OBJECT PATTERNS

Object Patterns

- Private and Privileged Members
- Mixins
- Scope-Safe Constructors

إعداد.

رتاج عمر بن طاهر.

OBJECT PATTERNS

- *object-oriented* **design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved**

- ***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.***
- ***You can use inheritance or other approaches, like as mixins, to share behavior between objects. Advanced JavaScript features can also be used to prevent an object's structure from being changed.***

Private and Privileged Members

- *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. At one moment or another

- However, you may not want your information to be made 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.

THE 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. The basic format for the module pattern is as follows:

```
Var yourObject =
(function() {
 // private data
variables
 return {
 // public methods and
properties
 };
u }
());
```

An anonymous function is created and executed instantly in this pattern. (Note the extra parentheses at the end of the function; this syntax allows you to execute 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.

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 . Consider the following scenario:

```
 function Person(name) {
 // define a variable only accessible inside of the Person constructor
 var age = 25;
 this.name = name;
u this.getAge = function() {
 return age;
 };

Object Patterns 83
v this.growOlder = function() {
 age++;
 };
}
var person = new Person("Nicholas");
console.log(person.name); // "Nicholas"
console.log(person.getAge()); // 25
person.age = 100;
console.log(person.getAge()); // 25
person.growOlder();
console.log(person.getAge()); // 26
```

The Person constructor has a local variable, age, in this code. The getAge() u and growOlder() v functions both use the variable. When you construct a Person object, it comes with its own age variable, as well as the getAge() and growOlder() methods. The constructor generates a local scope and returns the this object, which is comparable to the module pattern in many respects.

MIXINS

- Although pseudoclassical inheritance and prototypal inheritance are common in JavaScript, *mixins* can also be utilized to achieve pseudoinheritance
- When one object inherits the properties of another without altering 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.
- **NOTE** Keep in mind that *Object.keys()* returns only enumerable properties. If you want to also copy over nonenumerable properties, use *Object.getOwnPropertyNames()* instea

- *Traditionally, you'd use a function like this to construct mixins.*

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

The mixin() **function accepts two arguments: the receiver and the supplier. The goal of the function is to copy all enumerable properties from**
the supplier onto the receiver. You accomplish this using a for-in loop
that iterates over the properties in supplier and then assigns the value
Object Patterns 85
of that property to a property of the same name on receiver.

Scope-Safe Constructors

- **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. You may have come across the following example:**

example:

```
function Person(name) {
 this.name = name;
}
Object Patterns 91
Person.prototype.sayName = function()
{
 console.log(this.name);
};
u var person1 = Person("Nicholas");
// note: missing "new"
console.log(person1 instanceof
Person); // false
console.log(typeof person1);
// "undefined"
console.log(name); // "Nicholas"
```

. Keep in mind that this code is executing in nonstrict mode, because in strict mode, leaving out new would result in an error. The fact that the constructor starts with a capital letter generally means that new should come before it, but what if you want to accept this use case and have the function work without new?

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. As a result, you can use instance of to see if new was used in the function call.

```
function Person(name) {
 if (this instanceof Person) {
 // called with "new"
 } else {
 // called without "new"
 }
}
```

- You can control what a function does depending on whether it's called with new or not by using a pattern like this. You may wish to handle each situation differently, but you'll most likely want the function to behave consistently (*frequently, to protect against accidental omission of new*). This is how a scope-safe version of Person looks:

```
        function Person(name) {
 if (this instanceof Person) {
 this.name = name;
 } else {
 return new Person(name);
 }
}
```

- When new is used in this constructor, the name property is assigned as ***usual***.
- The constructor is invoked recursively via ***new*** to produce a valid instance of the object if ***new*** isn't used. The following are equal in this way:

```
var person1 = new Person("Nicholas");
var person2 = Person("Nicholas");
console.log(person1 instanceof Person); // true
console.log(person2 instanceof Person); // true
```

- Creating new objects without using the new operator is becoming more common as an effort to curb errors caused by omitting new. JavaScript itself has several reference types with scope-safe constructors, such as Object, Array, ***RegExp*,** and Error.

- ***Summary :***
- There are many different ways to create and compose objects in JavaScript. While JavaScript does not include the formal concept of private

properties, you can create data or functions that are accessible only from within an object. For singleton objects, you can use the module pattern to hide data from the outside world. You can use an immediately invoked function expression *(IIFE)* to define local variables and functions that are accessible only by the newly created object. Privileged methods are methods on the object that have access to private data. You can also create constructors that have private data by either defining variables in the constructor function or by using an IIFE to create private data that is shared among all instances

- Mixins are a powerful way to add functionality to objects while avoiding inheritance. A mixin copies properties from one object to another
so that the receiving object gains functionality without inheriting from
the supplying object. Unlike inheritance, mixins do not allow you to identify where the capabilities came from after the object is created. For this
reason, mixins are best used with data properties or small pieces of functionality. Inheritance is still preferable when you want to obtain more
functionality and know where that functionality came from.

- Scope-safe constructors are constructors that you can call with or
without new to create a new object instance. This pattern takes advantage
of the fact that this is an instance of the custom type as soon as the constructor begins to execute, which lets you alter the constructor's behavior
depending on whether or not you used the new operator.