# Functional JavaScript

IMY 220 ● Lecture 7

# 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"));
```

# Lambda

Lambda is the concept of passing functions as arguments to other functions.

Lambda expression = using a function as an argument in a call to another function.

# Anonymous functions

The event handler (onmousedown) calls the anonymous function that pops up an alert.

```
document.onmousedown = function(){
        alert("You pressed the mouse button");
}
```

This function is **anonymous** because we didn't declare a name for it, we can't call it later unless we somehow go through onmousedown.

If the function does not have a name, then the function is much closer to data than to the traditional function.

The function still works, but looking at a function as a piece of active data will help understand how to apply this technology.

# Self-invoking functions

```
var x = (function() {return 5 + 6})();

document.writeln(typeof x + '<br>');
// Outputs: Number

document.writeln(x);
// Outputs: 11;
```

This is an anonymous self-invoking function.

# Self-invoking functions

The function is wrapped in parenthesis:

```
var x = (function () {return 5 + 6})();
```

and the argument parenthesis are also added at the end:

```
var x = (function () {return 5 + 6})();
```

JavaScript executes the function and assigns the return value to x.

The function exists only long enough for it to assign the value to x.

x is not a function because we invoked the anonymous function inside the parenthesis with a set of argument parenthesis

When checking typeof x we get **Number**, not function

# Self-invoking functions

You can use self-invoking functions (lambda) do to some light processing before sending the data along to another function.

```
function doAlert(msg) {
        alert(msg);
}

doAlert( (function(animal1, animal2) {
      return animal1 + ' loves ' + animal2;
} ) ('cat', 'dog') );
```

The function doAlert is an traditional function but is invoked with an **anonymous self-invoking function as its msg argument**.

- msg will be the returned value of the anonymous function.

The self-invoking function accepts arguments.

The function gets the two arguments from the trailing parenthesis.

# Self-invoking functions

```
function doDisplay(msg) {
        document.writeln(msg + '<br>');
}

for(var i = 0; i < 10; i++) {
      doDisplay( (function(num1, num2) {
            return num1 + ' + ' + num2 + ' = ' + (num1 + num2);
      } ) (i, i + 5) );
}
```

Loops the function doDisplay 10 times.

The anonymous, self-invoking function sets up the string to be displayed by using arguments from the trailing parenthesis.

# Self-invoking functions

```
function doDisplay(msg) {
        document.writeln(msg + '<br>');
}

function buildString(num1, num2) {
        return num1 + ' + ' + num2 + ' = ' + (num1 + num2);
}

for(var i = 0; i < 10; i++) {
        doDisplay( (buildString) (i, i + 5) );
}
```

Named, self-invoking function.

A modification of the previous example where the anonymous function has been changed to be a regular function named buildString

# Self-invoking functions

```
function doDisplay(msg) {
        document.writeln(msg + '<br>');
}

function buildString(num1, num2) {
        return num1 + ' + ' + num2 + ' = ' + (num1 + num2);
}

for(var i = 0; i < 10; i++) {
        doDisplay( buildString(i, i + 5) );
}
```

The traditional function call

# Self-invoking functions

One common use of self-invoking functions is to control variable scope. Since variables are scoped to functions, declaring variables inside self-invoking functions lets you declare variables without polluting the global scope

```
var importantNumber = 1;
var importantString = "";

(function(){
   var importantNumber = 42;
   importantString += "The answer is " + importantNumber;
})();

console.log(importantNumber);  // 1
console.log(importantString);  // The answer is 42
```

# Understanding Function Pointers

When you create a function, that function occupies a section of memory.

Only the pointer to the functions location exists on the JavaScript heap.

When you pass a pointer to the function, you pass a pointer to the **location**, not the name of the function.

# Understanding Function Pointers

```
var originalFunction = function(){
       alert('hello world');
}
var copiedFunction = originalFunction;

var originalFunction = function(){
       alert('goodbye world');
}
copiedFunction();   // outputs Hello World.
```

copiedFunction points to the "**hello world**" function.

When originalFunction is changed to say "**goodbye world**", invoking copiedFunction will still provide us with "**hello world**".

It still points to the original function, where originalFunction now points to the new location at the top of the JavaScript heap

# Understanding Function Scope

JavaScript functions have the same scope as variables in the language.

Functions defined in the root level of the **<script>** tag are global, accessible by every function, object and sub function regardless of their location in the code.

Variables in JavaScript have *function scope*, not block scope:

- Variables defined in a function are private to that function and its sub function.

```
// global, can be accessed anywhere.
var globalFunction = function(){
   alert('hello world');
}

// global, can be accessed anywhere.
var containerFunction = function(){
   // private--global to containerFunction
   var subFunction = function(){
      alert("I'm Private");
      globalFunction();
      // We can access global Function here 2 levels deep.
   }

   globalFunction();
   // We can access global Function here 1 level deep.
   subFunction();
   // We can access subFunction inside containerFunction.
}

containerFunction();
subFunction();
// This produces an error because
// subfunction() only exists inside containerFunction
```

# Understanding Function Scope

The exception:

If you declare a variable within a function WITHOUT the var keyword, that variable becomes global in scope.

(NB: Bad practice)

In the example above, leaving out the var keyword will result in subfunction not giving an error when it is called.

# Understanding Function Scope

Hoisting also works within function scope. Functions declared inside other functions are hoisted inside their parent functions, but not globally.

```
console.log(sayHello("Diffie"));
// Output: Hello Diffie
console.log(makeMessage("Diffie"));
// Reference error: makeMessage is not defined
// (Since makeMessage is scoped to sayHello)

function sayHello(name){
        var msg = makeMessage(name);
        // makeMessage is hoisted inside sayHello

        function makeMessage(name){
                return "Hello " + name;
        }

        return msg;
}
```

# 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));
```

# References


- http://www.hunlock.com/blogs/Functional_Javascript
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures