

Object Orientated JavaScript

PROGRAMMING WITH JAVASCRIPT



Introduction

- Objects revisited
 - Object notation
 - Scope
- Creating your own objects
 - Adding functions to objects
 - Constructors
 - Prototypes
 - Chaining objects
- The with statement
 - Sealing objects

JavaScript objects (1)

- Everything in JavaScript is an object
 - Functions, dates, DOM elements
 - How we extend the language with our own types

- Creating...

- Use new keyword or { }

```
let myBike = new Object();  
myBike.make = "Honda";  
myBike.model = "Fireblade";  
  
let myBike2 = {  
  make: "Honda",  
  model: "Fireblade"  
};
```

- Properties

- Use dot notation or object literal notation

```
myBike.make = "Yamaha";  
myBike["model"] = "R1";
```

3

Everything in JavaScript is an Object. This includes functions, dates, strings and DOM Elements. This is how we are able to extend the language with our own types.

To create an object, we use the **new** keyword or we can simply use empty curly braces if we are creating an instance of the object type.

Objects' members are stored as an associative array. We can access members of this array using dot notation or via indexer syntax as shown above.

This gives us great flexibility and makes creating custom objects a simple process; however, there are more useful and better-performing ways to create reusable objects as we'll discover.

JavaScript objects (2)

- Use for...in loop to iterate over the properties

```
var myBike = {  
  make: "Honda",  
  model: "Fireblade",  
  year: 2008,  
  mileage: 12500,  
}  
  
for (let propName in myBike) {  
  print `${propName} :: ${myBike[propName]}`;  
}
```

4

To iterate over the members of the associative array, we can use a `for...in` loop as shown in the first example above. This is the basis for a reflection-like mechanism in JavaScript.

As well as simple properties, as Functions are first-class objects, we can add methods by having a property of our object with the type of function as in the second and third examples above.

The third example is the preferred method as it enables us to declare both properties and methods in the same block, making our code more readable.

Functions as Constructors

- Constructors
 - A function used in conjunction with the **new** operator
 - In the body of the constructor, **this** refers to the newly-created instance
 - If the new instance defines methods, within those methods **this** refers to the instance

```
function Dog() {  
  this._name = '';  
  this._age = 0;  
}  
var dog = new Dog();
```

- Private properties are prefixed with an underscore
 - Purely convention as there is no notion of private scope for properties in JavaScript

3

The idea of a constructor (code that is run when an object is first instantiated/initialised) is nothing new in the object oriented world. In JavaScript, this is simply a function that is used in conjunction with the **new** operator.

This is the preferred approach to creating custom objects. Within the body of the constructor, **this** refers to the newly-created instance; so, by using the **this** keyword, we can set instance properties.

(e.g. **this.propertyName= "some Value";**)

If the new instance defines any methods – other functions – then those methods will have the same context (i.e. the new instance).

The prototype object

- Holds all properties that will be inherited by the instance
 - Defines the structure of an object
 - All new instances inherit the members of the prototype
 - References to objects/arrays added to prototype shared
 - Slightly slower as instance is searched before prototype
 - General practice
 - Declare members in the constructor
 - Declare methods in the prototype

```
function Dog() {  
  this._name = '';  
  this._age = 0;  
}  
  
Dog.prototype.speak = function () {  
  alert("Woof!");  
}  
  
let dog = new Dog();  
dog.speak(); // alerts "Woof!"
```

The built-in prototype object holds all properties (including methods) that will be inherited by the instance. In essence, this defines the structure of the object and is comparable to a class. All new instances of the object will inherit the members of the prototype. References to data structures – such as objects/arrays – that are added to the prototype are shared between instances.

Accessing these shared properties is slightly slower as the instance is searched before the prototype, but it does mean that we save on memory allocation.

The general advice is to declare members in the constructor (private fields) and to declare methods (functions) within the prototype. This way gives the separation between data and behaviour that we expect.

Inheritance in JavaScript

Inheritance in JS is achieved through prototypes. Every object has a prototype from which it can inherit members.

```
function Vehicle() {
  this._make = '';
  this._model = '';
}
Vehicle.prototype = {
  accelerate: function () {
    throw Error("This method should be overridden");
  }
}

function Bike() {
  Vehicle.call(this);
}

Bike.prototype = new Vehicle();
Bike.prototype.accelerate = function () {
  //Concrete implementation here
}

let bike = new Bike();
bike.accelerate();
```

Call base class constructor

Inherit properties defined in prototype

Override accelerate method

7

The above example shows a way of simulating inheritance using pure JavaScript.

