Notes - JavaScript - Objects - Prototype

- Dr Nick Hayward

A brief introduction to object orientation using prototypes with plain JavaScript.

Contents

Intro

Along with the following traits of JS (ES6 ...),

- functions as first-class *objects*
- versatile and useful structure of functions with closures
- combine generator functions with promises to help manage async code

we may also consider an equally important aspect of modern JS, object *prototypes*.

A *prototype* object may be used to delegate the search for a particular property. In effect, a *prototype* is a useful and convenient option for defining properties and functionality accessible to other objects.

A *prototype* is a useful option for replicating many concepts in traditional object oriented programming.

understanding prototypes

In JS, we may create objects, for example, using *object-literal* notation, e.g.

```
let testObject = {
        property1: 1,
        prooerty2: function() {},
```

```
        property3: {}
}
```

So, in this object we have a simple value for the first property, a function assigned to the second property, and another object assigned to the third object.

As a dynamic language, JS will also allow us to modify these properties, delete any not required, or simply add a new one as necessary.

However, this dynamic nature may also completely change the properties in a given object. In traditional object-oriented programming languages, this issue is often solved using inheritance.

In JS, we can use ***prototypes*** to implement inheritance.

basic idea of prototypes

The basic idea of ***prototypes*** in JS is straightforward.

Every ***object*** can have a reference to its ***prototype*** - a delegate object with properties (default for child objects)

- search object for a property - then search ***prototype*** for this property

i.e. the prototype is a fall back object to search for a given property &c.

e.g.

```
const object1 = { title: 'the glass bead game' };
const object2 = { author: 'herman hesse' };

console.log(object1.title);

Object.setPrototypeOf(object1, object2);

console.log(object1.author);
```

In this example, we define two objects. Their properties can be called with standard object notation, and they can be modified and mutated as standard.

However, as an object's internal prototype is not directly accessible, we need to use the method `setPrototypeOf()` to set and update an object's prototype.

In this example, we pass `object1` as the object to update, and `object2` as the object to set as prototype. If a requested property is not found on `object1`, JS will then search its prototype.

So, we request `author` as a property of `object1`. It's not found on `object1`, but it's available as a property of its prototype (it's parent...).

prototype inheritance

Prototypes, and their properties, can also be inherited thereby creating a chain of inheritance.

e.g.

```
const object1 = { title: 'the glass bead game' };
const object2 = { author: 'herman hesse' };
const object3 = { genre: 'fiction' };

console.log(object1.title);

Object.setPrototypeOf(object1, object2);
Object.setPrototypeOf(object2, object3);

console.log(object1.author);
console.log(`genre from prototype chain = ${object1.genre}`); // use
template literal to output...
```

In this example, `object1` has access to the prototype of its parent, `object2`. A property search against `object1` will now include its own prototype, `object2`, and its prototype as well, `object3`.

The output for `object1.genre` will return the value stored in the property on `object3`.

object constructor & prototypes

Object-oriented languages, such as Java and C++, include a class constructor, which provides known encapsulation and structuring. Effectively, the constructor is initialising an object to a known initial state.

In essence, this pattern allows us to consolidate a set of properties and methods for a class of objects in one place.

JS offers such a mechanism, although in a slightly different form to Java, C++ &c. JS still uses the `new` operator to instantiate new objects via constructors. However, JS does not include a true class definition.

Instead, the `new` operator in JS is applied to a constructor function, which triggers the creation of a new object.

prototype method

In JS, every function includes their own prototype object, which is set automatically as the prototype of any created objects.

e.g.

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();
```

```
console.log(bookRecord.library);
```

Likewise, we may set a default method on an instantiated object's prototype.

instance properties

As JS searches an object for properties, values or methods, instance properties will be searched before trying the prototype. A known order of precedence will work.

e.g.

```
//constructor for object
function LibraryRecord() {
  // set property on instance of object
  this.library = 'waldzell';

  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();

console.log(bookRecord.library);
```

In this example, the `this` keyword refers directly to the newly created object. The properties set in the constructor are created directly on any object instantiated with the `LibraryRecord()` instance.

Therefore, if we later search for the property `library`, there is no need to search against the object's prototype as it exists on the instantiated object.

However, there's an interesting side-effect to this pattern. If we instantiate multiple objects with this constructor, each object gets its own copy of the constructor's properties, and access to the same prototype.

So, we may end up with multiple copies of the same properties in memory. Therefore, if replication is likely for objects, it's often more efficient to store properties and methods against the prototype.

n.b. if we need to mimic private object variables by defining methods within constructor functions, then we have to specify methods within the constructor itself.

side effects of JS dynamic nature

JS is a dynamic language, which means properties can easily be added, removed, and modified as necessary.

This dynamic nature is also true for both function prototypes and object prototypes.

e.g.

