Note: The motion graphics in this inventory may be reused in their respective stages

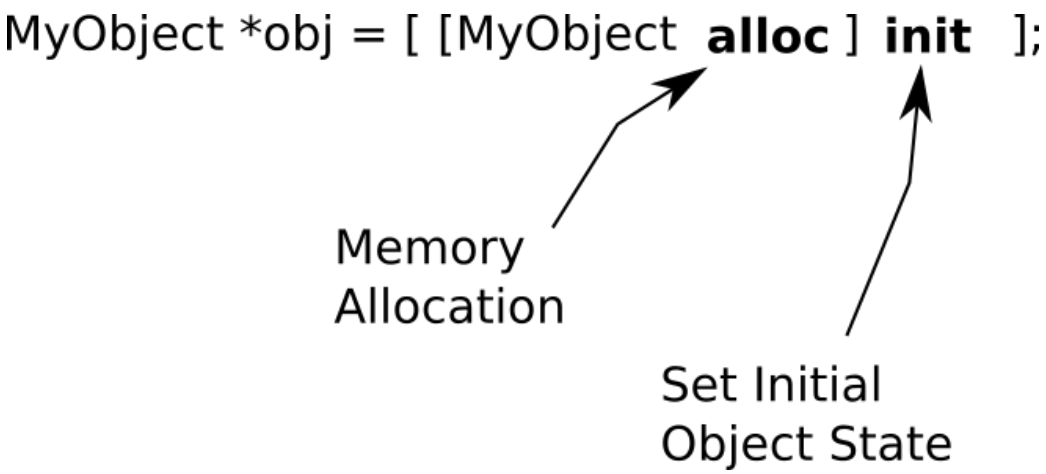
Stage 1: video 3

Now we'll go through a quick inventory of the 5 design patterns we'll be examining.

This is a lot of information, but don't worry, we'll revisit all five of these design patterns in-depth in later videos.

1. **Two-Stage Object Creation**

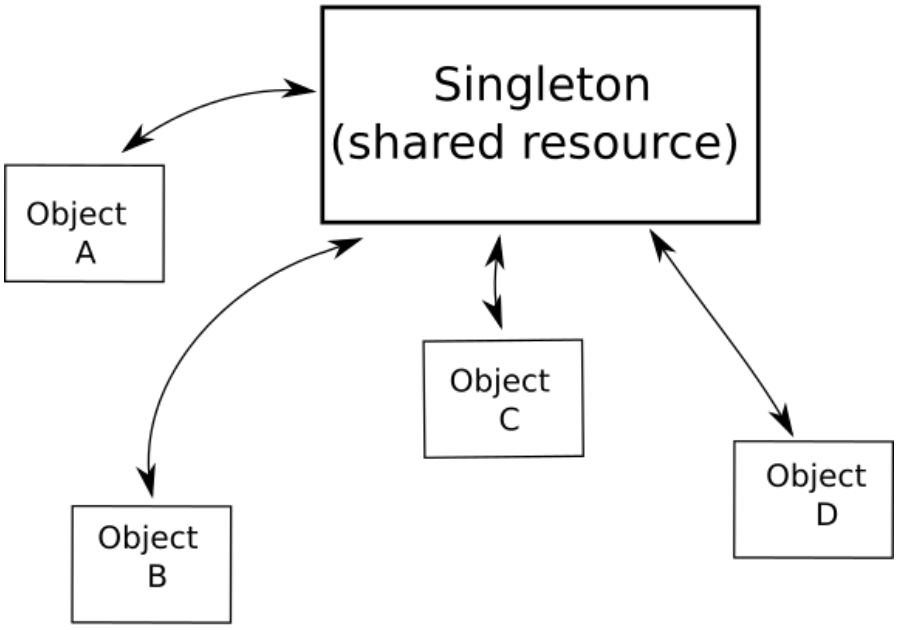
The first pattern we'll look at is two-stage object creation.
<MOTION>



Two-stage creation divides the process of object creation into separate memory allocation and initialization steps. This might appear cumbersome, but (as we'll see later) it allows greater flexibility in how we customize initialization methods. This facilitates code reuse both within classes and between classes in an inheritance hierarchy.
</MOTION>

