Factory Pattern JavaScript

The Factory pattern is one of the fundamental creational patterns alongside the pattern The Simple Factory is just a Singleton (or just an static class in most programming languages, but in JavaScript, they’re essentially the same) that has one or more functions for creating and returning objects.

Example

var myfactory = (function()

{

let instance;

function createObject()

{

var object = new Object({ name : 'I m factory Object'})

return object;

}

return{

getObject : function()

{

instance = createObject();

return instance

}

}

})();

var factoryobj = myfactory.getObject();

console.log(factoryobj == myfactory.getObject());

console.log(factoryobj)

Observer Pattern

The [observer pattern](https://en.wikipedia.org/wiki/Observer_pattern) is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

function Click() {

this.handlers = []; // observers

}

Click.prototype = {

subscribe: function(fn) {

this.handlers.push(fn);

},

unsubscribe: function(fn) {

this.handlers = this.handlers.filter(

function(item) {

if (item !== fn) {

return item;

}

}

);

},

fire: function(o, thisObj) {

var scope = thisObj || window;

this.handlers.forEach(function(item) {

item.call(scope, o);

});

}

}

//

function run() {

var clickHandler = function(item) {

log.add("fired: " + item);

};

var click = new Click();

click.subscribe(clickHandler);

click.fire('event #1');

click.unsubscribe(clickHandler);

click.fire('event #2');

click.subscribe(clickHandler);

click.fire('event #3');

}

Mediator Pattern

The Mediator pattern provides central authority over a group of objects by encapsulating how these objects interact. This model is useful for scenarios where there is a need to manage complex conditions in which every object is aware of any state change in any other object in the group.

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

var Participant = function(name) {

this.name = name;

this.chatroom = null;

};

Participant.prototype = {

send: function(message, to) {

this.chatroom.send(message, this, to);

},

receive: function(message, from) {

log.add(from.name + " to " + this.name + ": " + message);

}

};

var Chatroom = function() {

var participants = {};

return {

register: function(participant) {

participants[participant.name] = participant;

participant.chatroom = this;

},

send: function(message, from, to) {

if (to) { // single message

to.receive(message, from);

} else { // broadcast message

for (key in participants) {

if (participants[key] !== from) {

participants[key].receive(message, from);

}

}

}

}

};

};

function run() {

var yoko = new Participant("Yoko");

var john = new Participant("John");

var paul = new Participant("Paul");

var ringo = new Participant("Ringo");

var chatroom = new Chatroom();

chatroom.register(yoko);

chatroom.register(john);

chatroom.register(paul);

chatroom.register(ringo);

yoko.send("All you need is love.");

yoko.send("Wher r you John.");

john.send("Hey, no need to broadcast", yoko);

paul.send("Ha, I heard that!");

ringo.send("Paul, what do you think?", paul);

 }