```
```

```
//constructor for object
function LibraryRecord() {
  // set property on instance of object
  this.library = 'waldzell';
}

// create instance of LibraryRecord - call constructor with `new` operator
const bookRecord1 = new LibraryRecord();

// check output of value for library property from constructor
console.log(`this library = ${bookRecord1.library}`);

// add method to prototype after object created
LibraryRecord.prototype.updateLibrary = function() {
  return this.retreat = 'mariafels';
};

// check prototype updated with new method
console.log(`this retreat = ${bookRecord1.updateLibrary()}`);

// then overwrite prototype - constructor for existing object
unaffected...
LibraryRecord.prototype = {
  archive: 'mariafels',
  order: 'benedictine'
};

// create instance object of LibraryRecord...with updated prototype
const bookRecord2 = new LibraryRecord();

// check output for second instance object
console.log(`updated archive = ${bookRecord2.archive} and order =
${bookRecord2.order}`);
// check output for second instance object - library
console.log(`second instance object - library = ${bookRecord2.library}`);
// check if prototype updated for first instance object - NO
console.log(`first instance object = ${bookRecord1.order}`);
// manual update to prototype for first instance object still available
console.log(`this retreat2 = ${bookRecord1.updateLibrary()}`);

// check prototype has been fully overwritten - e.g. `updateLibrary()` no
longer available on prototype for new instance object
try {
// updates to original prototype are overridden - error is returned for
second instantiated object...
console.log(`this retreat = ${bookRecord2.updateLibrary()}`);
 } catch(error) {
  console.log(`modified prototype not available for new object...\n
${error}`);
 }
```

In this slightly contrived example we can see the dynamic nature of prototypes.

object typing via constructors

We can check the function used as a constructor to instantiate an object using the `constructor` property.

e.g.

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

// create instance object for LibraryRecord
const bookRecord = new LibraryRecord();

// output constructor for instance object
console.log(`constructor = ${bookRecord.constructor}`);

// check if function was constructor (use ternary conditional)
const check = bookRecord.constructor === LibraryRecord ? true : false;
// output result of check
console.log(check);
```

instantiate a new object using a constructor reference

We can use a constructor to create a new instance object.

However, we can also use the `constructor()` of the new object to then create another object.

The second object, created with the constructor method, is still an object of the original constructor function.

e.g.

```
//constructor for object
function LibraryRecord() {
  //set default value on prototype
  LibraryRecord.prototype.library = 'castalia';
}

const bookRecord = new LibraryRecord();
const bookRecord2 = new bookRecord.constructor();
```

achieving inheritance

Inheritance enables re-use of an object's properties by another object.

This helps us efficiently avoid repetition of code and logic, thereby improving reuse and data across an application.

Inheritance works in a slightly different manner in JavaScript when compared with other object-oriented programming languages.

So, in JS we're trying to effect a prototype chain to ensure that inheritance works beyond simply copying prototype properties.

For example, we might want a book to belong to a corpus, a corpus to an archive, an archive to a library, and so on...

inheritance with prototypes

For inheritance, we're trying to create a prototype chain by using an instance of an object as the prototype for another object.

i.e. `SubClass.prototype = new SuperClass()`

So, this pattern works as a prototype chain for inheritance. The prototype of the `SubClass` instance will work as an instance of the `SuperClass`. This prototype will have all the properties of `SuperClass`, which will also have properties from its superclass, and so on. Thereby creating a prototype chain of expected inheritance.

e.g.

```
//constructor for object
function Library() {
        // instance properties
  this.type = 'library';
  this.location = 'waldzell';
}

// constructor for Archive object
function Archive(){
        // instance property
  this.domain = 'gaming';
}

// update prototype to parent Libary - instance relative to parent & child
Archive.prototype = new Library();

// instantiate new Archive object
const archiveRecord = new Archive();

// check instance object - against constructor
if (archiveRecord instanceof Archive) {
  console.log(`archive domain = ${archiveRecord.domain}`);
}

// check instance of archiveRecord - instance of Library & Archive
if (archiveRecord instanceof Library) {
        // type property from Library
  console.log(`Library type = ${archiveRecord.type}`);
        // domain property from Archive
```

```
        console.log(`Archive domain = ${archiveRecord.domain}`);
}
```

Inheritance has been achieved by setting the prototype of the Archive constructor to a new instance of the Library object.

issues with overriding the constructor property

By setting the Library object as the defined prototype for the Archive constructor,

```
Archive.prototype = new Library();
```

we've now lost a connection to the Archive constructor itself. If we check the constructor,

```
// check constructor used for archiveRecord object
if (archiveRecord.constructor === Archive) {
  console.log('constructor found on Archive...');
} else {
        // Library constructor output - due to prototype
  console.log(`Archive constructor = ${archiveRecord.constructor}`);
}
```

the Library constructor will be returned in the `else` statement. This may become an issue, as the constructor property can be used to determine the function used to create the object.

