**JavaScript Design pattern**

Design Patterns solve specific programming challenges regarding usability and maintainability.

Writing “BAD” JavaScript is easy, writing “Good” JavaScript is not too hard

Let’s understand a little about JavaScript objects before jumping into design pattern

**Creation of object in JavaScript**

There are three basic ways in which we can create the object

1. var task = {} ;

2. var task = Object.create(Object.prototype)

3. var task = new Object()

**Defining of property of objects**

The most common ways in which we can define properties are

Example:

1. Property can be added using [ . ]

var task = {}; \\ task is an object

task.title = “this is a task”; \\ title is the property of the object

task.description = “this is a description” \\ description is the property of the object

2. Property can be added inside the object

var task = {

title: “ this is a task”,

description : ”this is a description”

};

3. Property can be added using the define property method of the object

var task = {

title: “ this is a task”,

description : ”this is a description”

};

Object.defineproperty(task,”toString”,{

Value : function (){

Return this.title + “ ” + this.description

},

Writable: true, // makes the function writable

Enumerable: true, // makes the function enumerable

Configurable: true // makes the function configurable

});

Console.log (task.toString());

Output: this is a task this is a description

**Uses of writable, enumerable, configurable**

**Writable**: in JavaScript any one can assign a value to any variable from any point to avoid that we use writable

For example:

Task.toString = “hi”

In the above example u can see that toString was function of the object but we have made toString in to property

**Enumerable**: when u want to hide the functions of the object we set enumerable to false

For example

Console.log(Task); \\ when enumerable set to true

Output: {title: “this is a task”, description: “this is a description”,toString :[function]}

Console.log(Task); \\ when enumerable set to False

Output: { title: “this is a task”, description:”this is a description]}

**Configurable**: once a property is create we can override the same property to have control over it we this method is used

For example:

Object.defineProperty(task, 'toString',{

value : function () {

Return this.title +' '+this.description;

},

Writable: false,

Enumerable: true,

Configurable: false

});

Object.defineProperty(task,'toString',{

Enumerable: true

});

When configurable is set to the second set of code will be executed

If it’s set to false it throws an error cannot redefine property

**Inheritance**

JavaScript does not have "methods" in the form that class-based languages define them. In JavaScript, any function can be added to an object in the form of a property. An inherited function acts just as any other property

Example

var urgentTask = Object.create(task);

Object.defineProperty(urgentTask, 'toString',{

value : function () {

return this.title +' urgent task ';

},

writable:false

enumerable:false,

configurable:false

});

console.log(urgentTask.toString())

Now we know the basics of JavaScript objects and properties let’s jump in to design patterns

There are three major classification design patterns

1. creational design pattern :

Creational design pattern deals with the creation of objects

1. Structural design pattern :

Structural design pattern are concerned with how objects are made up and simplify relationships between objects

1. Behavioral design pattern :

Behavioral design pattern are concerned with the assignment of responsibilities between objects and how they communicate

Let’s get start with the

**Creational design pattern**

Creational design pattern deals with the creation of objects

 Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

1. Constructor pattern :

Use to create new objects with their own object scope

When we are creating a new object with a “new” key word

Example:

var task = function(name){

this.name = name;

this.completed = false;

this .completed = function () {

this.completed =true;

}

this.save = function () {

console.log('saving task:' + this.name);

}

}

var task1 = new task("this is a test1");

var task2 = new task("this is a test2");

var task2 = new task("this is a test3");

task1.completed();

task2.save();

task3.save();

draw back: every time an instance is created of the object the completed function is recreated which is not efficient which is (this.Save) in the above example

that is why prototype is used

2. Prototype design pattern

Prototype is an encapsulation of properties that an object links to

Syntax

ClassName.prototype.methodName = function (argument) {

};

Example:

var task = function(name){

this.name = name;

this.completed = false;

}

task.prototype.completed = function (){

console.log("completed task by " + this.name )

this.completed =true;

}

task.prototype.save = function (){

console.log('saving task:' + this.name);

}

var task1 = new task("this is a test1");

var task2 = new task("this is a test2");

var task2 = new task("this is a test3");

task1.completed ();

task2.save ();

task3.save ();

In the above example:

* completed and save function are not recreate every time a task instance is created the prototype has access to it
* much more efficient than constructor pattern

3. Module pattern

Modular pattern is a simple way to encapsulate a service

Every time a service needs to called we can use this pattern

There are two types of modular pattern one is simple modular pattern and reveling modular pattern

Simple modular pattern example:

var repo = function(){

var db = {};

return {

Get: function (id) {

Console.log(id);

Return {

Name: "new name"

}

},

save: function (task){

Console .log ("this is a Task" + task);

}

}

};

Reveling module pattern Example:

var repo = function(){

var db = {};

var get = function(id){

Console.log(id);

return {

name: "new name"

}

};

var save = function (task){

console .log("this is a rask" + task);

Return {

Get: get,

Save: save

}

}

};

1. Singleton pattern

* Single ton pattern is used to restrict an object to one instance of the object across the application
* It remembers the last time you used the entity and Hands the same instance back
* Ensure a class has only one instance

Example:

var taskRepo = (function () {

var taskRepo;

function createreop() {

var taskRepo = new Object("Task");

return taskRepo;

}

return {

getInstance :function () {

if(!taskRepo){

taskRepo= createreop();

}

return taskRepo;

}

};

})();

var repo1 = taskRepo.getInstance();

var repo2 =taskRepo.getInstance();

in the above example u can see that if a task repo is created once the same instance is used for the next time if the instance is not created the a new instance is crated

1. Factory pattern

Factory pattern used to simplify object creation

Every time you want to do a set or a group of work use a factory pattern

Example:

var repoFactory = function (){

this.getReop= function (repoType){

if(repoType == 'Task')

