[advertise here](http://buysellads.com/buy/detail/141671/zone/1275681?utm_source=site_141671&utm_medium=website&utm_campaign=cpmadhere&utm_content=zone_1275681)

**JavaScript Design Patterns: Observer**

It’s time to introduce you guys to the Observer pattern. If you’ve been following this blog lately, you may have already had an introduction through [my post](http://www.joezimjs.com/javascript/how-and-why-jz-publish-subscribe-should-be-used/) talking about my jQuery plugin called [JZ Publish/Subscribe](http://www.joezimjs.com/projects/publish-subscribe-jquery-plugin/). Well, here we’ll be talking about a few other ways to implement the Observer pattern, so you’ll know the method that works best for you and your application.



Before we get started I’d like to point you to the fact that this post is part of a series of posts about JavaScript Design Patterns. If you’d like to read any of the other posts in this series you can find the list of them at the [bottom of this post](http://www.joezimjs.com/javascript/javascript-design-patterns-observer/#series). It would probably be beneficial for any programmer that isn’t already well versed in JavaScript design patterns to go through these.

**What is the Observer Pattern**

The Observer pattern is a pretty simple concept. An observer (aka subscriber) subscribes to an observable object (aka publisher), waiting for something of interest to happen. Observers are also able to unsubscribe from the observer. At this point the behavior is dependent upon the way you have implemented the pattern. There are two basic methods for observers to get information on what is going on: push and pull. In the push method, whenever something happens, the observable immediately notifies the observers of the event. In the pull method, the observer will check with the observable to see if anything has changed whenever the observer feels it necessary to check.

I’m betting you want to see an example. Of course you do! You’re a programmer and to you code makes more sense than plain English, right? Let’s start off with an example of the push method:

1. var Observable = function() {
2. this.subscribers = [];
3. }
5. Observable.prototype = {
6. subscribe: function(callback) {
7. // In most situations, you would check to see if the
8. // callback already exists within the subscribers array,
9. // but for the sake of keeping us on track and because
10. // this isn't necessarily included, we'll leave it out.
11. // Just add the callback to the subscribers list
12. this.subscribers.push(callback);
13. },
14. unsubscribe: function(callback) {
15. var i = 0,
16. len = this.subscribers.length;
18. // Iterate through the array and if the callback is
19. // found, remove it.
20. for (; i < len; i++) {
21. if (this.subscribers[i] === callback) {
22. this.subscribers.splice(i, 1);
23. // Once we've found it, we don't need to
24. // continue, so just return.
25. return;
26. }
27. }
28. },
29. publish: function(data) {
30. var i = 0,
31. len = this.subscribers.length;
33. // Iterate over the subscribers array and call each of
34. // the callback functions.
35. for (; i < len; i++) {
36. this.subscribers[i](data);
37. }
38. }
39. };
41. var Observer = function (data) {
42. console.log(data);
43. }
45. // Here's where it gets used.
46. observable = new Observable();
47. observable.subscribe(Observer);
48. observable.publish('We published!');

There are a few things to talk about here. First off, all the functions related to the observer pattern are implemented within Observable. With JavaScript’s flexibility, you could also make the observer able to do the subscribing and unsubscribing, but I believe that it’s more sensible and understandable to implement all of that within the observable object. Another noteworthy point is that the observer is simply a function that can be used as a callback. In languages like Java, an observer would be an object that implements a specified interface. Then the entire object would be subscribed and the observable would simply call the method specified by the observer interface. Finally, in this example Observable is class that can be used by itself, though it is far more useful to be inherited by other objects so that they can become observable.

Now we’ll implement the pull method of the observer pattern. When you’re using the pull method, it makes more sense to swap things around a bit:

1. Observable = function() {
2. this.status = "constructed";
3. }
4. Observable.prototype.getStatus = function() {
5. return this.status;
6. }
8. Observer = function() {
9. this.subscriptions = [];
10. }
11. Observer.prototype = {
12. subscribeTo: function(observable) {
13. this.subscriptions.push(observable);
14. },
15. unsubscribeFrom: function(observable) {
16. var i = 0,
17. len = this.subscriptions.length;
19. // Iterate through the array and if the observable is
20. // found, remove it.
21. for (; i < len; i++) {
22. if (this.subscriptions[i] === observable) {
23. this.subscriptions.splice(i, 1);
24. // Once we've found it and removed it, we
25. // don't need to continue, so just return.
26. return;
27. }
28. }
29. }
30. doSomethingIfOk: function() {
31. var i = 0;
32. len = this.subscriptions.length;
34. // Iterate through the subscriptions and determine
35. // whether the status has changed to ok on each of them,
36. // and do something for each subscription that has
37. for (; i < len; i++) {
38. if (this.subscriptions[i].getStatus() === "ok") {
39. // Do something because the status of the
40. // observable is what we want it to be
41. }
42. }
43. }
44. }
46. var observer = new Observer(),
47. observable = new Observable();
48. observer.subscribeTo(observable);
50. // Nothing will happen because the status hasn't changed
51. observer.doSomethingIfOk();
53. // Change the status to "ok" so now something will happen
54. observable.status = "ok";
55. observer.doSomethingIfOk();

This is quite different from the push way of doing things, isn’t it? Now, whenever the observer feels like it should – or in this case, whenever I tell it to – it will check on the status of the observable(s) it is subscribed to. Normally this would be on a timer or something, but I decided to keep it simple and just call it manually. Once again Observable in this code shouldn’t technically be used by itself. Instead it should be subclassed so that there are built in mechanisms that change the status, rather than manually changing it like I did in the example.

**The Observer Patterns You’ve Already Seen**

The examples I’ve given are very simple examples and generally an observable object can have more than one type of event to be observed. Speaking of events, you may or may not have realized that event handling on DOM elements is an implementation of the observer pattern. The observer pattern is everywhere and is quite powerful and useful. Also, many jQuery plugins that use animations will include the observer pattern so you can inject your own functionality into different points of an animation.

**I Observe That The End Is Near**

The observer pattern is a marvelous tool for maintaining and organizing large action-based applications or even just to make your jQuery plugin more accessible and flexible. It adds a nice level of abstraction to help decouple your code and keep it clean and maintainable. Obviously the observer pattern shouldn’t be used for everything, but it can be quite useful in innumerable situations.

If you haven’t seen it yet, you should check out [JZ Publish/Subscribe](http://www.joezimjs.com/projects/publish-subscribe-jquery-plugin/) to see another way the observer pattern can be implemented. You can even see [how and why you should use it](http://www.joezimjs.com/javascript/how-and-why-jz-publish-subscribe-should-be-used/). If you’d rather move on from the observer pattern, you can check out the rest of the posts in the JavaScript Design Patterns series in the list below. Otherwise, I’d be delighted if you’d share this post with your friends or leave me a comment below. In any case, Happy Coding!

http://www.joezimjs.com/javascript/javascript-design-patterns-observer/