Stage 3.3 - Refinements

Scene 1: On-Set

A few refinements can be added to our singleton to make it more robust.

One danger to our implementation is that an additional instance of the class might mistakenly be created, breaking the singleton pattern and the intended resource encapsulation.

A new programmer on a project might not be entirely familiar with our class or with singleton conventions. The danger is that, despite the availability of the +sharedInstance class method, a developer may simply call alloc and init instead.

We're going to "fool-proof" our implementation so that it becomes nearly impossible for a developer to accidently break our singleton.

Scene 2: Screencast

We'll continue working on our SingleObject in the same Singleton project.

The goal of our "fool-proofing" is to prevent the allocation of a SingleObject instance outside of the +sharedInstance method. We need a way to prevent a careless call to alloc/init from creating a singleton doppleganger.

Our solution is to override the *alloc* class method and locate the "real" allocation functionality in a "hidden' class method that we'll call +hiddenAlloc.

The new implementation of "alloc" will simply display a helpful log message and return nil. This should make it clear to anyone calling "alloc/init" that they should be using +sharedInstance.

Our +hiddenAlloc method will provide "alloc" functionality by simply calling +alloc in the superclass.

We change the call to [SingleObject alloc] to [SingleObject hiddenAlloc] in +sharedInstance.

Then we retest.

One downside to our "fool-proofing" is that it complicates potential subclassing since an instance of a child class can no longer call [super alloc] as we have in our -hiddenAlloc method. Also [[self class] hiddenAlloc] would fail if called by a child class instance unless the child class also implemented -hiddenAlloc.