Chapter 6

* Object patterns *

بإعداد : نسرين عبد السلام قروش .

introduction

* One of the most important aspects of writing maintainable code is being able to notice the recurring themes in that code and optimize them. This is an area where knowledge of design patterns can prove invaluable.

* A pattern is a reusable solution that can be applied to commonly occurring problems in software design - in our case - in writing JavaScript web applications. Another way of looking at patterns are as templates for how we solve problems - ones which can be used in quite a few different situations.

* Patterns are **not** an exact solution. It's important that we remember the role of a pattern is merely to provide us with a solution scheme. Patterns don't solve all design problems nor do they replace good software designers, however, they **do** support them. Next we'll take a look at some of the other advantages patterns have to offer.

Private and Privileged members

* in the JavaScript scope and closures article, by declaring a variable within a function, it is only available from within there. So, if we want a private property of the Kid object, we do it like this:

```
1.  // Constructor
2.  function Kid (name) {
3.      // Private
4.      var idol = "Paris Hilton";
5.  }
```

The idol property will only be available for the code inside the Kid function/object.

* A privileged method is a method having access to private properties, but at the same time publicly exposing itself (in JavaScript, also due to JavaScript scope and closures). You can delete or replace a privileged method, but you cannot alter its contents. I.e. this privileged method returns the value of a private property:

```
01.  // Constructor
02.  function Kid (name) {
03.      // Private
04.      var idol = "Paris Hilton";
05.
06.      // Privileged
07.      this.getIdol = function () {
08.          return idol;
09.      };
10.  }
```

The Module Pattern

* The Module pattern was originally defined as a way to provide both private and public encapsulation for classes in conventional software engineering.

In JavaScript, the Module pattern is used to further *emulate* the concept of classes in such a way that we're able to include both public/private methods and variables inside a single object, thus shielding particular parts from the global scope. What this results in is a reduction in the likelihood of our function names conflicting with other functions defined in additional scripts on the page.

```
var testModule = (function () {

  var counter = 0;

  return {

    incrementCounter: function () {
      return counter++;
    },

    resetCounter: function () {
      console.log( "counter value prior to reset: " + counter );
      counter = 0;
    }
  };

})();

// Usage:

// Increment our counter
testModule.incrementCounter();

// Check the counter value and reset
// Outputs: counter value prior to reset: 1
testModule.resetCounter();
```

Here, other parts of the code are unable to directly read the value of our `incrementCounter()` or `resetCounter()`. The counter variable is actually fully shielded from our global scope so it acts just like a private variable would - its existence is limited to within the module's closure so that the only code able to access its scope are our two functions. Our methods are effectively namespaced so in the test section of our code, we need to prefix any calls with the name of the module (e.g. "testModule").

When working with the Module pattern, we may find it useful to define a simple template that we use for getting started with it. Here's one that covers namespacing, public and private variables:

```
var myNamespace = (function () {

  var myPrivateVar, myPrivateMethod;

  // A private counter variable
  myPrivateVar = 0;

  // A private function which logs any arguments
  myPrivateMethod = function( foo ) {
      console.log( foo );
  };

  return {

    // A public variable
    myPublicVar: "foo",

    // A public function utilizing privates
    myPublicFunction: function( bar ) {

      // Increment our private counter
      myPrivateVar++;

      // Call our private method using bar
      myPrivateMethod( bar );

    }
  };

})();
```

Private Members for Constructors

* JavaScript is the world's most misunderstood programming language. Some believe that it lacks the property of *information hiding* because objects cannot have private instance variables and methods. But this is a misunderstanding. JavaScript objects can have private members. Here's how :

* In the constructor :

This technique is usually used to initialize public instance variables. The constructor's this variable is used to add members to the object.

```
function Container(param)
{
    this.member = param;
}
```

So, if we construct a new object

```
var myContainer = new
Container('abc');
```

then myContainer.member contains 'abc'.

* *Private* members are made by the constructor. Ordinary vars and parameters of the constructor become the private members.

```
function Container(param) {
    this.member = param;
    var secret = 3;
    var that = this;
}
```

This constructor makes three private instance variables: param, secret, and that. They are attached to the object, but they are not accessible to the outside, nor are they accessible to the object's own public methods. They are accessible to private methods. Private methods are inner functions of the constructor.

