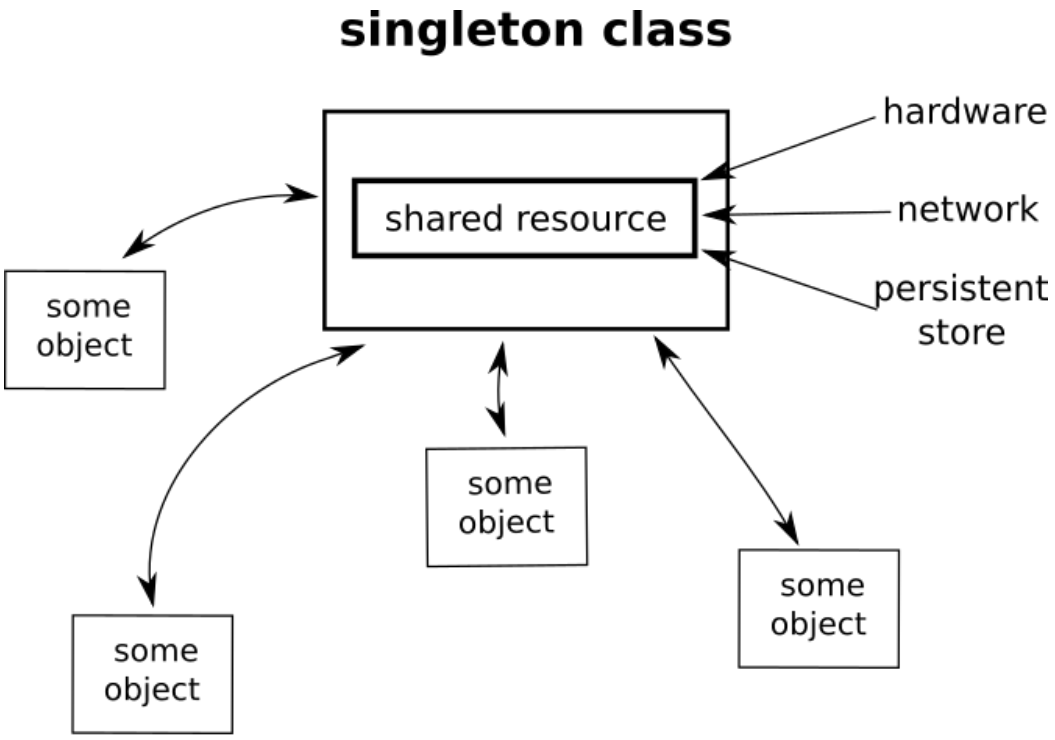
Stage 3.1 - Singletons

Scene 1: On-Set

Now we'll look at the Singleton pattern.

This design pattern defines the structure of a class that can only have one instance.
A singleton encapsulates a unique resource and makes it readily accessible throughout the application.

<MOTION>



Graphic shows objects arriving and interacting with the singleton class. The script will explain that the "shared resource" might be a hardware, network or persistent store resource.

The resource might be hardware, a network service, a persistence mechanism or anything else that can be modeled as a unique object or service.

One example from Cocoa-Touch is the physical device running an iOS application. For an executing app there is only one iPhone or iPad with one screen and one battery; UIDevice is a singleton class that provides an interface for detecting device orientation and querying battery charge. Having multiple UIDevice instances would be conceptually confusing since, in reality, there is only one underlying hardware device. In cases where the unique resource has a writeable configuration, this sort of discrepancy can lead to problems with race-conditions and deadlock.

Since they are unique, singletons act as arbiters, ensuring orderly access to the shared resource.

Singletons are often be modeled as a server within the application that accepts requests to send, store or retrieve data and configure resource state.

</MOTION>
Singletons can also promote decoupling by abstracting a service within an application.
For example, a simple data persistence singleton might provide *read* and *store* methods. Beneath this interface the persistence may be implemented as a file, then later as a database connection, and then as a cloud service. To client objects using the singleton, these implementation changes will go unnoticed.

