Notes - JavaScript - Getters & Setters

- Dr Nick Hayward

A brief introduction to various options for implementing *getters* and *setters* in JavaScript.

Contents

- Intro
- Control property access
- Define getters and setters
  - getters and setters in object literals
  - getters and setters with ES6 classes
  - getters and setters with `Object.defineProperty()`
  - getters and setters for validation
  - getters and setters with computed properties

Intro Getters and Setters are methods in JavaScript, which control access to specific object properties.

We may use such methods for various purposes in our applications, including custom logging, data validation, computed properties, and so on.

Explicit use of getters and setters has been supported in JavaScript since ES5.

Control property access Objects are, customarily, simple collections of properties in JavaScript.

We may use such object properties to track and modify an app's state. However, whilst we may simply update an object's property as follows

```
testObject.prop1 = 'hello world...";
```

we encounter some issues when we consider the following

- avoid accidental updates to an object property - e.g. assigning incorrect data and type
- logging changes to an object property
- ensure current property value is returned in an expected formate &c.

Each of these issues may be resolved in an elegant manner using *getters* and *setters*.

Define getters and setters Getters and setters may be defined in JavaScript using the following options,

- defined in object literals
- defined in ES6 `class` definitions
- use `Object.defineProperty()` method

getters and setters in object literals A common use of getters and setters is to control access to a given property of an object literal.

For example,

```
const planets = {
    names: ['mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune'],
    codes: {
        iau: ['Me', 'V', 'E', 'Ma', 'J', 'S', 'U', 'N'],
        unicode: ['U+263F', 'U+2640', 'U+2641', 'U+2642', 'U+2643', 'U+2644', 'U+2645',
              'U+2646']
    },
    get firstPlanet() {
        const planetDetails = {
            name: this.names[0],
            codes: {
                iau: this.codes['iau'][0],
                unicode: this.codes['unicode'][0]
            }
        };
        return planetDetails;
    },
    get latestPlanet() {
        const last = this.names.length-1;
        const planetDetails = {
            name: this.names[last],
            codes: {
                iau: this.codes['iau'][last],
                unicode: this.codes['unicode'][last]
            }
        };
        return planetDetails;
    },
    set planet(value) {
        this.names.push(value[0]);
        this.codes['iau'].push(value[1]);
        this.codes['unicode'].push(value[2]);
    }
};

console.log(planets.firstPlanet);
console.log(planets.latestPlanet);

planets.planet = ['alpha centauri', 'ac', 'N/A'];

console.log(planets.latestPlanet);
```

In this example, we have an object with various properties, `names` and `codes`. We define various getters and a setter to help us access specific values relative to these properties.

So, we may now call a single getter method to access the last data value regardless of updates. This is particularly useful for dynamic data sources.

Likewise, we may add a new planet to the data at a known position. In this example, we simply push the new data to the end of the current values. However, we might define a setter for the start of the values &c.

However, each of the above getter and setter methods are accessed as standard properties of the object literal. They are not accessed as standard methods.

getters and setters with ES6 classes We may also use the same pattern of getters and setters with ES6 (ES2015) classes.

For example, we might define the following class

```
class Planets {
    constructor() {
        this.planets = {
            names: ['mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus',
              'neptune'],
            codes: {
                iau: ['Me', 'V', 'E', 'Ma', 'J', 'S', 'U', 'N'],
                unicode: ['U+263F', 'U+2640', 'U+2641', 'U+2642', 'U+2643', 'U+2644',
                  'U+2645', 'U+2646']
            }
        };
    }

    get firstPlanet() {
        const planetDetails = {
            name: this.planets['names'][0],
            codes: {
                iau: this.planets['codes']['iau'][0],
                unicode: this.planets['codes']['unicode'][0]
            }
        };
        return planetDetails;
    }
}

const planets = new Planets();

console.log(planets.firstPlanet);
```

The basic usage of the getter method is the same as an object literal, we still call it as a property on an object.

