Functions, Scope and Closures

functions: an object that does something

- anatomy
- arguments and parameters
- constructors
- object methods
- function methods

anatomy

```
var multiply = function (a, b) {
    return a * b;
};
multiply(3, 4);
```

anatomy

```
// function declaration
function setStatus(status) {
    this.status = status;
}

// passing a function as a value
var myArr = [1, 2, 3, 4];
myArr.forEach(function (val) {
    console.log(val);
});

// function as callback
$('body').on('click', function (e) {
    console.log('clicked on body');
});
```

arguments and parameters:

- Parameters are the variables declared in a function definition that appear inside the parens, right next to the function keyword or name. Before function invocation, they have no value.
- Arguments are the values passed in to a function at invocation, and those values are assigned to the function's parameters.

order matters when passing arguments to a function invocation as they are assigned in the order in which the parameters appear

```
var multiply = function (a, b) {
    return a * b;
};
multiply(3, 4, 5); // ?
```

what changes to the function definition are required for the following invocation to produce 30?

```
var multiply = function (a, b) {
    return a * b;
};
multiply(4, 10, 3);
```

arguments: this keyword is a special value inside of functions that provides a list of the arguments passed to the function invocation. This is useful when there is a variable number of arguments.

what changes to the function body could be made for the following function invocation to use all three arguments?

```
var multiply = function (a, b) {
    return a * b;
};
multiply(3, 12, 2);
```

arguments is not an array, it is an array-like object, meaning you would need to convert it to an array to use array methods on it:

```
var args = Array.prototype.slice.call(arguments);
args.forEach(function (val) {
    console.log(val);
});
```

Constructors: a function that returns an object - used with the new keyword which, when used, creates a new object and binds "this" to that object, returning it from the function - any function can be a constructor.

```
var Student = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
};
var student = new Student('Joe', 'Smith');
student.firstName; // 'Joe'
```

Object methods: property values that are functions - using the "this" keyword inside the method refers to the object to which the method belongs

```
var student = {
    firstName: 'Joe',
    lastName: 'Smith',
    getFullName: function () {
        return this.firstName + ' ' + this.lastName;
    }
};
```

Function methods: methods that exist on every function object that can be used for such purposes as changing what is referenced by "this"

```
var student = {
    firstName: 'Joe',
    lastName: 'Smith',
    getFullName: function () {
        return this.firstName + ' ' + this.lastName;
    }
};

var person = {
    firstName: 'Samantha',
    lastName: 'Brooks'
};

// call
student.getFullName.call(person);
```

Function methods: the call method takes a comma-separated list of arguments, while it's sibling method, apply, takes an array of arguments

```
var person = {
    firstName: 'Joe',
    lastName: 'Smith',
    assignRoles: function (assignedDay, role) {
        console.log('On ' + assignedDay + this.firstName + ' ' +
                this.lastName + ' will be assigned a role of ' + role);
    }
};

var anotherPerson = {
    firstName: 'Steve',
    lastName: 'Johnson'
};

// call
person.assignRoles.call(anotherPerson, 'Tuesday', 'file clerk');

// apply
person.assignRoles.apply(anotherPerson, ['Tuesday', 'file clerk']);
```

scope

Scope is where to look for things, i.e. where variables can be accessed. The current version of JavaScript only has function scope*

the JavaScript compiler: as the JavaScript compiler enters a function, it will look for declarations inside that scope and recursively process them, placing references for those declarations in appropriate scope slots

```
var foo = 'bar';

function bar() {
    var foo = 'baz';
}

function baz(foo) {
    foo = 'bam';
    bam = 'yay';
}
```

after compilation has established variable references for each scope, how JavaScript behaves at execution time

```
var foo = 'bar';

function bar() {
    var foo = 'baz';

    function baz(foo) {
        foo = 'bam';
        bam = 'yay';
    }
    baz();
}

bar(); // ?
foo; // ?
bam; // ?
baz(); // ?
```

