Object Oriented Programming in Script

Author: Jeff Stephenson
Date: 4 April 1988

SIERRA CONFIDENTIAL

Table of Contents

Object Oriented Programming 3

Object Oriented Terminology 6

Classes 8
    Defining Classes 8
    Compiling Classes 9

Objects 11

Sending Messages 12

An Extended Example 13

Object Oriented Programming

Object oriented programming (OOP) is a relatively new paradigm in programming. The Smalltalk language, developed at Xerox PARC, was the first language to fully use the concept of OOP. Since then, many languages (such as C++, Scheme, Objective C, and Actor) have incorporated the concepts to different degrees. Because of the power of OOP for things such as simulations (and after all, what are adventure games?), Script was designed with the intention that much of the programming would use this concept. Those of us using the language over the last several months have found ourselves moving more and more away from its procedural aspects and toward the OOP aspects -- rooms, dialogs, menus, everything is now an object of one sort or another.

OOP languages basically turn the data/procedure relationship in conventional programming inside out: instead of sending data to a procedure for the procedure to act upon, we send a message to data (an object) telling it what to do. This allows the sender of the message to obtain a service from the receiver object with no knowledge of how this service is performed, something which is not possible in procedural languages.

As an example, consider displaying the 'value' of data elements in a program in a procedural language and then an OOP language. In this example, we'll assume that the procedural language is untyped (as opposed to the current direction of strong typing in C, Ada, Pascal, etc.). Strong typing would make the follow exercise even harder. To make the example more concrete, we'll imagine that we want to be able to display one of two data element types: a list of strings or an array of integers.

Procedurally, we'll use a C-like language (with lots of extensions which don't exist in C) and write

```
Show(data)
/* Here's one place C would have a problem -- data could either be
 * a list of strings or an array of integers.  What type does one
 * declare?
 */
{
     switch (typeof(data)) {       /* how is typeof() implemented? */
          case stringList:
               for (str = ListStart(data) ;
                    str != NULL ;
                    str = ListNext(data)
                   )
                         printf(str);
               break;
          case integerArray:
               for (i = 0 ; i < sizeof(data) ; ++i)
                    printf("%d", data[i]);
               break;
          }
}
```

Note that one could not easily write this in C because of strong data typing and the difficulty of implementing both typeof() and this particular use of sizeof(). What usually ends up happening in a procedural language is that

instead of one general-purpose Show() procedure, one writes a lot of special purpose ShowStringList() and ShowIntegerArray() procedures and whoever wants to display something has to be aware of the type of data being displayed and call the appropriate procedure.

Contrast this with object-oriented programming. Either type of data can be displayed by simply sending the data object the show message:

```
(data show:)
```

Since both Arrays and Lists are sub-types of the Collection type they implement this in the same way (they actually share the code):

```
(method (show)
     (self eachElementDo: #show:)
)
```

Both the list and the array simple send each of their elements the message requesting the element to display itself. The elements, strings and integers, implement the show method in different manners. The show method for a string might be

```
(method (show)
     (Print str)
)
```

whereas that for the integer might be

```
(method (show)
     (Print "%d" number)
)
```

Note how much simpler the OOP code is than the procedural code. This is because the responsibility for displaying something is delegated to the object being displayed, freeing the calling code from knowing anything about it. In fact, because of this passing on of responsibility, the above code will also handle lists which contain elements of several data types -- (data show:) would also work if data were a list which contained a string, an integer, a list of integers, an array of strings, and a list of arrays of lists of strings. No change in code is necessary.

Another benefit of the OOP code is that adding a new data type is as easy as writing the show method for the data type. In the procedural code, the Show() procedure would need re-writing, as would the typeof() and sizeof() procedures -- the addition of the new type propagates throughout the code rather than remaining isolated in the new type.

Rather than continuing to give examples in an unspecified language, we will first define the terms to be used in discussions of OOP languages and then give the syntax and commands which Script uses. Following that will be more examples of the use of OOP in adventure game coding.

Object Oriented Terminology

Property:
Properties are data associated with an object. A door object might have such properties as the room number into which it opens, its state (open or closed), and its position on the screen.

Method:
A method is a procedure which is internal to an object or class. A door object might have a method open which causes the door to change state from closed to open.

Message:
Methods of an object are invoked by sending the object messages. In Script a message is sent to an object by enclosing the object name, then the message, in parenthesis (much like a procedure call):

```
(doorToCloset open:)
```

sends the open: message to the object doorToCloset.

Selector:
Selectors are used in messages to indicate the method which is to be invoked. The symbol open: in the above example of messaging is the selector for the open method.

Object:
An object is a collection of properties and methods bundled together, and should be considered to be a distinct entity (hence the use of the word 'object').

