Patterns of Code Reuse

Notes

We will discuss in this section the main ways one can share functionality in Javascript. They are:

- **inheritance**
- **composition**
- **mixin**

Inheritance

Class inheritance is a fundamental component of object-oriented programming, but must be used with care and when appropriate. The key notion is that of a *subclass*, whereupon a class can inherit all of another class's properties and methods, and override those that it needs to or add new methods. The subclassing relation is often called an "is a" relation.

We will take a look at an example in a minute, but first of all, let us discuss a "newClass" method we will start using, which "creates" a new class.

```
function newClass(init, superclass) {
   var cls, proto;

   init = init || function() {};
   superclass = superclass || Object;
   proto = Object.create(superclass.prototype);
   cls = Object.create({}, {
      "prototype": { value: proto },
      "super": { value: superclass }
   });
   Object.defineProperty(cls.prototype, "class", { value: cls });
   cls.initialize = init;
   cls.new = function newObj() {
      var o = Object.create(proto);
      cls.initialize.apply(o, arguments);
      return o;
   };

   return cls;
}
```

This allows us to create classes and have them inherit from other classes.

The basic construction for inheritance would go something like this:

```
var Point = newClass(function init(x, y) {
   this.x = x;
   this.y = y;
});
Point.prototype.r = function() {
    return Math.sqrt(this.x * this.x + this.y * this.y);
};

var ColorPoint = newClass(
    function init(x, y, color) {
        Point.initialize.call(this, x, y);
        this.color = color;
    },
    Point  // <---- ColorPoint inherits from Point
);
ColorPoint.prototype.getColor = function() {
        return this.color;
};

var p1 = Point.new(2,3);
p1.x;
p1.y;
p1.r();

var p2 = ColorPoint.new(2, 3, "blue");
p2.x;
p2.y;
p2.r();
p2.getColor();
```

We set a class to inherit from another by passing the superclass as the second argument in newClass. Effectively this makes our subclass's prototype inherit from the superclass' prototype. So in our example, the ColorPoint class inherits all methods of the Point class.

This is then one way to "reuse code": The subclass' objects can reuse the methods of the superclass', provided the don't override them.

Disadvantages:

- The subclass now becomes dependent on the internals of the superclass. instance variables are shared, you have to be careful with method names, etc. Inheritance requires a very tight relationship between the class and its superclass.
- On deep inheritance chains, it might be hard to figure out for a given method which code will actually get executed. Looking at your code does not make that immediately clear, you have to find the "newest" superclass that implemented the method. Avoid long complicated inheritance chains.
- Cannot easily inherit from multiple classes.

Composition and Delegation

In object composition one object holds in its instance variables a reference to another object, and accesses it that way. It might optionally create methods that turn around and call methods of the composed object:

```
var ColorPoint = newClass(function init(x, y, color) {
      this.point = Point.new(x, y);
      this.color = color;
});
ColorPoint.prototype.setPoint = function(p) {
   this.point = p;
   return this;
};
// delegate calls to point
ColorPoint.prototype.getx = function() { return this.point.x; };
// Or even transparently
Object.defineProperty(ColorPoint.prototype, "x", {
   enumerable: true,
   get: function() { return this.point.x; },
   set: function(v) { this.point.x = v; return this; }
});
```

Advantages:

- The internals of the composed object are kept private; the object that included it just uses its interface. For instance in the above example, the Point class could have been implemented any old way and our ColorPoint wouldn't care. This leads to more loosely coupled designs.
- The class of the composed object can be selected dynamically at run time, as long as it still provides the interface needed. Inheritance on the other hand is more of a "compile-time" consideration.

Disadvantages:

- There is a certain level of indirection needed, as every time you need to access a property of the composed object you have to go through the enclosing object first.

One of our design principles is to favor composition over inheritance, as it tends to create more flexible designs.

Mixins

There is another possibility that can be helpful some times, called a mixin. It is typically used when you want to reuse some functionality. For instance, we can "mix in" the iterator prototype methods into an object with "next" and "hasNext" methods.

Mixing in involves directly copying all the methods from one or more objects into another, with something like this:

```
function mixin(target) {
   Array.prototype.slice.call(arguments, 1)
      .forEach(function(source) {
         Object.keys(source).forEach(function(key) {
            target[key] = source[key];
            });
        });

    return target;
}
```

Disadvantages:

- The link between the new methods and the old ones is broken. If something dynamically changes the old methods, the mixed in ones will not change.
- You end up with a lot of enumerable properties, that you may or may not have wanted there. (Though mixing methods into the prototype is probably a more stable idea)