Intro To JavaScript – Day 4

Functions

```
function addTwo(one, two) {
    return one + two;
}

let addThree = function(one, two, three) {
    return one + two + three;
}

console.log(addTwo(1, 2));
console.log(addThree(1, 2, 3));
```

At first glance, JavaScript functions have a lot in common with what you are probably used to with other functions, but there are a number of differences.

- Functions are objects! This means they can have properties and even methods of their own.
- A function assigned to a *property* of an object is called a *method*.
- Functions can be nested within other functions...which can lead to *closures*.
- The keyword `function` is used to define a function.
- A function may have a name or be anonymous. `addTwo` above is named, the function assigned to `addThree` is anonymous, but assigned to a variable.
- Functions can take parameters, but the keyword `var` is not used with them.
- Functions always return a value, even if you don't use the `return` keyword.
- You can immediately invoke a function after defining it:

  ```
  let cube = (function(x) { return x*x*x; }(3));
  console.log(cube);
  ```

Functions, Objects, & Closures

Let's take a look at making functions properties of an object.

```
let car = {
    position: { x: 0, y: 0 },
    direction: { x: 0.5, y: 0.5},
    speed: 1,
    color: 'rgb(255, 0, 0)',
    moveForward: function() {
        this.position.x += (this.direction.x * this.speed);
        this.position.y += (this.direction.y * this.speed);
    },

    moveBackward: function() {
        this.position.x -= (this.direction.x * this.speed);
        this.position.y -= (this.direction.y * this.speed);
```

```
    },

    report: function() {
        console.log('x: ' + this.position.x + ', y: ' + this.position.y);
    }
};
car.moveForward();
car.report();
car.moveBackward();
car.report();
```

- We are using an object literal to create a `car`.
- The object has value and function properties. Notice that function properties are defined the same way, only using the `function` keyword for their definition.

What if we don't want all those properties to be public and/or modifiable? Let's change our approach to how we define the object, consider the following...

```
function car() {
    let that = {};
    let position = { x: 0, y: 0 },
        direction = { x: 0.5, y: 0.5},
        speed = 1, color = 'rgb(255, 0, 0)';
    that.moveForward = function() {
        position.x += (direction.x * speed);
        position.y += (direction.y * speed);
    };

    that.moveBackward = function() {
        position.x -= (direction.x * speed);
        position.y -= (direction.y * speed);
    };

    that.report = function() {
        console.log('x: ' + position.x + ', y: ' + position.y);
    };

    return that;
}
let myCar = car();
myCar.moveForward();
myCar.report();
myCar.moveBackward();
myCar.report();
```

- Instead of having an object named `car`, we change it to a function named `car`., and that function returns an object that doesn't have a name, but it does have methods and (hidden) properties. `car` is now an object generator (not quite the same thing as a constructor)!
- We define, by convention, a local variable named `that` and assign some function properties to it. Alternatively, could move the functions directly inside the `that` declaration, for example...

```
let that = {
    moveForward: function() {
        position.x += (direction.x * speed);
        position.y += (direction.y * speed);
    }
};
```

- We then define a bunch of properties of the car and use them inside of the functions.
- Finally, `that` is returned from the function. This is what the calling code gets back, and only the functions defined on that are visible, the properties defined inside of `car` become part of the *closure* formed by the that object. `that` hangs onto the properties, but they aren't publicly visible.

The above approaches don't provide a way to specify the properties of the car when it is created, we still desire that. Let's try this another way...

```
function car(spec) {
    let that = {
        moveForward: function() {
            spec.position.x += (spec.direction.x * spec.speed);
            spec.position.y += (spec.direction.y * spec.speed);
        },

        moveBackward: function() {
            spec.position.x -= (spec.direction.x * spec.speed);
            spec.position.y -= (spec.direction.y * spec.speed);
        },

        report: function() {
            console.log('x: ' + spec.position.x + ', y: ' + spec.position.y);
        }
    };

    return that;
}
let myCar = car( {
    position: { x: 0, y: 0},
    direction: {x: 0.5, y: 0.5},
    speed: 1,
    color: 'rgb(255, 0, 0)'
});
myCar.moveForward();
myCar.report();
myCar.moveBackward();
myCar.report();
```

- We now have a single parameter named `spec` into which we can define anything we want. In reality, however, we must define the specified items.
- The various methods capture `spec` as part of their closure, keeping it in scope for as long as `that` is alive. *Isn't that cool!*

Let's finish the day with one more very interesting demonstration. First, let's see how you can test to see if a property is part of an object. Consider the following code...

```
let primes = {1:1 , 2:2, 3:3, 5:5, 7:7, 11:11, 13:13, 17:17};
function inObject(value, object) {
    if (value in object) {
        console.log('yes');
    }
    else {
        console.log('no');
    }
}
inObject(11, primes);
inObject(12, primes);
```

The `in` operator is used to test if a property (a string) is found within an object.

Let's use this and the ability to store properties (in an array-like fashion) on a function to make a really fast fibonacci function. Take a look at this...

```
fibonacci[0] = 1;
fibonacci[1] = 1;
function fibonacci(n) {
    if (!(n in fibonacci)) {
        fibonacci[n] = fibonacci(n - 1) + fibonacci(n - 2);
    }

    return fibonacci[n];
}
console.log(fibonacci(1));
console.log(fibonacci(2));
console.log(fibonacci(3));
console.log(fibonacci(4));
console.log(fibonacci(5));
console.log(fibonacci(6));
console.log(fibonacci(7));
console.log(fibonacci(8));
console.log(fibonacci(9));
console.log(fibonacci(10));
```

We start off by defining the first two fibonacci values, then once the value for a particular fibonacci value is computed, it is stored as part of the function and used in the future. Pretty slick!