Class:
A class is a 'generic' object of a certain type. It has no real existence of its own, but provides the default methods and property values to those objects which are instances (see below) of it. Classes may have sub-classes, which are more specific types of a class (for example, the class SlidingDoor is a sub-class of the class Door) and super-classes, which are generalizations of the class (the class Opening might be a super-class of the class Door).

Instance:
An instance of a class is a 'concrete' object which has the properties and methods of the class, but whose properties are distinctly its own. For example the object doorToCloset is an instance of the class Door which is different from the object doorToPatio, another instance of class Door. Changing the property values of an instance (object) of a class will not affect the property values of other instances of the class.

Inheritance:
Inheritance is one of the most important concepts in OOP, and is responsible for much of its power. A sub-class inherits all the methods and properties of its super-class, then goes on to add its own distinct properties and methods, or to modify an inherited method from its super-class. We might define the class ElevatorDoor as a sub-class of the class Door. It would thus inherit the properties of Door but would add a property to tell whether the light associated with the

door is on or off. It would also inherit Door's methods, but would modify the open method to not only change the state of the ElevatorDoor to open, but also to change the state of the light property to on. All other methods work just the same as for the Door class.

Self:

Often, an object needs to invoke one of its own methods, without really knowing who it is (it may be executing code inherited from its super-class). This is done by sending a message to the object self, which is always the object which invoked the current method.

Super:

An important factor in using inheritance to define new methods is being able to get access to the super-class's code for a method within the newly defined method. The class super is provided for this purpose -- sending a message to class super invokes the method within the super-class of self, rather than the method within self. Thus, the open method for the class ElevatorDoor might contain the code

```
(super open:)
(= lightState on)
```

which calls upon the open method of the class Door to do its stuff, then sets the light state to on.

A Note on Naming

There are a number of conventions which are normally (but not universally) followed in naming things in object oriented programming. Properties, being data, usually have names which are nouns; methods, being actions on data, usually have names which are verbs; classes and instances, which may represent either concrete objects or actions, may have either nouns or verbs as names.

The first letter of property, method, and instance names is lowercase, whereas the first letter of class name is uppercase. In both types of names, succeeding words in the names are set off in the name by capitalizing the first letter of the word, e.g. AutomaticDoor or elevatorDoor.

Classes

Defining Classes

The first step in any object-oriented programming project is to define the classes which will be used in the project. Generally there is already a large library of classes to draw on (in this case the classes documented in Script Classes for Adventure Games), and often these classes are all one will need in the project. However, in order to define new classes and to understand the old ones, it is necessary to understand the class statement for defining a class.

The form of the class statement is:

```
(class aClass kindof superClass
     (properties
          aProperty value
          ...
     )

     (methods
          aMethod
          ...
     )

     (method (aMethod [p1 p2 ...] [&tmp t1 t2 ...])
          code
     )
     ...

     [(procedure ...)]
)
```

This statement defines aClass as a sub-class (kindof) superClass. This means that it inherits all of superClass' properties and methods, and will either modify or add to them. In order for the sc compiler to compile this class definition, superClass must either have been defined earlier in the current source file or in another file which will be compiled before this file will be compiled, adding the super-class definition to the file classdef (more on this at the end of this section).

The properties section of the class statement is where new properties are introduced and old default property values redefined. All entries in this section are of the form aProperty value where aProperty is a symbol and value is a constant expression. If aProperty is not a property of superClass, it is added as a new property in the class being defined and its default value is value. If the property exists in superClass, its default value for the class being defined is just changed to value.

The methods section of the class statement lists any symbols which will be the names of methods added to superClass to create the class being defined. Listing a method which is already defined in superClass does nothing, and a method name need not be listed in the methods section if it is only being redefined.

The method statement in the class statement is almost identical to the

procedure statement described in The Script Programming Language. It differs in that the properties of the class in which the method is defined can be accessed as if they were variables, i.e. by simply using their names. Also, code in a method definition is the only place in which sends to the objects self and super are valid. Only methods inherited from the super-class or listed in the methods section may be defined. As in procedures, the compiler-defined variable argc gives the number of parameters passed to the method.

Note that procedures may also be included in a class statement. These procedures can be used by the methods defined within the class, but cannot be accessed from outside the class definition.

Compiling Classes

Source files containing class definitions are compiled with the sc compiler just like any other source file. There are some subtleties involved where class definitions are concerned, however, arising from the fact that all sub-classes of a class must be defined after the class' definition. Thus the order of source file compilation is very important -- the files containing the lowest level classes (such as Object) must be compiled first, followed by successive sub-classes. An understanding of how the compiler maintains information on all the classes will clarify how the class compilation process works.

