OBJECTIVE-C COMPOSITE OBJECTS

We can create subclass within a class cluster that defines a class that embeds within it an object. These class objects are composite objects.

So you might be wondering what's a class cluster. So we will first see what's a class cluster.

Class Clusters

Class clusters are a design pattern that the Foundation framework makes extensive use of. Class clusters group a number of private concrete subclasses under a public abstract superclass. The grouping of classes in this way simplifies the publicly visible architecture of an object-oriented framework without reducing its functional richness. Class clusters are based on the Abstract Factory design pattern.

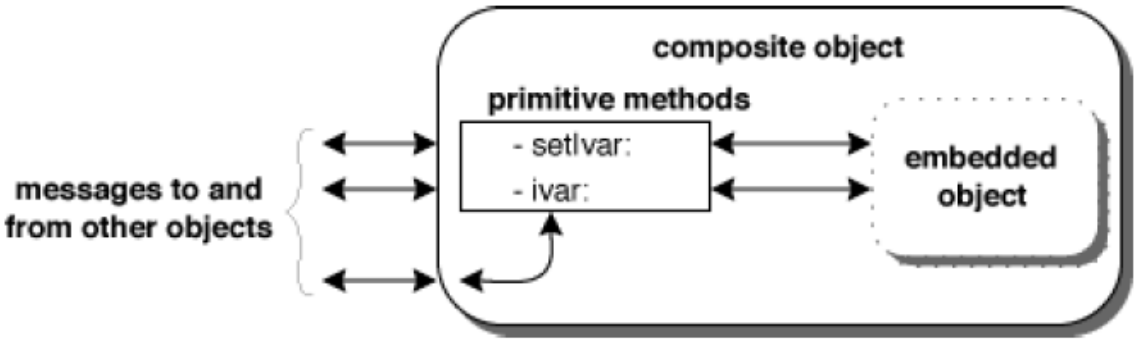
To make it simple, instead of creating multiple classes for similar functions, we create a single class that will take care of its handling based on the value of input.

For example, in NSNumber we have many clusters of classes like char, int, bool and so on. We group all of them to a single class that takes care of handling the similar operations in a single class. NSNumber actually wraps the value of these primitive types into objects.

So what's exactly composite object?

By embedding a private cluster object in an object of our own design, we create a composite object. This composite object can rely on the cluster object for its basic functionality, only intercepting messages that the composite object wants to handle in some particular way. This architecture reduces the amount of code we must write and lets you take advantage of the tested code provided by the Foundation Framework.

This is explained in the following figure.



Courtesy: Apple Documentation

The composite object must declare itself to be a subclass of the cluster's abstract superclass. As a subclass, it must override the superclass' primitive methods. It can also override derived methods, but this isn't necessary because the derived methods work through the primitive ones.

The count method of the NSArray class is an example; the intervening object's implementation of a method it overrides can be as simple as:

```
- (unsigned)count
{
return [embeddedObject count];
}
```

In the above example, embedded object is actually of type NSArray.

A Composite Object example

Now in order to see a complete example, let's look at the example from the Apple documentation which is given below.

```
#import <Foundation/Foundation.h>

@interface ValidatingArray : NSMutableArray
{
    NSMutableArray *embeddedArray;
}

+ validatingArray;
- init;
- (unsigned)count;
- objectAtIndex:(unsigned)index;
- (void)addObject:object;
- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
- (void)removeLastObject;
- (void)insertObject:object atIndex:(unsigned)index;
- (void)removeObjectAtIndex:(unsigned)index;

@end

@implementation ValidatingArray
- init
{
   self = [super init];
   if (self) {
      embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
   }
   return self;
}

+ validatingArray
{
   return [[self alloc] init] ;
}
- (unsigned)count
{
   return [embeddedArray count];
}
- objectAtIndex:(unsigned)index
{
    return [embeddedArray objectAtIndex:index];
}
- (void)addObject:(id)object
{
   if (object != nil) {
      [embeddedArray addObject:object];
   }
}
- (void)replaceObjectAtIndex:(unsigned)index withObject:(id)object;
{
   if (index <[embeddedArray count] && object != nil) {
       [embeddedArray replaceObjectAtIndex:index withObject:object];
   }
}
- (void)removeLastObject;
{
   if ([embeddedArray count] > 0) {
       [embeddedArray removeLastObject];
   }
}
- (void)insertObject:(id)object atIndex:(unsigned)index;
{
   if (object != nil) {
       [embeddedArray insertObject:object atIndex:index];
   }
}
- (void)removeObjectAtIndex:(unsigned)index;
```

```
{
   if (index <[embeddedArray count]) {
       [embeddedArray removeObjectAtIndex:index];
   }
}

@end

int main()
{
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   ValidatingArray *validatingArray = [ValidatingArray validatingArray];
   [validatingArray addObject:@"Object1"];
   [validatingArray addObject:@"Object2"];
   [validatingArray addObject:[NSNull null]];
   [validatingArray removeObjectAtIndex:2];
   NSString *aString = [validatingArray objectAtIndex:1];
   NSLog(@"The value at Index 1 is %@",aString);
   [pool drain];
   return 0;
}
```

Now when we compile and run the program, we will get the following result.

```
2013-09-28 22:03:54.294 demo[6247] The value at Index 1 is Object2
```

In the above example, we can see that validating array's one function would not allow adding null objects that will lead to crash in the normal scenario. But our validating array takes care of it. Similarly, each of the method in validating array adds validating processes apart from the normal sequence of operations.