Functions
and
Scope

In JavaScript

Agenda

- Functions
- Scope
- Hoisting

FUNCTIONS

Function Declaration

- A Function Declaration defines a named function variable without requiring variable assignment
- Function Declarations occur as standalone statements

- ECMA 5 defines the syntax as:
  function *Identifier* (*FormalParameterList*$_{opt}$) { *FunctionBody* }

- E.g.:

```
function bar() {
    return 3;
}

console.log(bar()); //3
```

Pure Functions

- Always return the same value when given the same arguments, and never have side effects

```
function add(a, b) {
  return a + b;
}

console.log(add(2, 3));  // 5
```

- The names of the arguments of a function are available as variables inside it
- Pure functions are easy to think about, and they are easy to re-use
- A return statement without an expression after it will cause the function to return undefined

Functions with Side Effects

- Functions with side effects do not have to contain a return statement
  - If no return statement is encountered, the function returns undefined

```
function yell(message) {
  alert(message + "!!");
}
console.log(yell("Wow"));
```

```
var count = 0;
function inc(step) {
  count += step;
  return count;
}
```

Note:
In JavaScript function names don't start with a capital letter

Calling a Function

- The formal parameters are type less so you can pass any kind of value to a function
  - But the function may not work with all kind of values!
- And the number of arguments do not have to match the number of formal parameters

```
function add(a, b) {
  return a + b;
}
console.log(add('2', '3'));      // 23
console.log(add(true, false));   // 1
console.log(add(4));             // NaN
console.log(add(5,6,7,8));      // 11
```

The Arguments Object

- Within a function, the arguments may also be accessed through the arguments object
  - This provides access to all arguments using indices

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

console.log(add(1,2,3));  // 6
```

Function Expression

- When the function keyword is used in a place where an expression is expected, it is treated as an expression producing a function value
  - The function name is optional

```
var add = function addFunc(a, b) {
    return a + b;
};
console.log(add(5, 5));
```

```
var add = function (a, b) {
    return a + b;
};
console.log(add(5, 5));
console.log(typeof add);
// prints function
```

- ECMA 5 defines the syntax as:
  function *Identifier*$_{opt}$ (*FormalParameterList*$_{opt}$) { *FunctionBody* }

Self Invoking Function Expression

- Function Expressions must not start with “function”
  - therefore the parentheses around the self invoking function expression

```
(function sayHello() {
    alert("hello!");
})();
```

SCOPE

Aka environment

Functions in JavaScript have lexical scoping

Functions Form a Local Scope

- The variables in this local environment are only visible to the code inside the function
- The variables are not accessible from outside of the function

```
var str = "top-level";

function printVariable() {
    console.log("inside printVariable, str holds '" + str + "'.");
}

function test() {
    var str = "local";
    console.log ("inside test, str holds '"+ str + "'.");
    printVariable();
}

test();
```

```
inside test, str holds 'local'.
inside printVariable, str holds 'top-level'.
```

Nested Functions scope

- When a function is defined *inside* another function, its local environment will be based on the local environment that surrounds it instead of the top-level environment

```
var variable = "top-level";

function parentFunction() {
    var variable = "local";

    function childFunction() {
        console.log(variable);
    }

    childFunction();
}
parentFunction();
```

```
local
```

“Static” Variables

- Any variables used in a function which are not explicitly defined as var are assumed to belong to an outer scope, possibly to the Global Object

```
function printVariable() {
    console.log("inside printVariable, myStr holds '" + myStr +
"'.");
}

function test() {
    myStr = "Static";
    console.log("inside test, myStr holds '"+ myStr + "'.");
    printVariable();
}

test();
```

```
inside test, myStr holds 'Static'.
inside printVariable, myStr holds 'Static'.
```

Always use var or let!

- There is no namespaces or class scopes in JavaScript, so all static variables goes into the global scope (environment)
- And if you by accident use a variable name already there you will overwrite it
  - This way you can block for access to important functionality!

