Extra Notes - Web Development - Patterns

- Dr Nick Hayward

Observer pattern

A brief overview of implementing the *observer* pattern using plain JavaScript.

contents

- intro
- pattern usage
- example

intro

The *observer* pattern is used to help define a *one to many* dependency between objects.

So, as the **subject** (object) changes state, any dependent **observers** (object/s) are then notified automatically and may update accordingly.

We're trying to ensure that those affected by such changes in state are kept in sync with the application itself. Instead of standard push/pull concepts between components, we're able to create bindings that become event driven.

So, we often simply use this pattern with bindings that are *one to many*, *one way*, and commonly event driven.

e.g.

```
                  -----------
                 |  subject  |
                  -----------
                    ^     |
                    |     |
         subscribe  |     |   notify
       unsubscribe  |     |
                    |     |
                    |     v
                  -----------
                 | observers |
                  -----------
```

In effect, the observer pattern creates a model of event subscription with notifications.

A benefit of this pattern is its tendency to promote *loose coupling* in component design and development.

This pattern is used a lot in JavaScript based applications, and client-side development in general. User events, for example, are a common example of this usage.

n.b. this pattern is often known by an alternative name, *Publication/Subscription* (or Pub/Sub). However, there are also subtle differences between these two patterns. Be careful with each implementation.

pattern usage

As noted above, the pattern includes two primary objects,

- **subject**
  - provides *interface* for observers to subscribe and unsubscribe

    - sends notifications to observers for changes in state
    - maintains record of subscribed observers
    - e.g. a *click* in the UI
- **observer**
    - includes a function to respond to *subject* notifications
    - e.g. a *handler* for the click

example

```
// constructor for subject
function Subject () {
  // keep track of observers
  this.observers = [];
}

// add subscribe to constructor prototype
Subject.prototype.subscribe = function(fn) {
  this.observers.push(fn);
};

// add unsubscribe to constructor prototype
Subject.prototype.unsubscribe = function(fn) {
  // ...
};

// add broadcast to constructor prototype
Subject.prototype.broadcast = function(status) {
  // each subscriber function called in response to state change...
  this.observers.forEach((subscriber) => subscriber(status));
};

// instantiate subject object
const domSubject = new Subject();

// subscribe & define function to call when broadcast message is sent
domSubject.subscribe((status) => {
  // check dom load
  let domCheck = status === true ? `dom loaded = ${status}` : `dom still loading...`;
  // log dom check
  console.log(domCheck)
});

document.addEventListener('DOMContentLoaded', () => domSubject.broadcast(true));
```