2. **Singletons**

Then we'll look at singletons...
<MOTION>



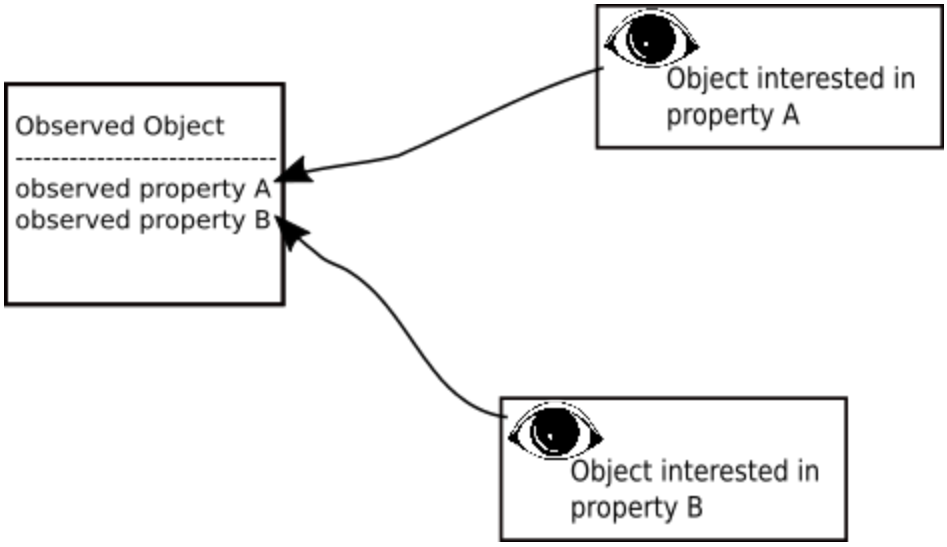
The Singleton pattern encapsulates a shared resource within a single, unique class instance. This instance arbitrates access to the resource and stores related state information. A class method provides the reference to this instance so there's no need to pass a reference around; any object that has access to the singleton class header can use the singleton.

</MOTION>

3. **KVC & KVO**

Then we'll take a look at Key-Value-Coding(or KVC) and Key-Value-Observing (a.k.a KVO)

<MOTION>



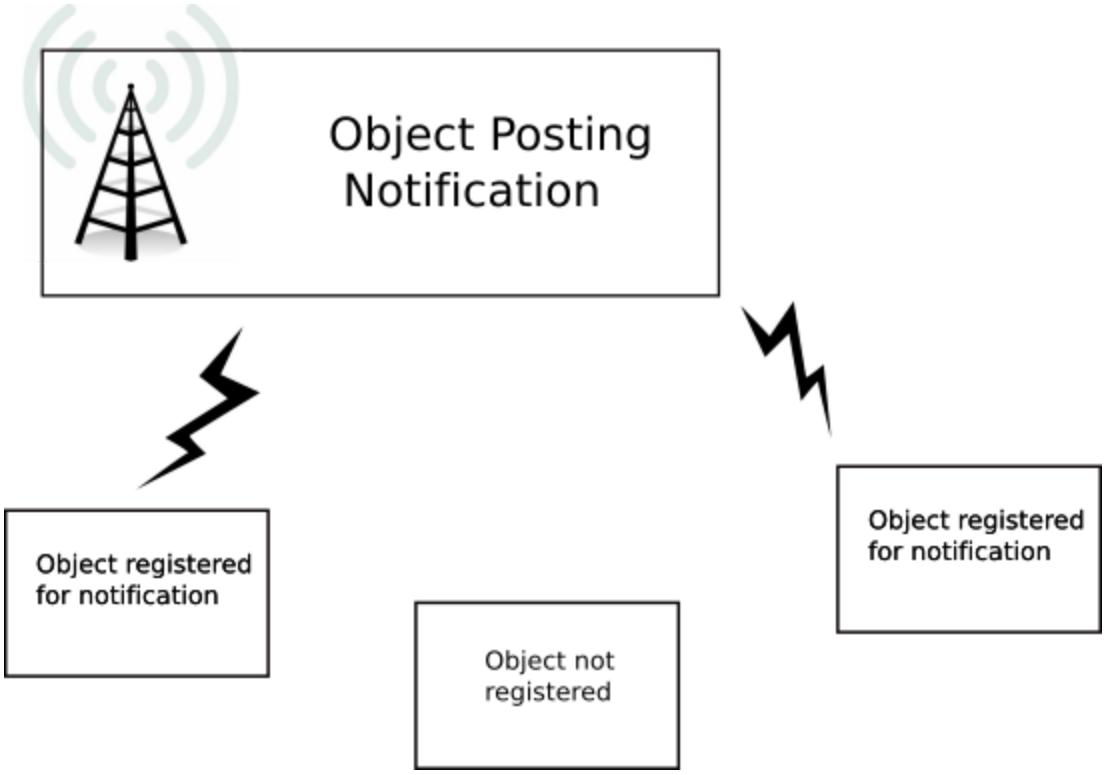
KVC is a universal dictionary interface that uses keys and key-paths for setting and accessing object properties. This interface opens the door to a number of powerful techniques, including KVO.