Two files, classdef and selector constitute the database in which the compiler tracks the classes being defined. The first thing that the compiler does when compiling a file is to read in the file selector, which defines the symbols which are known to be selectors and the selector numbers corresponding to them. From this it also obtains the largest selector number currently defined, which will mark the point from which it will assign new selector numbers when they are needed. If you are compiling on a network, this file will be locked when you start compiling. This prevents anyone else from starting a compile (and possibly changing the information in the database) while you are using it.

The compiler next reads the file classdef (if it exists), from which it obtains information about the classes which have already been compiled. The information in classdef includes the class number (assigned by the compiler), the number of the script in which it is defined, the class number of the class' super-class, the names of the class' methods, and the names of the class' properties with their default values.

The compiler now gets around to reading your source code. As the compiler encounters each class definition, it records in its class symbol table the characteristics of the class. If the class already has an entry in the symbol table because it has been previously defined, that entry is cleared and the class is redefined using the current source code. This guarantees that the compiler is always using the most current class definition.

The class definition is built from the information contained in the methods and properties sections -- the selectors encountered here are added to those inherited from the class' super-class to arrive at the structure of the class being defined. Any symbol encountered in a properties or methods section of a class definition is looked up in a special symbol table for selectors. If it is not present there, it is entered in the selector table

and assigned the next available selector number. Symbols are also entered in the selector symbol table if they are undefined symbols encountered in the position which a selector would occupy in a message to an object.

Assuming that the source file compiles with no errors, the compiler rewrites classdef and selector from its class and selector symbol tables. On multi-files compilations this is done after each successful compilation. The external database is thus kept as up-to-date as possible. Along with these files, the compiler rewrites classtbl and vocab.001 after each successful compilation. Classtbl is a table which gives the script number in which each class is defined, and is used by the kernel to load the appropriate script when a class is referenced. Vocab.001 is a file which contains the names of all the selectors in a format which can be used by the kernel to display selector names in error and debugging messages.

At the start of a project, you start with no classdef file and a selector file which is a copy of the file selector.new. The latter file contains the definitions of those selectors which the kernel needs to know about and which thus must have particular numbers. As you compile files, beginning with system.sc, the files classdef and selector grow with the addition of new classes and selectors. This growth is a one-way street, though -- nothing you do to your source code will cause a class or selector to be removed from these files. Thus, by the end of the project you are likely to have a number of obsolete classes and selectors hanging around in the database. To winnow the chaff from these files you should, near the end of the project, delete classdef, copy selector.new over your existing selector, and recompile all your files (in the proper order, of course!). This leaves you with a class/selector database with only those classes and selectors which you are using.

Objects

Objects are specific instances of a class, and are defined using the instance statement:

```
(instance anObject of aClass
     (properties
          aProperty: value
          ...
     )

     (method (aMethod [p1 p2 ...] [&tmp t1 t2 ...])
          code
     )
     ...

     [(procedure ...)]
)
```

This defines anObject as an instance of class aClass. The properties and method statements are optional and are used to over-ride the default values and methods inherited from aClass.

The ID of an object or a class is what tells the sci kernel where to send messages. The object ID is obtained by simply writing the object's name wherever an expression is valid -- it can be assigned to a variable or passed as a parameter to a procedure. Thus, if we define egoObj as an instance of class Ego,

```
(instance egoObj of Ego)
```

we can assign the ID of egoObj to the global variable ego:

```
(global   ego       0)
(= ego egoObj)
```

Once this has been done, the following two expressions are equivalent:

```
(ego x?)
(egoObj x?)
```

To find the distance from the object ego to the object wolf, we can pass the wolf's ID as an argument to ego's distanceTo: method:

```
(ego distanceTo: wolf)
```

Any unknown symbol encountered in a compilation is assumed to be the ID of some object which is to be defined later in the source file. If no object with the symbol as its name is encountered by the end of the file, an error will be raised.

Sending Messages

The syntax for sending a message to an object is identical to that for a procedure call. The object name (or an expression which evaluates to an object ID) is followed by the message selector and any parameters, all enclosed in parentheses. There are three different kinds of messages which can be sent to objects:

Setting a property:

A property of an object can be set by sending a message whose message selector is the name of the property followed by a colon (':') followed by the new value of the property:

```
(ego x:23) or (ego x: 23)
```

sets the x property of ego to 23.

Requesting the value of a property:

The value of a property can be obtained by sending a message with no parameters whose message selector is the name of the property followed by a question mark ('?'):

```
(ego x?)
```

will return the value of the x property of ego.