First, we declare the constructor of the base class Vehicle, then the prototype is defined with an abstract method (accelerate) where we simply throw an Error as this should be overridden by derived classes. After that, we create a constructor for the Bike class that will inherit from Vehicle.

We use the call method, which allows us to change the context of the call so that **this** will refer to a different object (we pass in our new Bike – **this**). So, when the Vehicle constructor is called, the `_make` and `_model` will be added to the new Bike – we have inherited the field members.

Then we inherit the properties of Vehicle by setting the prototype of Bike to a new Vehicle instance.

Finally, we override the accelerate method by redefining it on the Bike prototype.

We can then use the Bike class and it will have all the capabilities of a Bike that inherits from Vehicle.

However, there are some drawbacks to this method – we have simulated the inheritance but have no idea (from a user perspective) of the base class or the inheritance chain. Nor can we call base implementations without detailed knowledge of the class structure.

Inheritance in action – Custom Error handler

- We previously discussed the inbuilt Error object
 - Through inheritance, we can create our own error types

```
function DivisionByZeroError(message) {  
    this.name = "DivisionByZeroError";  
    this.message = (message || "");  
}  
  
DivisionByZeroError.prototype = new Error();
```

8

We learned earlier in the course how to generate customised error messages using the built-in exception types. However, another approach is to create new exception types by extending the existing "Error" type. Because the new type inherits from "Error", it can be used like the other built-in exception types.

The following example returns to the problem of dealing with division by zero. Instead of using an "Error" or "RangeError" object as we did earlier, we are going to create our own type of exception.

In this example, we are creating the "DivisionByZeroError" exception type. The function in the example acts as the constructor for our new type. The constructor takes care of assigning the "name" and "message" properties.

Classes

- Syntactic sugar over prototypal inheritance
- Gotcha: NOT hoisted like functions
- Executed in strict mode

```
class Car {  
  constructor (wheels, power) {  
    this._wheels = wheels;  
    this._power = power;  
    this._speed = 0;  
  }  
  
  accelerate(time) {  
    this._speed = this._speed + 0.5*this._power*time;  
  }  
}  
const MY_CAR = new Car(4, 20); //constructor called
```

9

Constructor method is called when the class is instantiated through the “new” keyword.

Methods can be created without using the function keyword or assigning to “this”.

Classes: Extends

- The extends and super keywords allow sub-classing

```
class Vehicle {  
  constructor (wheels, power) {  
    this._wheels = wheels;  
    this._power = power;  
    this._speed = 0;  
  }  
  
  accelerate(time) {  
    this._speed = this._speed + 0.5*this._power*time;  
  }  
}  
class Car extends Vehicle {  
  constructor (wheels, power) {  
    super(wheels, power); //call the parent constructor  
    this._gps = true; //GPS as standard  
  }  
}  
const MY_CAR = new Car(4, 20);
```

10

GOTCHA! : “this” will be undefined before you call “super()” in a subclass

Classes: Static

- The static keyword allows for method calls to a class that hasn't been instantiated

```
class Circle {  
  constructor (radius, centre) {  
    this.radius = radius;  
    this.centre = centre;  
  }  
  
  static area(circle) {  
    return Math.PI * Math.pow(circle.radius,2);  
  }  
}  
  
const MY_CIRCLE = new Circle(5,[0,0]);  
console.log(Circle.area(myCircle)); //78.53981633974483
```

Sealing objects to prevent expando errors

- Extensibility of objects can be toggled
 - Turning off extensibility prevents new properties changing the object
 - `Object.preventExtensions(obj)`
 - `Object.isExtensible(obj)`

```
let obj = {
  name: "Dave";
};
print(obj.name); //Dave

console.log(Object.isExtensible(obj)); // true

Object.preventExtensions(obj);

obj.url = "http://ejohn.org/"; //Exception in strict mode (silent fail otherwise)

console.log(Object.isExtensible(obj)); //false
```

12

A new feature of ECMAScript 5 is that the extensibility of objects can now be toggled. Turning off extensibility can prevent new properties from getting added to an object.

ES5 provides two methods for manipulating and verifying the extensibility of objects.

`Object.preventExtensions(obj)`

`Object.isExtensible(obj)`

`preventExtensions` locks down an object and prevents any future property additions from occurring.

Review

- Objects revisited
 - Object notation
 - Scope
- Creating your own objects
 - Adding functions to objects
 - Constructors
 - Prototypes
 - Chaining objects
- Sealing objects
 - Defend against unexpected object mutation

Exercise

- Recreating the slideshow as an OO object
- Working with simple object inheritance