                                                                       PROTOCOL V/S SUBCLASS

In the world of object-oriented programming, it’s important to be able to define a set of behavior that is expected of an object in a given situation.

As an example, a table view expects to be able to communicate with a data source object in order to find out what it is required to display.

This means that the data source must respond to a specific set of messages that the table view might send.

The data source could be an instance of any class, such as a view controller (a subclass of NSViewController on OS X or UIViewController on iOS) or a

 dedicated data source class that perhaps just inherits from NSObject. In order for the table view to know whether an object is suitable as a data source,

it’s important to be able to declare that the object implements the necessary methods.

Objective-C allows you to define protocols, which declare the methods expected to be used for a particular situation. This chapter describes the syntax to

define a formal protocol, and explains how to mark a class interface as conforming to a protocol, which means that the class must implement the required methods.

A class interface declares the methods and properties associated with that class. A protocol, by contrast, is used to declare methods and properties

that are independent of any specific class.

The basic syntax to define a protocol looks like this:

@protocol ProtocolName

// list of methods and properties

@end

The syntax to indicate that a class adopts a protocol again uses angle brackets, like this

@interface MyClass : NSObject <MyProtocol>

...

@end

Protocols Are Used for Anonymity

\* Protocols are also useful in situations where the class of an object isn’t known, or needs to stay hidden.

  As an example, the developer of a framework may choose not to publish the interface for one of the classes within the framework.

  Because the class name isn’t known, it’s not possible for a user of the framework to create an instance of that class directly.

#Subclass-

Every object you create in your Cocoa application descends from the 'NSObject' foundation class. The NSObject class identifies properties and

methods which apply to all objects. The NSObject class is divided into smaller groups of objects, called subclasses.

Objects in these subclasses not only conform to the protocol of NSObject, they are also defined more precisely by the methods that govern their subclass.

Every object class inherits from the superclasses above it in the object hierarchy, and also declares the methods which make it a unique class.

\*Why might I need or want to subclass?

It is common for Applescript developers to look for alternate methods of achieving goals and performing tasks. We often rely on complex shell scripts,

unreliable brute-force parsing, or klunky gui scripts to accomplish simple tasks. In plain applescript we do not have the luxury of accessing

Objective-C classes directly. But in ASStudio, we have a comprehensive set of powerful tools available to us in our ability to

incorporate Objective-C calls directly into our applescripts. Many of our problems in ASStudio can be easily solved by creating an Objective-C subclass

that can both handle our task, and also avoid messy workarounds and hacks.

There are three primary reasons that your application may have to use subclasses...

1) ASStudio does not support all of the capabilities available to developers using other languages like Objective-C or java. Unfortunately,

   the dictionaries that govern which objects and methods you have access to in ASStudio are not complete... leaving gaps in feature support

   that range from trivial to severe. To gain access to these features, you must write custom subclasses that add the unsupported code you need.

2) You may want to override the methods in existing classes. Some objects have default methods and settings that may not be suitable for your needs.

   By creating a custom subclass, you can set your interface elements to inherit qualities according to your criteria, not just from the object's default

   class implementation.

3) It is often beneficial to create your own generic subclasses, to act as a convenient location to keep any methods you write that execute

   custom Objective-C code. In many cases, Objective-C has methods of performing certain actions or managing data in ways that Applescript is not

   capable of. Objective-C has a versatile set of existing methods that you can use to make your application work more efficiently and effectively.