Guide

PATTERNS

LEVON ISAJANYAN

1

Patterns are layout or templates which generate thing .
There are three types of patterns

1. *Design pattern*

2. *Coding patterns*

3. *Antipattern*

Design pattern

Singleton, factory, decorator, observer, prototype, module, middleware

Coding patterns

Coding patterns are JS specific related patterns, which consider to function using in variety purposes

Antipatterns

They cause more problems to solve problems .

Do not use global variable or minimize using global variables

Example

// This example is antipattern because variable is not declared and it can be part of global object (window object in browser)

```
function foo (){
    a = 'Some value'
    return a
}

// Correct example

function foo (){
    let a = 'some value'
    return a
}
```

Antipattern

3

Side effect which can impact to global variables

```
let a = 1
b = 3
(function () {
 c = 3
}())

delete a  //  false
delete b  //  true
delete x  //  true
```

Antipattern

4

Iterate over object

```
// Antipattern
for (element in object){
    console.log(object[element])
}
```

```
// Correct way is to use hasOwnProperty
for (let element in object) {
    if (Object.prototype.hasOwnProperty.call(object, element) {
        console.log(element : element[i])
    }
}
```

Antipattern

5

Create object

```
    // Antipattern because assigned variable maybe in other component you can use with another value
    let a = new Object();
    a.go = 'some value'

    // correct
    let a = { go : null }
    a.go = 'some value'
```

6

Create function object

```
    // Antipattern without new value assign to global
window object
    function createObject (value) {
        this.objectKey = value
    }


    let newFunc = createObject (' jame ')
    console.log(typeof newFunc)  //undefined
    console.log(window.objectKey) // jame

    // Correct

    let newFunc = createObject('Yea!')
    console.log(typeof newFunc) // Object
    console.log(newFunc) // Yea!
```

7

Array literal

```
// Antipattern
let a = new Array('its', 'bit')
console.log(typeof a)  // Object, because array is object
console.log(a.contructor === Array) // true

// Antipattern 2
let a = new Array(3)
console.log(a.length)  // 3 this will create array with
three elements but each element value will be undefiend
console.log(a[1]) // undefined

// Correct
let a = [3]
console.log(a.length)  // 1
console.log(a[0]) // 3
```

8

Callback pattern

```
function foo (callback){
    callback()
}

function invokeMe ( arg){
    alert(arg)
}
```

foo(invokeMe) // after invoking foo function invokeMe will be called as function Callback

9

Callback pattern examples

```
let boo = function (){
    console.log('invoke')
}

setTimeout(boo, 500) // Settimeout function is browser global function an dthis is callback

document.addEventListener('click', boo, false)  // In this case callBack will be stored but not will be called till we shall call it from UI side

let setupFunction  = function (){
    let number = 0
    return function (){
            return count = count + 1
    }
}

let next = setupFunction () // number will be stored
next() // 1
```

10

Self definig functions

```
let a = function (){
    alert('invoke')
    a = function (){
            alert('double invoke')
    }
}

a()  // invoke
a()  // double invoke
```

11

Imidiate invoking function or

```
(function (arg1, arg2) {
    console.log(arg1, arg2)
}, ('Fisrts', 'Second'))

// Result will be console.log('First', 'Second') and this will be invoked when js engine will defined it
```

12

Imidiate object initilization

```
({
    a: 1,
    b: 3,
    formula: function (argument1, argument2) {
        return (this.a + this.b) * (this.a - this.b) +
argument1 + argument2
    },
    total: function (total1, total2) {
        let result = this.formula(total1, total2)
        console.log(`Total Formula value is ${result}`)
    }
}).total(10, 20)
// Result will be console.log(Total Formula value is 22)
let t = ({
    a: 1,
    b: 3,
    formula: function () {
        return this.a + this.b
    },
    total: function () {
        this.formula()
        return this
    }
}).total()
console.log(t.formula()) // t global object, which returned inside last function scope
```

13

Call function

```
let sayHi = function (who) {
    return who
}

let Person = function  () {
    sayHi: function (who) {
            return who
    }
}
sayHi.call(Person, 'Jack')
```

Singleton

```
let object = {
    key: 'some_value'
}

let newObject = object
let anotherObject = object

newObject === anotherObject  // true
```

SIngleton use data srtructure type linked list. Which say that if you define one variable to another it doesn't create new place in memory . New defined variable just give refference to first created value . In out example newObject and anotherObject variable just contain refference to object variable value in memeory of RAM.

And SIngleton means that if object created then its unique and it cant be present in other of place memory if you assign will take new place in memeory and will assign to this place previously created value. New memory just will give you refference to another memory slot

Factory pattern

Instead of this you can use Class constructor

```
function carMaker () {}

carMaker.prototype.startEngine = function () {
  return this.doors
}

carMaker.factory = function (type) {
  var constr = type,
      newcar;
  // error if the constructor doesn't exist
  if (typeof carMaker[constr] !== "function") {
    throw {
      name: "Error",
      message: constr + " doesn't exist"
    };
  }
  // at this point the constructor is known to exist
  // let's have it inherit the parent but only once
```

```
      if (typeof carMaker[constr].prototype.startEngine !==
"function") {
        carMaker[constr].prototype = new carMaker();
       }
        // create a new instance
        newcar = new carMaker[constr]();
       // optionally call some methods and then return...
return newcar;
       };
     // define specific car makers
     carMaker.Compact = function () {
      this.doors = 4;
     };
     carMaker.Convertible = function () {
      this.doors = 2;
     };
     carMaker.SUV = function () {
      this.doors = 24;
     };

     var corolla =  carMaker.factory('Compact');
```

```
var solstice = carMaker.factory('Convertible');
var cherokee = carMaker.factory('SUV');
console.log(carMaker)
corolla.startEngine(); // "Vroom, I have 4 doors"
solstice.startEngine(); // "Vroom, I have 2 doors"
cherokee.startEngine(); // "Vroom, I have 17 doors"
```

// Difference between Factory pattern and Class constructor pattern are in `this` keyword in Class constructor you can access to Constructor function , but in Factory pattern you can't value will be undefined

Strategy pattern

```
    function doSomething() {
        switch (this.firstStrategy)
                case simepleStaretegy
                        // do smthing
                        break
                case advancedStartegy
                        // do smthing
                        break
    }


    // Use this with algorythms , which is convenoent
method
```

Facade pattern

```
function stopSmthing (arg) {

        arg.preventDefault();

        arg.stopPropogation();

}



// Call function this for example in variety parts of your code  to

stop smthing in browser part
```

Module pattern

See page 11 (eleven) , becuase they are similiar . But famous example of this pattern implementation is NPM modules or packages

```
var myModule = (function() {
  'use strict';

  var _privateProperty = 'Hello World';
  var publicProperty = 'I am a public
property';

  function _privateMethod() {
    console.log(_privateProperty);
  }

  function publicMethod() {
    _privateMethod();
  }

  return {
    publicMethod: publicMethod,
```

```
 publicProperty: publicProperty
  };
}());

myModule.publicMethod();
                  // outputs 'Hello World'
console.log(myModule.publicProperty);
// outputs 'I am a public property'
console.log(myModule._privateProperty);
// is undefined protected by the module
closure
myModule._privateMethod();
// is TypeError protected by the module
closure
```

Conclusion

There are about 23 types of pattern some of them are popular another are not satisfy todays requirements . This little articles was a result of inspire which gave me Github user ***https://github.com/nairihar*** . During his speech in JSCONFAM 2019 .

Creditinals ***https://github.com/nairihar***
Stojan Stefanov - JS patterns (2010)