```
function foo() {
    a = 2;
    b = "Hello";

    return 7 * a;
}

console.log(foo());
console.log(a);
console.log(b);
```

Don't do this!

Blocks

- Unlike the other languages in the C-family, a block of code (between braces) does NOT produces a new local environment
  - Functions are the only things that create a new scope

```
var something = 1;
{
    var something = 2;
    console.log ("Inside: " + something);
}
console.log("Outside: " + something);
```

```
Inside: 2
Outside: 2
```

  - Unless you use **let**

```
let nothing = 1;
{
    let nothing = 2;
    console.log ("Inside: " + nothing);
}
console.log("Outside: " + nothing);
```

```
Inside: 2
Outside: 1
```

Closure

- A function defined inside another function retains access to the environment that existed in that function at the point when it was defined

```
var variable = "top-level";

function parentFunction() {
    var variable = "local";
    function childFunction() {
        console.log(variable);
    }
    return childFunction;
}

var child = parentFunction();
child();
```

local

Closure

- A function defined inside another function retains access to the environment that existed in that function at the point when it was defined

```
function makeAddFunction(amount) {
    function add(number) {
        return number + amount;
    }
    return add;
}

var addTwo = makeAddFunction(2);
var addFive = makeAddFunction(5);
console.log(addTwo(1) + addFive(1));
```

9

HOISTING

Question 1

- What is alerted?
  - 3, 8 or TypeError?

```
function foo() {
    function bar() {
        return 3;
    }
    return bar();

    function bar() {
        return 8;
    }
}
alert(foo());
```

Question 2

- What is alerted?
  - 3, 8 or TypeError?

```
function foo() {
    var bar = function() {
        return 3;
    }
    return bar();

    var bar = function() {
        return 8;
    }
}
alert(foo());
```

Question 3

- What is alerted?
  - 3, 8 or TypeError?

```
alert(foo());

function foo() {
    var bar = function() {
        return 3;
    }
    return bar();

    var bar = function() {
        return 8;
    }
}
```

Question 4

- What is alerted?
  - 3, 8 or TypeError?

```
function foo() {
    return bar();

    var bar = function() {
        return 3;
    }

    var bar = function() {
        return 8;
    }
}

alert(foo());
```

What's Hoisting?

- Function declarations and function variables are always moved ('hoisted') to the top of their JavaScript scope by the JavaScript interpreter

Question 1 Revisited

- When a function declaration is hoisted the entire function body is lifted with it
- So after the interpreter has finished with the code in Question 1 it runs more like this:

```
function foo() {
    function bar() {
        return 3;
    }
    return bar();

    function bar() {
        return 8;
    }
}
alert(foo());
```

```
function foo() {
    function bar() {
        return 3;
    }
    function bar() {
        return 8;
    }

    return bar();
}
alert(foo());
```

Question 2 Revisited

- The left hand side (*var bar*) is a Variable Declaration
- Variable Declarations get hoisted
  - but their Assignment Expressions don't
- So when **bar** is hoisted the interpreter initially sets

  var bar = undefined

```
function foo() {
    //a declaration for each function expression
    var bar = undefined;
    var bar = undefined;
    //first Function Expression is executed
    bar = function () {
        return 3;
    }
    // Function created by first Function Expression is invoked
    return bar();
    // second Function Expression unreachable
} alert(foo());
```

Question 3 Revisited

- When a function declaration is hoisted the entire function body is lifted with it

```
alert(foo());

function foo() {
    var bar = function() {
        return 3;
    }
    return bar();

    var bar = function() {
        return 8;
    }
}
```

```
//Hoisting
function foo() {
    //Hoisting
    var bar = undefined;
    var bar = undefined;
    bar = function () {
        return 3;
    }
    return bar();
    // second expression unreachable
}
alert(foo());
```

Question 4 Revisited

- var bar is not a function when it is returned.

```
function foo() {
    return bar();

    var bar = function() {
        return 3;
    }

    var bar = function() {
        return 8;
    }
}

alert(foo());
```

```
function foo() {
    //Hoisting
    var bar = undefined;
    var bar = undefined;

    return bar();
    // Both expressions unreachable
}

alert(foo());
```

References and Links

- **Eloquent JavaScript** by Marijn Haverbeke
  http://eloquentjavascript.net
- The JavaScript guru: **Douglas Crockford**'s blog
  http://www.crockford.com/javascript/

- Function Declarations vs. Function Expressions (Hoisting)
  http://javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/