Object Patterns

الطالبة : بشرى عصام منكوسة

رقم القيد:172032

Contents

Object Patterns

- There are various patterns for creating objects in JavaScript, and there are usually multiple ways to achieve the same task.
- Whenever you want, you can create your own custom kinds or generic objects.
- Inheritance can be used to share behavior between objects, Other strategies, such as mixins, can also be used.
- Advanced JavaScript features can also be used to prevent an object's structure from being changed.
- The patterns covered in this chapter provide you with effective methods for managing and creating objects based on your use cases.

Private and Privileged

- In JavaScript, all object properties are public, and there's no way to explicitly say that a property shouldn't be accessed from outside the object. However, there may come a time when you don't want your data to be public.
- When an object requires a value to establish its state, for example, changing that data without the object's awareness causes havoc with the state management process.
- Using naming standards is one approach to avoid this. When properties are not intended to be public, it's usual to prefix them with an underscore (such as this. name).
- There are, however, methods of hiding data that do not rely on convention and are thus more "bulletproof" in terms of preventing the change of private data.

Module pattern

- **The module pattern is an object-creation pattern for creating singletons with private data. The most basic way is to employ an object-returning instantly invoked function expression (IIFE).**
- **An IIFE is a function expression that is defined and then called to provide a result immediately.**
- **Any number of local variables that aren't accessible from outside the function can be found in that function expression. Because the returned object is defined within that function, the data is accessible to the object's methods. (All of the IIFE's objects have access to the same local variables.) Privilege methods are those that have this kind of access to private data.**
- **Here's the basic format for the module pattern:**

```
var yourObject = (function() {

    // private data variables

    return {
        // public methods and properties
    };

}());
```

Module pattern

- **An anonymous function is created and executed instantly in this pattern. (Notice the extra parentheses at the function's end.)This syntax allows you to run anonymous functions right away.)That is, the function exists for a brief duration before being executed and then deleted. IIFEs are a popular JavaScript pattern, thanks in part to their use in the module pattern.**
- **Regular variables can be used as de facto object attributes that aren't visible to the public using the module pattern. Closure functions are created as object methods to do this. Closures are just functions that access data that isn't within their scope. When you access a global object in a function, such as a window in a web browser, for example ,**
- **That function is attempting to access a variable that is not in its scope. The distinction with the module function is that the variables are declared within the IIFE, and those variables are accessed by a function that is likewise declared within the IIFE.**

Example

Using the module pattern, this code builds the person object. The object's age variable 1 works as a private attribute. It can't be accessed directly from outside the object, but the object methods can use it. The object has two privileged methods: getAge() 2, which reads the age variable's value, and growOlder() 3, which increases age. Because the variable age is declared in the outer function in which they are defined, both of these methods can access it directly.

```
> var person = (function() {
    var age = 25;
  return {
  name: "Mohammed",
  getAge: function() {
  return age;
  },
    growOlder: function() {
  age++;
  }
  };
  }());
  console.log(person.name);
  console.log(person.getAge());
  person.age = 100;
  console.log(person.getAge());
  person.growOlder();
  console.log(person.getAge());
  Mohammed
  25
  25
  26
```

Private Members For Constructors

The module pattern is fantastic for defining separate objects with private properties, but what about custom types with private properties of their own? To build instance-specific private data, you can use a pattern similar to the module pattern inside the constructor.

```
function Person(name) {
var age = 25;
this.name = name;
 this.getAge = function() {
return age;
};
this.growOlder = function() {
age++;
};
}
var person = new Person("Mohammed");
console.log(person.name);
console.log(person.getAge());
person.age = 100;
console.log(person.getAge());
person.growOlder();
console.log(person.getAge());
```

```
Mohammed
25
25
26
```

In this code, the Person constructor has a local variable, age. That variable is used as part of the getAge() 1 and growOlder()2methods. When you create an instance of Person, that instance receives its own age variable, getAge() method, and growOlder() method. In many ways, this is similar to the module pattern, where the constructor creates a local scope and returns the this object. placing methods on an object instance is less efficient than doing so on the prototype, but this is the only approach possible when you want private, instance-specific data.