Key-value-Observing allows one object to use the key-coding interface to “put an eyeball” on a property of a second object. The observing object is alerted each time the property is modified. This allows the observer to respond to specific state conditions within other objects with minimal coupling. The observer only needs to know a simple key-path.

</MOTION>

4. **Notifications**

<MOTION>



NSNotifications work by having interested objects register for specific message types. When any instance posts a notification of that type, all the registered objects are notified. The system works something like a radio station broadcasting on a certain frequency. Any radios tuned to that frequency can listen in. Like a radio station, the posting object has no idea how many, if any, objects are listening and listeners don't know the source of the notification.

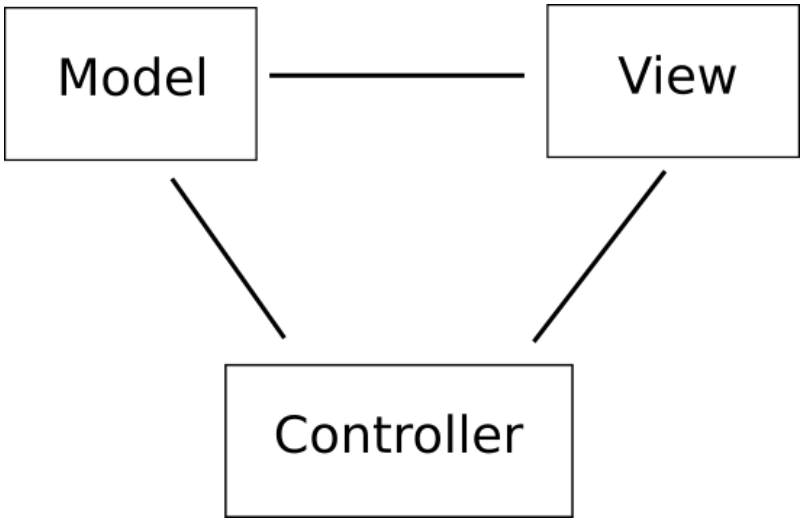
The notifications system allows for objects to pass signals between them with almost no coupling; objects posting or listening have no information whatsoever about other objects participating in the system.

Notifications are the extreme example of decoupled communication.
</MOTION>

5. **MVC**

Model-View-Controller, or MVC is the most abstract pattern we'll be examining. It structures the relationship between objects that provide data and those that present information in a user interface.

<MOTION>



As I mentioned in an earlier video, in Model-View-Controller, classes are members of either the Model (which manages application data), the View (which manages the interface) or the Controller (which acts as a mediator between the Model and the View). This separation into three distinct subsystems clarifies the roles and responsibilities of each, and allows classes to be more specialized.
</MOTION>

Those are the 5 design patterns we'll be examining.

In the next video, we'll look at two-step initialization in depth…

Questions:

1. The two parts of two-stage model creation are: (ans. allocation & initialization)
2. True or False: Decoupling is dangerous in software. (ans. false)
3. True or False: A singleton class can only be accessed by one object (ans. false)
4. MVC stands for? (ans: Model-View-Controller)
5. KVC stands for: (ans: Key-Value-Coding)
6. True or False: Objects need to register to receive notifications. (ans. true)