chapter 6

* Object patterns *

بإعداد : سفيان يوسف الفازع .

introduction

* One of the most essential components of designing maintainable code is being able to understand and optimize recurrent patterns. Design patterns can be quite helpful in this situation.

*A pattern is a reusable solution that may be used to solve common challenges in software development, such as building JavaScript web apps in our example. Another approach to think of patterns is as templates for how we handle issues, which may be used to a variety of scenarios.

* Patterns aren't always the best answer. It's critical to remember that a pattern's sole purpose is to offer us with a solution scheme. Patterns don't solve all design issues, and they don't replace excellent software designers, but they do help them out. Then we'll look at some of the additional benefits that patterns have to offer.

Private and Privileged members

* As mentioned in the JavaScript scope and closures article, defining a variable within a function restricts access to that function. So, if we want the Kid object to have a private attribute, we do it like this:

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

Only the code within the Kid function/object will have access to the idol property.

* A privileged method is one that has access to secret properties while also exposing itself to the public (in JavaScript, also due to JavaScript scope and closures). A privileged method can be deleted or replaced, but its contents cannot be changed. In other words, the value of a private property is returned via this privileged method:

```
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

*** In traditional software engineering, the Module pattern was initially designed as a mechanism to offer both private and public encapsulation for classes.**

The Module pattern is used in JavaScript to better simulate the notion of classes by allowing us to incorporate both public and private functions and variables within a single object, concealing certain areas from the global scope. As a result, the risk of our function names clashing with other functions declared in extra scripts on the page is reduced.

```
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();
```

Other portions of the code are unable to read the value of our incrementCounter() or resetCounter() functions directly (). The counter variable is totally insulated from our global scope, acting exactly like a private variable - its existence is confined to within the module's closure, allowing only our two functions to access its scope.Because our methods are essentially namespaced, we must prefix any calls in the test part of our code with the module name (e.g. "testModule").

When working with the Module pattern, it might be helpful to create a basic template to utilize as a jumping off point. Here's one that takes care of namespacing, public variables, 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

* The most misunderstood programming language in the world is JavaScript. Some argue that because objects cannot have private instance variables and methods, it lacks the feature of information hiding. This, however, is a misconception. Private members are possible in JavaScript objects. Here's how to do it:

* In the constructor :

This method is commonly used to set up public instance variables. This variable in the constructor 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'.

** The constructor creates private members. The constructor's ordinary vars and arguments become private members.*

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

There are three private instance variables created by this constructor: param, secret, and that. They're tied to the object, but they're not visible to the outside world, and they're not available to the object's public functions either. Private methods have access to them. The constructor's private methods are internal functions.

The secret instance variable is examined by the private function dec. It decrements secret and returns true if it is larger than zero. Otherwise, false is returned. It can be used to limit the use of this item to three.

```
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

* We can only inherit from one object in JavaScript. For each item, there can only be one [[Prototype]]. A class can only extend one other class.

* A mixin is a class that has methods that other classes can utilize without having to inherit from it.

To put it another way, a mixin contains methods that implement a certain behavior, but we don't use it on its own; instead, we use it to add the behavior to other classes.

* In JavaScript, the easiest approach to construct a mixin is to create an object with helpful methods that can be readily merged into a prototype of any class.

* In this case, the sayHiMixin mixin is used to add some "speak" for User:

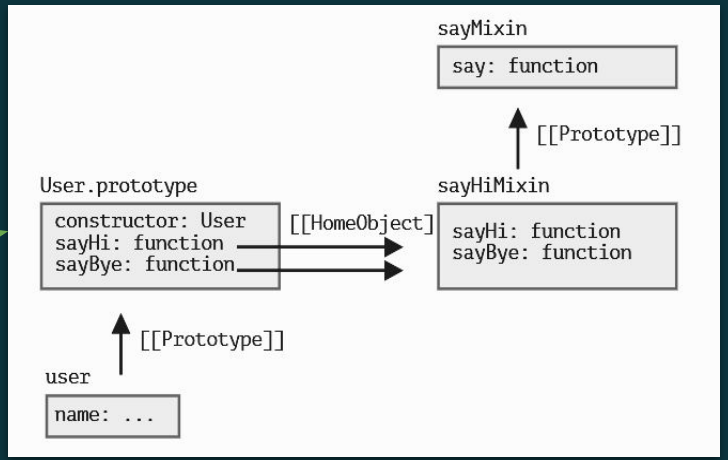
```
// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};

```

```
// 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!
```

No inheritance is used; instead, a simple method copying is used. As a result, User can inherit from another class while also include the mixin to "mix-in" the extra methods, as seen below:

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

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

Here's the diagram :



scope-safe constructors

* Because a scope-safe constructor is designed to deliver the same result whether it is used with or without new, it avoids these problems.

* Most built-in constructors are scope-safe, including Object, Regex, and Array. To establish how the constructor is invoked, they employ a unique pattern.

* If new isn't used, they call the constructor again with new to return a correct instance of the object. Consider the code below:

```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

* There are various techniques for constructing and constructing objects in JavaScript.

* While JavaScript does not have a formal concept of private properties, you may construct data or methods that are only accessible from within an object.

* For singleton objects, the module pattern may be used to hide data from the outside world.

* Using an instantly called function expression, you may create local variables and functions that are exclusively available to the newly constructed object (IIFE).

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

* To build constructors with private data, you may specify variables in the constructor function or use an IIFE to generate private data that is shared across all instances.

* Using mixins, you can add functionality to objects without worrying about inheritance.

* A mixin copies properties from one object to another, allowing the receiving object to benefit from the providing object's functionality without having to inherit it.

summary

* Unlike inheritance, mixins don't reveal where the capabilities originated from once the object has been constructed.

* Because of this, mixins are best used with data attributes or small bits of functionality.

* Inheritance is still the way to go if you want additional functionality and know where it came from.

* Scope-safe constructors are those that may be invoked with or without new to produce a new object instance.

* This pattern takes use of the fact that this is an instance of the custom type as soon as the constructor begins to run, allowing you to vary the constructor's behavior based on whether or not you used the new operator.