Mixins

Although pseudo classical and prototypal inheritance are common in JavaScript, there is also a sort of pseudo inheritance that is achieved by mixins. When one object inherits the properties of another without changing the prototype chain, this is known as a mixin.

The properties of the second item (the supplier) are received by the first item (a receiver) by copying them directly.

```
function mixin(receiver, supplier) {
for (var property in supplier) {
if (supplier.hasOwnProperty(property)) {
receiver[property] = supplier[property]
}
}
return receiver;
}
```

Example

- The receiver and supplier arguments are passed to the mixin() function.
- The function's objective is to replicate all of the supplier's enumerable attributes to the receiver. This is done with a for-in loop that iterates over the properties in supplier and then assigns the value of each property to a receiver property with the same name.
- Keep in mind that this is a shallow copy, so if a property contains an object, both the supplier and the receiver will point to it.
- This approach is widely used to add new behaviors to JavaScript objects that already exist on other objects.
- Any object can use the EventTarget type to handle basic events.
- You can add 1 listener, remove 3 listeners, and fire events 2 on the object directly. The event listeners are saved in the _listeners property, which is only created the first time addListener() is invoked (this makes it easier to mix in).

```
function EventTarget(){
}
EventTarget.prototype = {
constructor: EventTarget,
 addListener: function(type, listener){

if (!this.hasOwnProperty("_listeners")) {
this._listeners = [];
}
if (typeof this._listeners[type] == "undefined"){
this._listeners[type] = [];
}
this._listeners[type].push(listener);
},
 fire: function(event){
if (!event.target){
event.target = this;
}
if (!event.type){ y
throw new Error("Event object missing 'type' property.");
}
if (this._listeners && this._listeners[event.type] instanceof Array){
var listeners = this._listeners[event.type];
for (var i=0, len=listeners.length; i < len; i++){
listeners[i].call(this, event);
}
}
},removeListener: function(type, listener){
if (this._listeners && this._listeners[type] instanceof Array){
var listeners = this._listeners[type];
for (var i=0, len=listeners.length; i < len; i++){
if (listeners[i] === listener){
listeners.splice(i, 1);
break;
}
}
}
}
};
▸ {constructor: ƒ, addListener: ƒ, fire: ƒ, removeListener: ƒ}
```

1

2

3

Scope - Safe Constructors

- One frequent problem when developing in Object Oriented JavaScript is how to handle the removal of the "new" keyword when creating an instance of a class. It's often expected that an object should be viably instantiated two ways, with the "new" keyword and without.
- Because all constructors are merely functions, you can call them without using the new operator and thereby change this's value. This can have unforeseen consequences, like as the constructor throwing an error in strict mode or being forced to the global object in nonstrict mode.

Example:

```
> function Person(name) {
  this.name = name;
  }
  Person.prototype.sayName = function() {
  console.log(this.name);
  };
   var person1 = Person("Mohammed");
  console.log(person1 instanceof Person);
  console.log(typeof person1);
  console.log(name);
  false
  undefined
  Mohammed
```

- In this case, name is created as a global variable because the Person constructor is called without new 1. Keep in mind that this code is running in nonstrict mode, as leaving out new would throw an error in strict mode.
- The fact that the constructor starts with a capital letter normally means that new should come before it, But what if you want to allow this use case while still having the function work? Because they are constructed to be scope safe, several built-in constructors, such as Array and RegExp, also work without new.
- A scope-safe constructor can be used with or without new and will always return the same type of object.
- The newly generated object represented by this is already an instance of the custom type represented by the constructor when new is called with a function.

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. 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 comes from, inheritance is still the way to go.
- Constructors that can be called with or without new to produce 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.