The private method dec examines the secret instance variable. If it is greater than zero, it decrements secret and returns true. Otherwise it returns false. It can be used to make this object limited to three uses.

```
function Container(param) {
    function dec() {
        if (secret > 0) {
            secret -= 1;
            return true;
        } else {
            return false;
        }
    }
    this.member = param;
    var secret = 3;
    var that = this;
}
```

mixins

* In JavaScript we can only inherit from a single object. There can be only one `[[Prototype]]` for an object. And a class may extend only one other class.

* a **mixin** is a class containing methods that can be used by other classes without a need to inherit from it.

In other words, a *mixin* provides methods that implement a certain behavior, but we do not use it alone, we use it to add the behavior to other classes.

* The simplest way to implement a mixin in JavaScript is to make an object with useful methods, so that we can easily merge them into a prototype of any class.

* For instance here the mixin `sayHiMixin` is used to add some “speech” for `User`:

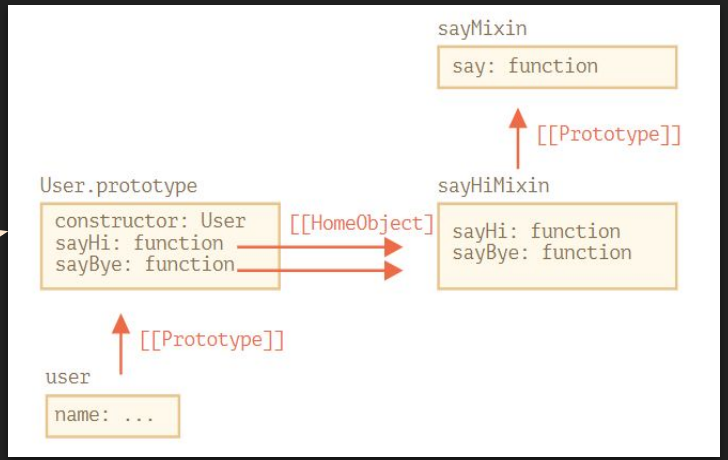
```
// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};
```

```
// usage:
class User {
  constructor(name) {
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, sayHiMixin);

// now User can say hi
new User("Dude").sayHi(); // Hello Dude!
```

There's no inheritance, but a simple method copying. So `User` may inherit from another class and also include the mixin to "mix-in" the additional methods, like this:

```
class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);
```

Here's the diagram :



scope-safe constructors

* A scope-safe constructor is designed to return the same result regardless of whether it's called with or without `new`, so it doesn't suffer from those issues.

* Most built-in constructors, such as `Object`, `Regex` and `Array`, are scope-safe. They use a special pattern to determine how the constructor is called.

* If `new` isn't used, they return a proper instance of the object by calling the constructor again with `new`. Consider the following code:

```javascript
function Fn(argument) {

  // if "this" is not an instance of the constructor
  // it means it was called without new
  if (!(this instanceof Fn)) {

    // call the constructor again with new
    return new Fn(argument);
  }
}
```

So, a scope-safe version of our constructor would look like this:

```javascript
function Book(name, year) {
  if (!(this instanceof Book)) {
    return new Book(name, year);
  }
  this.name = name;
  this.year = year;
}

var person1 = new Book("js book", 2014);
var person2 = Book("js book", 2014);

console.log(person1 instanceof Book);     // true
console.log(person2 instanceof Book);     // true
```

summary

* In JavaScript, there are numerous methods for creating and composing objects.

* While there is no formal idea of private properties in JavaScript, you can define data or functions that are only available from within an object.

* The module pattern can be used to hide data from the outside world for singleton objects.

* You can establish local variables and functions that are only available by the newly generated object using an immediately invoked function expression (IIFE).

* Privileged methods are those that have access to private data on the object.

* You may also define variables in the constructor function or use an IIFE to produce private data that is shared across all instances to create constructors with private data.

* Mixins are a great method to add functionality to objects without having to worry about inheritance.

* A mixin replicates properties from one object to another so that the receiving object can benefit from the functionality of the supplying object without inheriting it.

summary

* Mixins, unlike inheritance, don't let you know where the capabilities came from after the object has been built.

* Mixins are best utilized with data properties or little amounts of functionality because of this.

* When you want to get more functionality and know where it came from, inheritance is still the way to go.

* Constructors that can be called with or without new to create a new object instance are known as scope-safe constructors.

* This pattern takes use of the fact that this is an instance of the custom type as soon as the constructor starts to execute, allowing you to change the behavior of the constructor depending on whether you used the new operator or not.