr.toString());
Output: ???
```

Extra reading

- Extra reading: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
- Useful tool for debugging JS: https://validatejavascript.com/

References

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions
- http://nefariousdesigns.co.uk/object-oriented-javascript.html
- http://www.w3.org/wiki/Objects_in_JavaScript
- http://www.sitepoint.com/oriented-programming-1-4/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
- https://developer.mozilla.org/en-US/docs/Glossary/Hoisting
- https://developer.mozilla.org/en-US/docs/web/JavaScript/Reference/Operators/function