function declarations vs function expressions assigned to variables:

```
// named function expressions: variable name occurs inside function scope
var foo = function bar() {
    var foo = "baz";

    /*
     * function declarations: first term of the statement
     * must be the function keyword
     */
    function baz(foo) {
        foo = bar;
        foo;
    }
    baz();
};

foo();
bar(); // error
```

block scope:

```
var foo;

try {
    foo.length;
} catch (err) {
    console.log(err); // TypeError
}
console.log(err); // ReferenceError
```

lexical scope: compile-time scope - decisions about scope were made when the compiler ran

```
var foo;

try {
    foo.length;
} catch (err) {
    console.log(err); // TypeError
}
console.log(err); // ReferenceError
```

visualize scope as a series of nested bubbles

```
// outside the function is the outer level of scope(bubble) enclosing everythi

// this is the second level of scope(bubble) enclosing the inner scope
function foo() {
    var bar = 'bar';

    // this is the innermost scope(bubble)
    function baz() {
        console.log(bar);
    }
    baz();
}
foo();
```

IIFE: immediately invoked function expression

```
var foo = 'foo';

(function iife() {

    var foo = 'foo2';
    console.log(foo); // 'foo2'
})();

console.log(foo); // 'foo'
```

The enclosing of the function expression allows for the creation of a new scope that doesn't leak a name into the global namespace.

```
var foo = 'foo';

function createScope() {

    var foo = 'foo2';
    console.log(foo);
}
createScope();
console.log(foo);
```

IIFE variations: passing in arguments

```
var foo = 'foo';

(function iife(bar) {

    var foo = bar;
    console.log(foo); // 'foo'
})(foo);
console.log(foo); // 'foo'
```

hoisting: a mental model to describe what, effectively, it appears the compiler is doing

```
a;
b;
var a = b;
var b = 2;
b;
a;
```

hoisting: a mental model to describe what, effectively, it appears the compiler is doing

```
var a;
var b;
a;
b;
a = b;
b = 2;
b;
a;
```

hoisting: function declarations get hoisted, but function expressions do not

```
var a = b();
var c = d();
a;
c;

function b() {
    return c;
}

var d = function d() {
    return b();
};
```

closure

"Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope."

Kyle Simpson - "Scope & Closures"

closure: an example

```
function foo() {
    var bar = 'bar';

    function baz() {
        console.log(bar);
    }

    bam(baz);
}

function bam(cb) {
    cb();
}

foo();
```

closure: an example - returning functions

```
function foo() {
    var bar = 'bar';

    return function fn() {
        console.log(bar);
    };
}

function bam() {
    foo()();
}

bam();
```

closure: another example with callbacks

```
function foo() {
    var bar = 'bar';

    setTimeout(function cb() {
        console.log(bar);
    }, 1000);
}

foo();
```

closure: another example with event handlers - closures are what allow event handlers to work the way in which they do

```
function foo() {
    var bar = 'bar';

    $('.btn').on('click', function handler(e) {
        console.log(bar);
    });
}

foo();
```

closure: shared scope - any function creating a closure over a given scope has access to the same scope as another function

```
function foo() {
    var bar = 0;

    setTimeout(function cb() {
        console.log(bar++);
    }, 100);
    setTimeout(function cb2() {
        console.log(bar++);
    }, 200);
}

foo();
```

closure: nested scope

```
function foo() {
    var bar = 0;

    setTimeout(function cb() {
        var baz = 1;
        console.log(bar++);

        setTimeout(function cb2() {
            console.log(bar + baz);
        }, 200);

    }, 100);

}

foo();
```

closure: loops

```
for (var i = 1; i <= 5; i += 1) {
    setTimeout(function cb() {
        console.log('i: ' + i);
    }, i * 1000);
}
```

closure: loops

```
for (var i = 1; i <= 5; i += 1) {
    (function iife(i) {
        setTimeout(function cb() {
            console.log('i: ' + i);
        }, i * 1000);
    })(i);
}
```

module pattern

module pattern: the most common pattern found using closure

```
var foo = (function iife() {

    var o = { bar: 'bar' };

    return {
        bar: function bar() {
            console.log(o.bar);
        }
    };
})();

foo.bar();
```

module pattern has two primary characteristics:

