WEB230: JavaScript 1

Module 7: The Secret Life of Objects

Encapsulation

- Object Oriented programming was developed in the 1970s and '80s
- the idea is to divide programs into smaller pieces
- each piece is responsible for managing its own state

Methods

- properties that hold functions

```
let rabbit = {};
rabbit.speak = function(line) {
   console.log(`The rabbit says "${line}"`);
};

rabbit.speak('Hello');
```

this

- usually a method needs to do something with the object
- inside a function that is called as a method of an object, `this` refers to the object

```
function speak(line) {
   console.log(`The ${this.type} rabbit says "${line}"`);
}
let whiteRabbit = {
   type: 'white',
   speak: speak
};
whiteRabbit.speak('Oh, how late it is!');
```

Arrow Functions

- arrow functions don't have their own `this`
- instead they use `this` from the surrounding scope

```
const speak = line => {
   console.log(`The ${this.type} rabbit says "${line}"`);
}
let whiteRabbit = {type: 'white', speak: speak};

whiteRabbit.speak('Oh, how late it is!');
```

- **dosen't work** - this.type == undefined

Classes

- a class defines the shape of a type of object
    - what methods and properties it has
- objects based on a class are called an "instance" of the class

Prototypes

- Even an empty object has properties:

```
let empty = {};
console.log(empty.toString);
console.log(empty.toString());
```

- It has a method called `toString`
- Objects have **prototypes**
- `Object.getPrototypeOf()` will return the prototype of an object
- `Object.prototype` is the base prototype of most objects
- Other prototypes can be layered on top

```
console.log(Object.getPrototypeOf([]) === Array.prototype);
console.log(Object.getPrototypeOf(Array.prototype) === Object.prototype);
```

Prototype

- defines properties that are shared
- properties that are different for each must be stored directly on the object

```
let empty = {};
console.log(empty.toString);
// → function toString(){…}
console.log(empty.toString());
// → [object Object]
```

- JavaScript prototypes can be considered informal classes

Constructors

- constructor functions create objects that derive from some shared prototype
- calling a function with the `new` keyword in front of it causes it to be treated as a constructor
- the constructor will have its `this` variable bound to a new object
- the new object will be returned

```
function Rabbit(type) {
   this.type = type;
}

let killerRabbit = new Rabbit('killer');
let blackRabbit = new Rabbit('black');
console.log(blackRabbit.type);
```

- the constructor has a property named `prototype`
    - holds a empty object that derives from `Object.prototype`
    - every instance created with this constructor will have this object as its prototype

```
Rabbit.prototype.speak = function(line) {
   console.log(`The ${this.type} rabbit says "${line}"`);
};
blackRabbit.speak('Doom...');
```

Class Notation

- a JavaScript `class` is a constructor function
- newer, less awkward notation
- not supported in Internet Explorer

- the `class` keyword starts the declaration
- `constructor()` (optional) is the constructor function
- methods can be declared *after* the constructor
    - don't use the `function` keyword
    - these methods are put in the prototype
    - can't declare properties inside a `class`
- the `class` block is run in strict mode

```
class Rabbit {
   constructor(type) {
      this.type = type;
   }
   speak(line) {
      console.log(`The ${this.type} rabbit says "${line}"`);
   }
}

let killerRabbit = new Rabbit('killer');
let blackRabbit = new Rabbit('black');
```

Overriding Derived Properties

- If the object does not have a property it will look to the prototype
- If we add a property (or method) earlier in the prototype chain, it will be used

- also know as "Prototype Interference"

```
Rabbit.prototype.teeth = 'small';

console.log(killerRabbit.teeth);

killerRabbit.teeth = 'long, sharp, and bloody';

console.log(killerRabbit.teeth);
console.log(blackRabbit.teeth);
```

`in` Operator

- tells us if an object has access to a property
- `in` evaluates to `true` if the property is present

```
console.log('teeth' in blackRabbit);
// true
console.log('teeth' in killerRabbit);
// true
```

`.hasOwnProperty()` Method

- check if a property belongs to the object but **not** on it's prototype
- `.hasOwnProperty()` returns `true` if the property is on the object

```
console.log(blackRabbit.hasOwnProperty('teeth'));
// true
console.log(killerRabbit.hasOwnProperty('teeth'));
// false
```

`for…in` Loop

- `for…in` will loop through properties of an object

```
for( let prop in blackRabbit ) {
   console.log(prop, blackRabbit[prop]);
}
```

Polymorphism

- polymorphism is when a method or an operator does different things on different data types
- JavaScript methods can be polymorphic
- Eg. all values have a method `.toString()`
    - `.toString()` is used to convert values to strings
- We can write our own `.toString()`, to work with our object

```
class Rabbit {
    ...
    toString() {
        return this.type + ' rabbit';
    }
}
```

Getters and Setters

- Often need to control setting or getting values of a property
- This created the style of writing getter and setter methods

```
class Temperature {
    constructor(celsius) {
        this.celsius = celsius;
    }
    getFahrenheit() {
        return this.celsius * 1.8 + 32;
    }
    setFahrenheit(value) {
        this.celsius = (value - 32) / 1.8;
    }
}

let temp = new Temperature(22);
console.log(temp.getFahrenheit());
```

JavaScript has Getters and Setters

- JavaScript has built-in getters and setters
- Act like properties but call methods

```
class Temperature {
    constructor(celsius) {
        this.celsius = celsius;
    }
    get fahrenheit() {
        return this.celsius * 1.8 + 32;
    }
    set fahrenheit(value) {
        this.celsius = (value - 32) / 1.8;
    }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
```