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.

For instance we can define a normal property that is constant as follows:

[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

```
var o = {};
Object.defineProperty(o, 'theAnswer', {
    configurable: false,      // Can be omitted, it's the default
    enumerable: true,
    value: 42,
    writable: false
});
o.theAnswer;        // 42
o.theAnswer = 3;    // No error, but does not actually change the value.
o.theAnswer;        // Still 42
```

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.

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. Every time you try to access the value you get an ever increasing number back

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

Use these features sparingly! It is not an expected behavior.

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. The "C" property is a standard value and expresses the temperature in Celsius, while the "F" property is defined via accessors, and is a "derived property" that simply relates back to the "C" property.

```
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.

Getters and Setters for classes

Classes also support a special getter-setter syntax. We could carry out the above example as follows:

```
function f2c(F) { return (F - 32) * 5 / 9; }
function c2f(C) { return C * 9 / 5 + 32; }
class Temp {
    constructor(Ctemp) { this.C = Ctemp; }
    get F() { return c2f(this.C); }
    set F(Ftemp) { this.C = f2c(Ftemp); }
}

let t = new Temp(0);
t.F;   // 32
t.F = 100; //  (changes C to 37.7777)
t.C;   // 37.7777 now
```

A common idiom is to create a "private" version of an a value, using an underscore to the beginning of the name, then using getters and setters for it:

```
class Person {
    constructor(first, last, age) {
        this._first = first;  // Could also hide via defineProperty
        this._last = last;
        this._age = age;
    }
    get first() { return this._first; }
    set first(firstName) { this._first = firstName; }
    get last() { return this._last; }
    set last(lastName) { this._last = lastName; }
    get full() { return `${this._first} ${this._last}`; }
    get age() { return this._age; }
    set age(newAge) {
        if (newAge < 0) { throw new Error("Incorrect age value"); }
        this._age = newAge;
    }
}

let haris = new Person("Haris", "Skiadas", 24); // I wish
haris.full;
haris.age = -1;  // Throws error
```