Javascript Inheritance: Prototypes and Classes

Until recently Javascript did not have a class system. This doesn't mean that we couldn't create reusable objects but that we had to work through hoops to make it work.

Prototypal Inheritance: It's all about the prototype

Early versions of Javascript until ES2015 used prototypal inheritance. We create a master object, inherit from the base object, and assign new properties to the object's prototype.

In this example, `Employee` is the base objects with a few attributes. Other types of objects will inherit from this directly or indirectly.

```
function Employee() {
  this.name = '';
  this.dept = 'general';
}
```

The next block of functions show how objects can inherit directly from the base class object.

We use `Employee.call` with the object we want to use as parent and this to represent the object we're calling it from as a super contstructor call; this will give us access to the Employee object methods and attributes

```
function Manager() {
  Employee.call(this);
  this.reports = [];
}
Manager.prototype = Object.create(Employee.prototype);
```

```
Manager.prototype.constructor = Manager;

function WorkerBee() {
  Employee.call(this);
  this.projects = [];
}
WorkerBee.prototype = Object.create(Employee.prototype);
WorkerBee.prototype.constructor = WorkerBee;
```

To further illustrate how prototypal inheritance works, we've defined two additional types of employees and, rather than define them based on the base Employee object we define it based on one of the children, `WorkerBee`.

`Engineer` and `SalesPerson` inherit from WorkerBee, they also get all the properties in `Employee` without having to do any further setup.

```
function SalesPerson() {
   WorkerBee.call(this);
   this.dept = 'sales';
   this.quota = 100;
}
SalesPerson.prototype = Object.create(WorkerBee.prototype);
SalesPerson.prototype.constructor = SalesPerson;
console.log(SalesPerson.prototype);

function Engineer() {
   WorkerBee.call(this);
   this.dept = 'engineering';
   this.machine = '';
}
Engineer.prototype = Object.create(WorkerBee.prototype)
Engineer.prototype.constructor = Engineer;
console.log(Engineer.prototype);
```

Next we instantiate different objects:

```
let jim = new Employee();
```

```
let sally = new Manager();
let mark = new WorkerBee();
let fred = new SalesPerson();
let jane = new Engineer();
```

An earlier post, Prototypal Inheritance and Classes showed how to create objjects and add to the prototype, but did not talk about how to inherit from the prototype chain.

Classes to the rescue

If you've done any programming at all you're likely to have found classes. Surprisingly until ES2015 there were no classes in JavaScript forcing us to use the ealier prototypal chain modell. Under the hood, Javascript classes use protypal inheritance so only the sugar on top has changed.

I've reworked some of the prototypal inheritance demos to classes and added some additional things that will make life easier.

The base `Employee` class has the basic information for all the people that we will work with.

Because we will change the department for an individual we create a getter/setter pair to set and retrieve the value of the property, `dept` in this case.

We also set two class methods: `greeting()` and `farewell`.

```
class Employee {
  // default constructor
  constructor(first, last, dept) {
    this.name = {
      first,
      last
    };
    this._dept = dept;
  }

  // Getters and setters
```

```
  get dept() {
    return this._dept;
  }

  set dept(newDept) {
    this._dept = newDept;
  }

  // class methods

  greeting() {
    console.log(`Hi! I'm ${this.name.first}`);
  }

  farewell() {
    console.log(`${this.name.first} has left the building. Bye for now!`)
  }
}
`Hi! I'm ${this.name.first}`
```

We instantiate

```
let han = new Employee('Han', 'Solo');
han._dept = 'smuggl'Solo'console.log(han.dept);
han.greeting();
// Hi! I'm Han

let leia = new Employee('Leia', 'Organa');
leia._dept = 'politics';
console.log(leia.dept);
leia.farewell();
'Leia'// Leia has left the building. Bye for now
```

We can create new classes that extend from our base `Employee` class by using the `extends` keyword and the name of the class we want to base it on, and using the `super()` method in the constructor with the name of the fields from the parent class we want to use.

```
class SalesPerson extends Employee {
  constructor(first, last, dept, quota) {
    // calls the parent class constructor
    // with only the items we want to
    super(first, last);
    // override the dept item that we didn't
    // take from the constructor
    this.dept = 'sales';
    'sales'// set the quota
    this.quota = 100;
  }

  deptWork() {
    console.log(`${this.name.last} works in the ${this.dept}`);
  }
}
`${this.name.last} works in the ${this.dept}`
```

The `SalesPerson` class uses values from `Employee` and adds functionality that goes beyond the parent class. We didn't write `greeting()` and `farewell()` methods for `SalesPerson` but they are available as part of `Employee`, so we can still use them, we get them "for free" in the derived class.

```
let snape = new SalesPerson('Severus', 'Snape');
snape.greeting();
snape.deptWork();
snape.farewell();
```

We'll use the `WorkerBee` class to extend `Employee` and `Engineer` to extend `WorkerBee`. You can say that a WorkerBee is an Employee and an Engineer is a WorkerBee.

The `WorkerBee` class introduces one additional element to `Employee`. `projects` will list the projects this particular WorkerBee is working on.

```
class WorkerBee extends Employee {
  constructor(first, last, dept, projects) {
```

```
    super(first, last, dept);
    this.projects = [];
  }
}
```

When we instantiate a new `WorkerBee`, we get all the methods, setters and getters from Employee without having to duplicate any of the functionality

```
let newbee = new WorkerBee('buble', 'bee', 'working', 'cleanup')

newbee.greeting();
newbee._dept = 'accountring';
console.log(newbee.dept);
newbee.farewell();
```

Engineer is a child of WorkerBee. It adds its own attribute, getter, setter and class method.

```
class Engineer extends WorkerBee {
  constructor(first, last, dept, projects, language) {
    super(first, last, dept, projects);
    this._language = language;
  }

  get language() {
    return this._language;
  }

  set language(newLanguage) {
    this._language = newLanguage;
  }

  programIn() {
    console.log(`${this.name.last} programs in ${this._language}`)
  }
}
```

When we instantiate a new Engineer we get objects from all its ancestors available without having to write additional or duplicate code.

```
let pip = new Engineer('engineer', 'dude', 'SRE', 'DEVOPS');

pip.greeting();
pip._language = 'Go';
pip.programIn();
pip.farewell();
```

Public and Private Class Fields

While classes work well as they are now, TC 39 introduced a couple additional features that make working with classes smoother and without having to necessarily use the class constructor.

Public class fields provide an easier way to provide name and values for class variables.

Private class fields provide a way to encapsulate data so it's only available in the body of the class and causing an error when you attempt to use it outside.

Class fields (both public and private) are at Stage 3 in the TC39 process. They are already implemented in V8.

See Mathias Bynens article Public and private class fields for more information.