#JavaScript Design pattern

* **Module** helps to create private methods; just accessible by module.

->The Revealing Module pattern came about as Heilmann was frustrated with the fact that he had to repeat the name of the main object when we wanted to call one public method from another or access public variables. He also disliked the Module pattern’s requirement for having to switch to object literal notation for the things he wished to make public.

->we would simply define all of our functions and variables in the private scope and return an anonymous object with pointers to the private functionality we wished to reveal as public.

* The **Singleton pattern** is thus known because it restricts instantiation of a class to a single object.

Singletons differ from static classes (or objects) as we can delay their initialization, generally because they require some information that may not be available during initialization time.

* The **Observer** is a design pattern where an object (known as a subject) maintains a list of objects depending on it (observers), automatically notifying them of any changes to state.

Observer pattern is where we need to maintain consistency between related objects without making classes tightly coupled.

Dynamic relationships can exist between observers and subjects when using either pattern

* The **Mediator Pattern**: The dictionary refers to a mediator as a neutral party that assists in negotiations and conflict resolution. In our world, a mediator is a behavioral design pattern that allows us to expose a unified interface through which the different parts of a system may communicate.

The Mediator promotes loose coupling by ensuring that instead of components referring to each other explicitly, their interaction is handled through this central point.

This can help us decouple systems and improve the potential for component reusability.

A real-world analogy could be a typical airport traffic control system. A tower (Mediator) handles what planes can take off and land because all communications (notifications being listened out for or broadcast) are done from the planes to the control tower, rather than from plane-to-plane. A centralized controller is key to the success of this system and that's really the role a Mediator plays in software design.

* We can think of the **prototype pattern** as being based on prototypal inheritance where we create objects which act as prototypes for other objects

One of the benefits of using the prototype pattern is that we're working with the prototypal strengths JavaScript has to offer natively rather than attempting to imitate features of other languages.

 It can also come with a performance boost as well: when defining a function in an object, they're all created by reference (so all child objects point to the same function) instead of creating their own individual copies.

Object.create creates an object which has a specified prototype and optionally contains specified properties as well (e.g Object.create( prototype, optionalDescriptorObjects )).

* The general idea behind the **Command pattern** is that it provides us a means to **separate the** responsibilities of issuing commands from anything executing commands, delegating this responsibility to different objects instead.
* **Façade Pattern**: This pattern provides a convenient higher-level interface to a larger body of code, hiding its true underlying complexity. Think of it as simplifying the API being presented to other developers, something which almost always improves usability.

Whenever we use jQuery's $(el).css() or $(el).animate() methods, we're actually using a Facade - the simpler public interface that avoid us having to manually call the many internal methods in jQuery core required to get some behavior working. This also avoids the need to manually interact with DOM APIs and maintain state variables.

* **Factory Pattern** differs from the other patterns in its category is that it doesn't explicitly require us use a constructor. Instead, a Factory can provide a generic interface for creating objects, where we can specify the type of factory object we wish to be created.

This is particularly useful if the object creation process is relatively complex, e.g. if it strongly depends on dynamic factors or application configuration.

* **MVC** is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic and user-input.

**Models** manage the data for an application. They are concerned with neither the user-interface nor presentation layers but instead represent unique forms of data that an application may require. When a model changes (e.g when it is updated), it will typically notify its observers (e.g views, a concept we will cover shortly) that a change has occurred so that they may react accordingly.

**Views** are a visual representation of models that present a filtered view of their current state. JavaScript views are about building and maintaining a DOM element.

A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as "dumb" given that their knowledge of models and controllers in an application is limited.