Object Properties

We discuss here various issues related to Javascript object properties.

Relevant Links

- Flanagan's book, 6.6-6.8
- Object methods in MDN[1]
- Object.defineProperty in MDN[2]

Notes

An object's properties have a number of customizable "attributes", that are exposed via methods of Object. These attributes are used when we try to set properties via Object.defineProperty or Object,defineProperties or when we pass a second argument to Object.create.

A call to Object.defineProperty would look something like this:

```
Object.defineProperty(obj, prop, desc)
```

where the third argument is a "descriptor" object containing the attributes. There are two types of descriptor objects, "data" and "accessor".

Both types of descriptors are allowed to contain the following two properties:

configurable Boolean, default false. Determines if the property can be configured and deleted.

enumerable Boolean, default false. Whether this property will show up during enumeration (say in a for-in loop).

Data descriptors are meant for properties that simply hold values, and will contain the following keys:

value The initial value. Defaults to undefined.

writable Boolean, determining whether the variable is writable or not.

Accessor descriptors are meant for properties that require getter and setter methods, and will contain the following keys:

get A function f whose return value is what the property returns. A value of undefined would mean there is no getter.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

set A function f(v) used to "set" the value of the property to the value v. A value of undefined will mean there is no setter.

Here is an example of this:

```
var o = {};
Object.defineProperty(o, 'countMe', {
    set: undefined,
    get: (function() {
        var count = 0;
        return function() { count += 1; return count; };
    }())
})
```

Use these features sparingly!

Small get/set Illustration: Temperatures

Here is a small example where these getters and setters might be useful. Say we want to create a "temperature" object that understands both Celsius and Fahrenheit. Here is a way to do it:

```
function f2c(F) { return (F - 32) * 5 / 9; }
function c2f(C) { return C * 9 / 5 + 32; }
function makeTemp() {
    var o = Object.create(null);
    o.C = 0;
    Object.defineProperty(o, "F", {
        configurable: false,
        enumerable: true,
        get: function() { return c2f(this.C); },
        set: function(F) { this.C = f2c(F); return F; }
    });
    return o;
}
```

We are essentially turning "F" into a "virtual property", that instead just sets the corresponding C value appropriately transformed.

You can also use these get/set methods to do some validation.

Stack implementation, again. Locking things down.

Let us look at what was the second version of the prototype-based implementation of stacks we had in previous classes:

```
var Stack = {};
Stack.new = function makeStack() {
    var o = Object.create(Stack._proto);
    o.values = [];
    return o;
};
Stack._proto = {
```

```
    push: function push(el) {
        this.values.push(el);
        return this;
    },
    pop: function pop() {
        if (this.isEmpty()) {
            throw new Error("Attempt to pop from empty stack");
        } else {
            return this.values.pop();
        }
    },
    isEmpty: function isEmpty() {
        return this.values.length === 0;
    }
};
```

This was not a bad implementation, but it had some "flaws". It allows people to change it. We will try to "lock it down". This is not always a good idea, so think carefully before trying to do this.

Here are the "flaws":

1. The _proto property is visible: If someone enumerates the keys in Stack, it will see it. We will want to make it not enumerable, so that it can be accessed only if someone wants to find it.
2. The _proto property is configurable and writeable. Someone can change what prototype object our stack objects will inherit.
3. The object pointed to by the _proto property is editable. This means someone can change how our prototype methods behave, add their own, remove them altogether, and so on.
4. The object pointed to by the _proto property inherits from Object.prototype. We do not really need it to.
5. The values property of individual objects, that contains the array of values in the stack, is enumerable. This should not be visible to the world.
6. The values property is writeable. Someone could simply replace our array with their own.
7. The stack object we return is extensible. Someone could implement "push" and "pop" methods on it, and they will hide our prototype methods.
8. The array stored in the values property is subject to all array methods, so someone could manually add and remove properties. Nothing we can do to protect from that, if we want to use prototypes (though we will discuss a somewhat obscure solution further down).

So let us see how we can address these issues:

1. We will make sure that the _proto property is marked as not enumerable, not configurable, and not writeable.

2. We will freeze[3] the object that _proto points to. We will also create it to inherit from null.
3. We will mark the values property as being not enumerable, not configurable, and not writeable.
4. We will freeze the stack object we return.

Here is an implementation:

```
var Stack = Object.freeze(Object.create(null, {
    new: {
        enumerable: true,
        configurable: false,
        writeable: false,
        value: function makeStack() {
            return Object.freeze(Object.create(Stack._proto, {
                values: {
                    enumerable: false,
                    configurable: false,
                    writeable: false,
                    value: []
                }
            }));
        }
    },
    _proto: {
        enumerable: false,
        configurable: false,
        writeable: false,
        value: Object.freeze({
            push: function push(el) {
                this.values.push(el);
                return this;
            },
            pop: function pop() {
                if (this.isEmpty()) {
                    throw new Error("Attempt to pop from empty stack");
                } else {
                    return this.values.pop();
                }
            },
            isEmpty: function isEmpty() {
                return this.values.length === 0;
            }
        })
    }
}));

Object.keys(Stack);     // Only ["new"]
Stack._proto.f = 2;     // Fails to change _proto
var s1 = Stack.new();
s1.push = 2;            // Fails to create a push property
Object.keys(s1);        // Empty
s1.push(2).push(3);     // These work fine.
```

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

```
s1.pop();
s1.values;              // Shows us what's there
s1.values = [];         // Fails to change the values array.
s1.values.push(4);      // But this works and modifies the array.
```

Optional: Further lockdown

We will describe here a way to lock down even that part of the story, where the values array is accessible to users of our stack. Here is the plan:

- Create a local variable keeping an array called cache, whose elements are the arrays for our stacks. This variable will be visible to the prototype methods, but not to the outside world. We wrap the whole thing in a immediate function invocation to create this local scope.
- When a new stack object is created, an index is stored in the object and a place for it is created in cache, to hold its array of elements. The index stored in the object creates a link between the cache and the object.
- Our stack methods will need to first look the array up based on the index in their this object, then perform the appropriate operation on it.

```
var Stack = (function() {
    var cache, proto;

    cache = [];

    function makeStack() {
        cache.push([]);
        return Object.freeze(Object.create(proto, {
            index: {
                enumerable: false,
                configurable: false,
                writeable: false,
                value: cache.length - 1
            }
        }));
    }

    proto = Object.freeze({
        push: function push(el) {
            cache[this.index].push(el);
            return this;
        },
        pop: function pop() {
            if (this.isEmpty()) {
                throw new Error("Attempt to pop from empty stack");
            } else {
                return cache[this.index].pop();
            }
        },
        isEmpty: function isEmpty() {
            return cache[this.index].length === 0;
```

```
        }
    });

    return Object.freeze(Object.create(null, {
        "new": {
            enumerable: true,
            configurable: false,
            writeable: false,
            value: makeStack
        }
    }));
}());
```

One downside to this pattern, other than its complexity, is that automatic garbage collection cannot clear the cache array for us: If a stack object is garbage-collected, we have no real way of knowing it. We would have to add a manual method for people to call when they no longer need the stack, so we can do some cleanup.