Implementations of the pattern will typically make the singleton instance available through a special factory method: often called something like: “+sharedInstance”. Providing access to the instance through a class method eliminates the need to pass a reference. In any module that includes the singleton’s class declaration, the instance is available through the factory method.

Making the singleton available in this way also allows for lazy instantiation of the instance, since the allocation and initialization can be deferred until the first call to the factory method.

Now, we'll look at a singleton implementation.

Scene 2: Screencast

We'll use the "Command Line Tool" Xcode template, and create a project called "Singleton"

Our singleton class will be called SingleObject and it will be an NSObject subclass.
Opening the SingleObject header, we'll add a class method, +sharedInstance, since this is how the class will make the singleton available. Its return type will be SingleObject*, but id or instancetype would also work.

Now we'll go to the implementation file, and define our class method: +sharedInstance will store the instance in a *static local* reference called *ourSharedInstance.* Static locals are much like globals in that they retain their values for the lifetime of application, yet they are limited in scope. These qualities that make them ideal for a singleton, since they're permanent, ensure that our singleton is only available through +sharedInstance, which is one of the ways this singleton implementation will make sure the singleton stays singular.

The basic structure of +sharedInstance consists of a conditional block that tests if the singleton instance has been allocated. If so, then that instance is returned. This is the execution path that will be taken on any invocation after the first. If this is the first call to +sharedInstance, the singleton instance is allocated and initialized in the usual way using a nested alloc/init call.

In main() we'll add two calls to +sharedInstance along with log() statements that reference our variables, so XCode won't complain about them being "unused". We'll set breakpoints at these log statements, since they are the first lines after our sharedInstance calls return.

Running the application, we stop at the breakpoint after s1 is assigned. Using the command “po s1” we see that s1 is of SingleObject type. We’re also given the address of the s1 reference. If we continue, and break again after s2 is assigned. Using “po s2” we see that the s1 and s2 references are at identical memory addresses, indicating that they are the same object and that our singleton implementation is working correctly.

Since the two calls to +sharedInstance are executed sequentially on a single thread, our singleton class is guaranteed to work. However in a multithreaded environment, it is possible that two calls to +sharedInstance might occur almost simultaneously. If this happens with the first two calls to +sharedInstance, two threads of execution might enter the conditional block that tests for an existing instance of SingleObject. This would result in the creation of two instances of SingleObject, which would break our singleton and likely give rise to some horribly difficult to find bugs.

A solution to this problem is provided by the Grand Central Dispatch function dispatch_once(). Grand Central Dispatch (or GCD) is a library of Cocoa functions that provide concurrency support by offering a queue abstraction for threads of execution. Queues are conceptually much simpler than threads, making programming in a multithreaded environment more manageable

The function dispatch_once() method takes two arguments: a flag of of type *dispatch_once_t* and a block that can be entered only once by any thread of execution. This ensures that only one instance of SingleObject can ever be created. In the case where we have nearly simultaneous first calls to +sharedInstance, whichever thread reaches the dispatch_once() function first, will return the singleton, the other will return nil. +sharedInstance returning nil isn’t ideal, but it’s an obvious error that’s and easy to check for and address with retry logic, either within +sharedInstance, or the calling method. The important thing is we don’t have both a singleton and it’s doppleganger, being used throughout the application.

An additional change we could make, which will facilitate subclassing, is to change “[SingleObject alloc]” to “[[self class] alloc]”. This would allow a singleton subclass to use SingleObject’s +sharedInstance method while ensuring that the

correct alloc and init implementations are called. We might also change the return type of +sharedInstance to instancetype.

We'll run one more time just to be sure everything still works..

Questions

1. The method providing the shared single instance (e.g. sharedInstance) should be implemented as:
    a. An instance method
    b. A class method (correct answer)
    c. It doesn't need to be implemented
2. True or False: dispatch_once() is only really needed in a multithreaded environment (ans true)
3. What does GCD stand for? (ans. "grand central dispatch")
4. What is the singleton class in Cocoa Touch used to query the battery and screen orientation? (ans. UIDevice)
5. True or False: The class declaration(the header file) is sufficient for gaining access to a singleton instance? (ans. true)