Functions in JavaScript

Lecture 7
Chapter 8 in JavaScript text

Functions

- The way JavaScript handles functions is one of its most powerful features
- Its also where there are significant departures from C++ and other languages you are familiar with
- The *style* of programming with functions is also vastly different

In JavaScript, **functions are objects**. They have **literal representations**. They are frequently **anonymous**. They are constantly **passed to other functions**!

Function Mechanics

- Functions *can* have names
  - the usual naming rules
- Functions can have parameters
- Functions have an **invocation context**
  - We'll talk more about this shortly - its very important.
  - The invocation context is what makes JavaScript functions **closures**.
- Functions can return data, but they do not have return **types**.

Defining functions

```
function myfunction  (a, b, c) {
    console.log(a);
    console.log(b);
    console.log(c);
    return “something”;
}
```

- Functions can have any number of parameters.
- They can create local variables.

Assigning functions

Variables are often used to ***store*** functions

```
var f = myFunction;
var squared = function (x) { return x * x; }
```

- Here we see the **literal** notation of a functions - its anonymous.
- They can be assigned as properties of objects (and array elements)

```
var math = { pi: 3.1415 }
math.squared = function (x) { return x * x; }
```

Calling functions

```
var value = squared(9);
var pi_squared = math.squared(math.pi);
```

When calling a function, you are not obligated to provide all the arguments

```
var value = squared();
function squared(x) {
   if ( x === undefined) return 0;
   return x * x;
}
```

Function arguments

You can access function arguments (optionally) through a built in array defined for each function - called **arguments**.

```
function f(a, b, c) {
   console.log(arguments[0]);
   console.log(a);
}
f(1, 2, 3);
```

Additional arguments

There is nothing stopping someone from calling your function with **more** values than defined

```
function f (a, b) {
    if ( arguments.length > 2 ) {
        var sum = 0;
        for ( i = 0; i < arguments.length; i++ )
            sum += arguments[i];
        return sum;
    }
    else {
        return a + b;
    }
}
```

Certainly this is a nonsense function!

Why?

There are some really nice uses for this..

```
function max() {
    var max = Number.NEGATIVE_INFINITY;
    for ( i = 0; i < arguments.length; i++ )
        if ( arguments[i] > max ) max = arguments[i];
    return max;
}

console.log(max ( 1, 2, 3, 4, 5));
console.log(max (6, 5));
```

Passing objects

When you have many parameters its often better to use an **object**.

```
function printName(person) {
        var n = [ ];
        if ( person.first !== undefined )        n.push(person.first);
        if ( person.middle !== undefined )   n.push(person.middle);
        if ( person.last !== undefined ) n.push(person.middle)
        return n.joint(“*”);
}
```

This is especially useful when you want a function to take a bunch of optional parameters

Function assignment

```
var operators = {
    add :    function (x, y) { return x + y; },
    sub :    function (x, y) { return x - y; },
    mult:    function (x, y) { return x * y; },
    div :    function (x, y) { return x / y; }m
    pow :    Math.pow
}
console.log( operators.div(4, 2) );
operators.div = operators.mult;
console.log( operators.div(4, 2) );
```

Functions as parameters

```
function printOpResult(x, y, f) {
   var result = f(x, y);
   console.log(result);
}

printOpResult(2, 3, operators.div);
printOpResult(5, 3,
   function (x, y) {
      return (x - y)* (x - y);
   }
);
```

Anonymous function

Functional Programming

- The typical JavaScript functions will have many anonymous functions, and frequently passes functions as arguments to others.

```
var a = [4, 1, 2, 6, 5, 3];
var r = a.sort ( function (x, y) {
     var x_even = x % 2 == 0;
     var y_even = y % 2 == 0;
     if ( x_even && !y_even ) {
          return 1;
     }
     else if ( !x_even && y_even ) {
          return -1;
     }
     else {
          return x - y;
     }
} ).join(".");
console.log(r);
```

The sort function can accept a function to act as a comparator

Return < 0 if x is “less” than y
Return > 0 if x is “greater” than y

You can define “less” and “greater” however you want of course!

Iterating arrays

A very powerful “functional” feature of arrays is the forEach function.

It accepts a function to call on each element.

```
var a = [1, 2, 3, 4, 5];
a.forEach(function (x) { console.log( x * x); })
```

Filtering arrays

You can also filter arrays using customized functions.

```
var a = [1, 2, 3, 4, 5, 6, 7];
var evens = a.filter(
   function (x) { return x % 2 ==0;});

console.log(evens.join());
```

map operation on arrays

If you want to return an array representing a transformation of another - use the map function

```
var a = [1, 2, 3, 4];
var b = a.map( function (x) { return x * x; });
console.log(b.join());
```

“Invocation Context”

- Functions have scope attached to them
- It works differently than C++ though…

```
function makeFunction(x) {
    var y = 10;
    return  function () {
                return x + y;
            }
}
f = makeFunction(6);
console.log(f());
```

Notice that the “local” y variable some how is usable well after it went “out of scope”...

Closures

- Functions are closures.
- Their scope (variables in their scope) are carried with them.

If you think too hard about this (in the C/C++ way) you'll confuse the issue. Its actually very simple to use and work with!

Read the inset on pg 182 of the JavaScript text book for a good explanation on how this actually implemented in the language

In short - there is no call stack in JavaScript, a function's scope is just an object that is managed by JavaScript itself.

Functions….

There's more to functions, objects, and even arrays. However we now have enough to start implementing server-side logic in Node.js

Please make sure you read the JavaScript textbook through **Chapter 9**. We will cover most of the skipped sections as they appear in future lectures.

Up next

- Learn to get data from the user - HTML forms.

- And then we'll see how we put dynamic data into the HTML we send to the browser - using EJS.