1. there must be an outer, wrapping function (doesn't have to be an IIFE)
2. there must be one or more functions returned from that function to close over the inner scope

```
var foo = (function iife() {

    // private members
    var o = { bar: 'bar' };

    // public api
    return {
        bar: function bar() {
            console.log(o.bar);
        }
    };
})();

foo.bar();
```

module pattern allows notion of "private" methods

```
var foo = (function iife() {

    // private method
    function __doStuff() {
        // do something you only want done inside the module
    }
})();

foo.bar();
```

modified module pattern

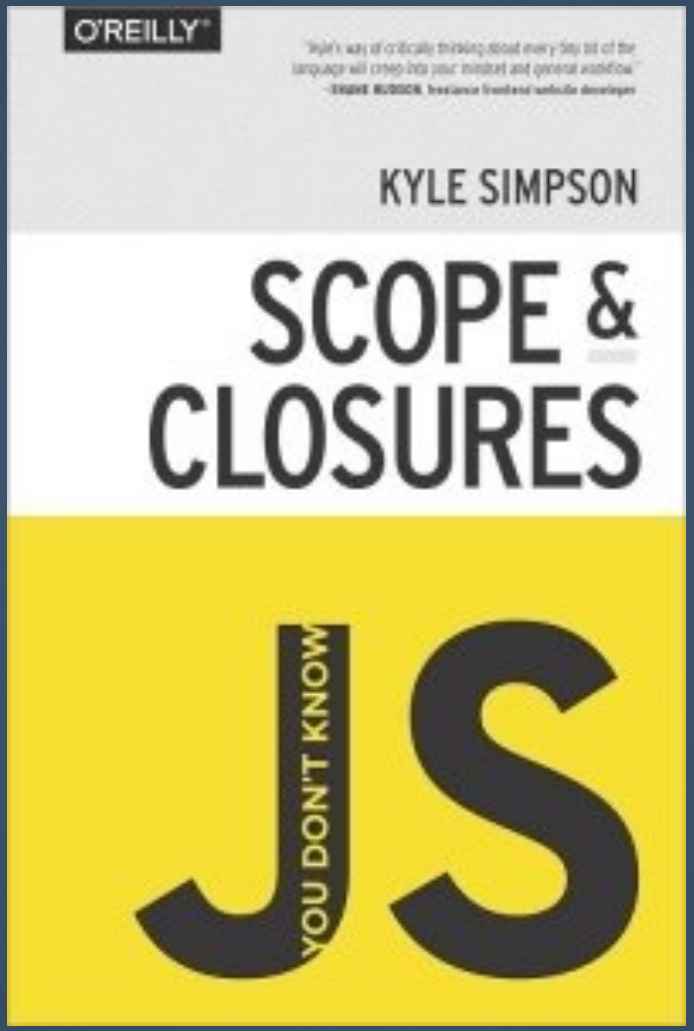
```
var foo = (function iife() {

    // explicit object to easily identify the API
    var publicAPI = {
        bar: function bar() {
            publicAPI.baz();
        },
        baz: function baz() {
            console.log('baz');
        }
    };
    return publicAPI;
})();

foo.bar(); // 'baz'
```

revealing module pattern

```
var foo = (function iife() {

    function bar() {
        console.log('bar');
    }

    function baz() {
        console.log('baz');
    }

    return {
        bar: bar,
        baz: baz
    };
})();

foo.bar(); // 'baz'
```

variant of the module pattern: similar to how something like Require.js works - the library does the function invocation for you as part of the implementation

```
define('foo', function iife() {

    var o = { bar: 'bar'};

    return {
        bar: function bar() {
            console.log(o.bar);
        }
    };
});
```


O'REILLY
KYLE SIMPSON
SCOPE & CLOSURES
YOU DON'T KNOW JS