However, the object is instantiated with a `planets` object as the default property. Then, we can access the `names` and `codes` data using the getter method as a property on the instantiated `planets` object.

Likewise, we may add a setter method to this class for updating this object.

```
// setter method - new planet data pushed to end of data
set planet(value) {
    this.planets['names'].push(value[0]);
    this.planets['codes']['iau'].push(value[1]);
    this.planets['codes']['unicode'].push(value[2]);
}
```

The pattern for the method is the same as the previous object literal. We may also access this latest data update using the same getter method we added to the object literal example,

```
// getter method - latest planet data
get latestPlanet() {
    const last = this.planets['names'].length-1;
    const planetDetails = {
        name: this.planets['names'][last],
        codes: {
            iau: this.planets['codes']['iau'][last],
            unicode: this.planets['codes']['unicode'][last]
        }
    };
    return planetDetails;
}
```

getters and setters with `Object.defineProperty()` A noticeable issue with getters and setter for Object literals and classes is the lack of wrapped private variables.

In other programming languages, a common usage for getters and setters is curated acces to private variables and properties.

Whilst JavaScript does not natively support private object properties, we may commonly approximate such usage with closures. In effect, we may define variables, and then object methods as closures around them.

However, we may use `Object.defineProperty()` to help partly resolve this issue, defining a new propery descriptor for an object. This descriptor may handle `get` and `set` properties for the custom methods.

For example, we may implement getters and setters to control access to a *private* object property.

```
// constructor for Planets object
function Planets() {

    // mimick private variable usage...
    const _names = ['mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus',
                    'neptune'];

    Object.defineProperty(this, 'names',  {
        get: () => {
            return _names;
        },
        set: value => {
            _names.push(value);
        }
    });

}
```

Using the `Planets` constructor, we can instantiate an object,

```
// test usage for Planets and *private* variable access
const planets = new Planets();
```

and try calling the private variable `_names`,

```
// returns undefined for _names variable
console.log(planets._names);
```

However, direct access to the variable returns `undefined`. So, we need to use the defined getter for the `names` property

```
// use getter to retrieve array of names
console.log(planets.names);
```

Likewise, we may use the setter method to update the `_names` array with a new value,

```
// use setter to update array values - push new planet name
planets.names = 'alpha centauri'
```

getters and setters for validation We may also use getters and setters as an option for validation, notably as an option to validate property values.

For example, the previous setter can now be updated as follows

```
set: value => {
    // check value passed is a string
    if (typeof value !== 'string') {
        throw new TypeError('planet name needs to be a String value...');
    }
    _names.push(value);
}
```

Then, if we try to pass a number, the setter will not validate the `value` and simply return a type error,

```
// use setter to update array values
// - push new planet name as number to test type validation
planets.names = 1252389;
```

getters and setters with computed properties Getters and setters may also be used to return computed properties. This usage may be combined with private variables, providing an option to return curated information without direct access to the underlying data.

For example, we might define private variables for planet names and planet codes. Then, we compute the relationship between the name and the known codes, returning a complete object for a single planet.

The benefit, of course, is that we do not need to provide access to all of the data to simply return information on a single planet.

e.g.

```
get: () => {
    const planetDetails = {
        names: _names,
        codes: _codes
    };
    return planetDetails;
},
```

So, this example will combine the data for the names and codes, and return a single object for planet details.

However, to work with a single, passed planet name we may define a `get` method on the instantiated object, e.g.

```
// add method to instantiated object - return computed details for specified planet
this.getPlanet = (name) => {
    if (_names.includes(name)) {
        const index = _names.indexOf(name);
        const planetDetails = {
            name: name,
            codes: {
                iau: _codes['iau'][index],
                unicode: _codes['unicode'][index],
            }
        }
        return planetDetails;
    }
}
```

This binds the method to the instantiated object and returns a computed value for the passed planet name. The private variable is still restricted, but we may use it within the scope of the instantiated object with the defined method.