{

var taskRepo = require('./taskRepository');

return taskRepo;

}

if(reopType == 'user')

{

var userRepo = require('./userfactory');

return userRepo;

}

}

}

In The above example

Whenever a set of task needs to be done put all that in a separate file and call that file based on the task

By doing this we can avoids lot of confusion of the task and recreating the same task

1. **Structural Design Pattern**

Structural design patterns are concerned with how objects are made up

And simplify relationships between objects

Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities

We deal with it in two ways

1. Extend functionality

2. Simplify functionality

Let’s look at few structural design patterns

1. **Decorator pattern**

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class

Example :

**Simple Decorator**

var task = function(name){

this.name = name;

this.completed = false;

}

task.prototype.completed = function (){

console.log("completed task by " + this.name )

this.completed =true;

}

task.prototype.save = function (){

console.log('saving task:' + this.name);

}

var task1 = new task("this is a test1");

task1.completed();

task1.save();

var urgentTask= new task("this is test2");

urgentTask.priority =2 ;

urgentTask.notify = function () {

console.log('notifying important people')

};

urgentTask.save =function(){

this.notify();

task.prototype.save.call(this);

};

urgentTask.save();

in the above example u can see that urgent task is inheriting task object and calling the save function of the task object by prototype.save.call

**Complex Decorator**

We will create a sub object to wrap an object to create more than one object

We will achieve it by doing a constructor

var task = function(name){

this.name = name;

this.completed = false;

}

task.prototype.completed = function (){

console.log("completed task by " + this.name )

this.completed =true;

}

task.prototype.save = function (){

console.log('saving task:' + this.name);

}

var task1 = new task("this is a test1");

task1.completed();

task1.save();

var urgentTask = function(name,priority){

task.call(this,name)

this.priority = priority;

}

urgentTask.prototype = Object.create(task.prototype);

// by creating it like this we will have accesse to the prototype object of the tasK

urgentTask.prototype.notify = function(){

console.log("noteify")

Task.priority.save.call()

}

urgentTask.prototype.save = function(){

console.log("lets save")

}

var ut = new urgentTask("this is urgent", 1);

console.log(ut)

/\*

output will be

this.name , this.completed , this.priority

A true sub object is created

Complete inheritance is achieved

\*/

By creating decorator pattern the main advantage is

The task object is exactly the same no changes is done to the task object

Only the inherited object is changed

**2**. Facade pattern

Used to provide simplified interface for a complicated system

Facade hides the chaos from us

Simplifies the interface, Wrap a complicated subsystem with a simpler interface

Example:

var Task = function (data) {

this.name = data.name ;

this.priority = data.priority;

this.project = data.project;

this.completed = data.completed;

}

var TaskService = function () {

Return {

complete: function(task){

Task.complete =true;

console.log('completed task :'+ task.name);

},

setCompletdDate : function(task){

task.completedDate = new Date();

},

notifyCompleted : function(task,user){

console.log("noteify user");

}

};

}

var mytask = new Task({

name : "task",

priority:1,

project: "test project",

completed: true

});

**// this below code can be changed**

taskService.complete(mytask);

if(mytask.completed == true)

{

taskService.setCompletdDate(mytask);

TaskService.notifyCompleted(mytask);

}

**// by creating a wrapper we can achive it**

var TaskServiceWrapper = function(){

var completeAndNotify =function (task){

taskService.complete(mytask);

if(mytask.completed == true)

{

taskService.setCompletdDate(mytask);

TaskService.notifyCompleted(mytask);

}

}

return {

completeAndNotify: completeAndNotify

}

}

TaskServiceWrapper.completeAndNotify(mytask);

1. **Behavioral Design pattern**

* Behavioral design pattern deals with the responsibilities of the objects
* Help objects to cooperate
* Assign clear hierarchy
* Can encapsulate requests
* Behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication

Fist let’s look at

1. **observer pattern**

Allows a collection of objects to watch an object and be notified of changes

**Benefits**:

* Allows for loosely coupled system
* One object is the focal point
* Group of objects watch for changes

Example :

Let’s consider a task object and three services called as logging, notification and auditing

In this scenario the logging, notification and auditing services are called as Observers it will watch the tasks object and watch for changes in the task object

The task object will become the subject. Will not be directly dealing with task will be decorating the task with subject.Which come with

ObserverList {} // list of observer

Notify() // a notiefy method

Then the interaction between them will be called as notification

The task object will be notifying the Observers whenever there is any change in the task object using call backs

For example : logging is going to supply a call back to ObserverList{} and when the Notify() is called it’s going to execute the call back that logging gave to us

**Things to take in to consideration**

* Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
* Model the independent functionality with a "subject" abstraction.
* Model the dependent functionality with an "observer" hierarchy.
* The Subject is coupled only to the Observer base class.
* The client configures the number and type of Observers.
* Observers register themselves with the Subject.
* The Subject broadcasts events to all registered Observers.
* The Subject may "push" information at the Observers, or, the Observers may "pull" the information they need from the Subject.

1. **Mediator pattern**

Controls communication between objects so neither objects has to be coupled to the others

If you look at the observer pattern u will see that observer didn’t know about subject but subject knew about observer

**Benefits:**

* Allows for loosely coupled system neither side has to know anything about the other side
* One object manages all other communication
* This allows for many to many relationship because several object can send notification to the event and several object can subscribe to the event all flows through one spot

**Things to take in to consideration**

* Identify a collection of interacting objects that would benefit from mutual decoupling.
* Encapsulate those interactions in the abstraction of a new class.
* Create an instance of that new class and rework all "peer" objects to interact with the Mediator only.
* Balance the principle of decoupling with the principle of distributing responsibility evenly.
* Be careful not to create a "controller" or "god" object.