some benefits of overriding the constructor property

```
//constructor for object
function Library() {
        // instance properties
  this.type = 'library';
  this.location = 'waldzell';
}

// extend prototype
Library.prototype.addArchive = function(archive) {
  console.log(`archive added to library - ${archive}`);
        // add archive property to instantiate object
        this.archive = archive;
        // add property to Library prototype
        Library.prototype.administrator = 'knechts';
}

// constructor for Archive object
function Archive(){
        // instance property
```

```
  this.domain = 'gaming';
}

// update prototype to parent Libary - instance relative to parent & child
Archive.prototype = new Library();

// instantiate new Archive object
const archiveRecord = new Archive();
// call addArchive on Library prototype
archiveRecord.addArchive('mariafels');

// check instance object - against constructor
if (archiveRecord instanceof Archive) {
  console.log(`archive domain = ${archiveRecord.domain}`);
}

// check constructor used for archiveRecord object
if (archiveRecord.constructor === Archive) {
  console.log('constructor found on Archive...');
} else {
  console.log(`Archive constructor = ${archiveRecord.constructor}`);
        console.log(`Archive domain = ${archiveRecord.domain}`);
        console.log(`Archive = ${archiveRecord.archive}`);
        console.log(`Archive admin = ${archiveRecord.administrator}`);
}

// check instance of archiveRecord - instance of Library & Archive
if (archiveRecord instanceof Library) {
        // type property from Library
  console.log(`Library type = ${archiveRecord.type}`);
        // domain property from Archive
        console.log(`Archive domain = ${archiveRecord.domain}`);
}

// instantiate another Archive object
const archiveRecord2 = new Archive();
// output instance object for second archive
console.log('Archive2 object = ', archiveRecord2);
// check if archiveRecord2 object has access to updated archive
property...NO
console.log(`Archive2 = ${archiveRecord2.archive}`);
// check if archiveRecord2 object has access to updated adminstrator
property...YES
console.log(`Archive2 administrator = ${archiveRecord2.administrator}`);
```

configure object properties

Each object property in JS is described with a **property descriptor**. We can use such descriptors to configure specific keys, e.g.

- *configurable* - boolean setting
    - true = property's descriptor may be changed and the property deleted

    - false = no changes &c.
- *enumerable* - boolean setting
    - true = specified property will be visible in a `for-in` loop through object's properties
- *value* - specifies value for property (default is undefined)
- *writable* - boolean setting
    - true = the property value may be changed using an assignment
- *get* - defines the getter function, called when we access the property
    - **n.b.** can't be defined with *value* and *writable*
- *set* - defines the setter function, used whenever an assignment is made to the property
    - **n.b.** can't be defined with *value* and *writable*

For example, if we create the following property for an object,

```
archive.type = 'private';
```

it will be *configurable*, *enumerable*, *writable*, and with a value of *private*. Its *get* and *set* will currently be undefined.

To update or modify a property configuration, we can use the built-in `Object.defineProperty()` method. This method takes an object, which may be used to,

- define or update the property
- define or update the name of the property
- define a property descriptor object

e.g.

```
// empty object
const archive = {};

// add properties to object
archive.name = "waldzell";
archive.type = "game";

// define property access, usage, &c.
Object.defineProperty(archive, "access", {
        configurable: false,
        enumerable: false,
        value: true,
        writable: true
});

// check access to new property
console.log(`${archive.access}, access property available on the
object...`);

/*
* check we can't access new property in loop
* - for..in iterates over enumerable properties
```

```
*/
for (let property in archive) {
        // log enumerable
        console.log(`key = ${property}, value = ${archive[property]}`);
}

/*
* plain object values not iterable...
* - returns expected TyoeError - archive is not iterable
*/
for (let value of archive) {
        // value not logged...
        console.log(value);
}
```

Using ES6 Classes

ES6 provides a new `class` keyword to enable object creation, and aid in inheritance. In effect, it's *syntactic sugar* for prototypes and constructors.

e.g.

```
// class with constructor & methods
class Archive {
  constructor(name, admin) {
    this.name = name;
          this.admin = admin;
  }
        // class method
  static access() {
    return false;
  }
        // instance method
        administrator() {
                return this.admin;
        }
}

// instantiate archive object
const archive = new Archive('Waldzell', 'Knechts');

// check parameter usage with class
const nameCheck = archive.name === `Waldzell` ? archive.name : false;

// log archive name
console.log(`class archive name = ${nameCheck}`);
// call class method
console.log(Archive.access());
// call instance method
console.log(`archive administrator = ${archive.administrator()}`);
console.log(`archive name = ${nameCheck}`);
```

ES6 classes as syntactic sugar

Classes in ES6 are simply syntactic sugar for prototypes. In effect, we use the new keyword to mimick the concept of object-oriented classes.

A prototype implementation of the above `class`, and usage.

```
// constructor function
function Archive(name, admin) {
  this.name = name;
        this.admin = admin;

        // instance method
        this.administrator = function () {
                return this.admin;
        }

        // add property to constructor
        Archive.access = function() {
        return false;
        };
}

// instantiate object - pass arguments
const archive = new Archive('Waldzell', 'Knechts');

// check parameter usage with ternary conditional...
const nameCheck = archive.name === `Waldzell` ? archive.name : false;

// output name check...
console.log(`prototype archive name = ${nameCheck}`);
// call constructor only method
console.log(Archive.access());
// call instance method
console.log(`archive administrator = ${archive.administrator()}`);
```