Invoking a method:

A method of an object can be invoked by sending a message with any number of parameters whose message selector is the name of the method followed by a colon (':'):

```
(ego moveTo: x y) or (ego moveTo:x y)
```

tells ego to move to coordinates x and y by invoking the moveTo method of ego.

Any number of messages can be sent in one fell swoop:

```
(ego
     x:50
     y:50
     setMotion: MoveTo 100 100
     setCycle: Reverse self
)
```

will position ego at coordinates (50, 50), start him moving to coordinates (100, 100), and set him to cycle in reverse cel order. When multiple messages are sent to an object, the messager in the kernel sends them one at a time in left to right order. In multiple message sends, all parameters are evaluated before the messages are sent.

An Extended Example

The only way to really understand OOP is not to read about it, but to dive into an example and get a feel for how it is used (or better yet to actually write code!). The following extended example goes through the development of a new class, the AutomaticDoor, for adventure games. Before continuing, it is advisable to read through Script Classes for Adventure Games in order to become familiar with the classes upon which the AutomaticDoor is built.

The AutomaticDoor concept was inspired by Space Quest, in which there are lots of doors which open whenever ego gets near them and close when he moves away. We would like the doors to do this by themselves, like good automatic doors, rather than having to explicitly write code in each room which checks ego's position, remembers whether the door is open or closed, opens or closes the door, etc.

The first step in defining a class is conceptual -- determining what the class represents and thus what properties and methods it should have in order to carry out its role in the scheme of things.

We'll say that a door is a subclass of the Actor class, since it will be an object visible on the screen. A door goes somewhere, so it should have an entranceTo property which tells us which room is on the other side of the door. The door may or may not be locked, so we'll need a locked property to keep track of this, along with a key property which is the ID of a key object which locks or unlocks the door. Doors generally make some sort of noise when opening and closing, so we'll add openSnd and closeSnd as properties to tell us what sound the door makes. Then there is the question of keeping track of whether the door is opening, open, closing, or closed. This state will be kept in the property doorState. Finally, since this is an automatic door, it will need some way of telling when an Actor is near enough to cause it to open. This will be dealt with by an object of class Code, whose ID will be kept in the actorNearBy property.

This gives us the beginnings of the class statement:

```
(class AutomaticDoor kindof Actor
     (properties
          entranceTo 0
          locked FALSE
          key 0
          openSnd 0
          closeSnd 0
          doorState 0
          actorNearBy 0
     )
)
```

We'll need symbolic definitions for the state of the door:

```
(enum
     doorOpen
     doorOpening
     doorClosed
     doorClosing
```

```
)
```

To this we now need to add the methods section. The methods are the things we wish to have the door do. Thus, we will want methods open and close, as well as lock and unlock:

```
(methods
     open                ;open the door
     close               ;close the door
     lock                ;lock the door
     unlock              ;unlock the door
)
```

We'll start with the init: method (inherited from Prop), which adds the door to a room when we first enter the room. This should handle having the door be open if it is the door to the room which we have come from and closed otherwise.

```
(method (init &tmp doorState)
     (= doorState
          (if (== prevRoomNum entranceTo)
               ;We just came from the room to which this door
               ;is an entrance -- the door should be open.
               doorOpen
          else
               ;We didn't come through this door -- have
               ;it closed.
               doorClosed
     )

     ;Set the cel based on whether the door is open or closed.  This
     ;assumes that cel 0 is the cel with the door entirely closed.
     (= cel
          (if (== doorState closed)
               0
          else
               (- (NumCels view loop) 1)
          )
     )

     ;Pass the initialization along to the super-class to add the
     ;door to the cast, etc.
     (super init:)

     ;Stop updating the door, to reduce the burden on the animation
     ;system.
     (self stopUpd:)
)
```

Note that we have not added the actorNearBy code at this point -- that will be specific to each door.

We now want methods to open and close doors. They should know enough not to try opening a locked door or one which is either already opened or opening.

Also, if a door-opening sound has been defined for the door (by putting a the object ID of a Sound in openSnd), that sound should be played as the door opens. The same goes for the door closing.

```
(method (open)
     (if  (and (! locked)
               (!= doorState doorOpening)
               (!= doorState doorOpen)
          )

          ;If the door is not opened or opening, start it doing so by
          ;having it cycle to the end of its loop.  When it is done,
          ;the cycle instance will cue us.
          (= doorState opening)
          (self setCycle: EndLoop self:)
          (if openSnd
               (openSnd doit:)
          )
     )
)

(method (close)
     ;We don't have to worry about the door being locked,
     ;since in that case it wouldn't be open.
     (if  (and
               (!= doorState closing)
               (!= doorState closed)
          )
          (= doorState closing)
          (self setCycle: BegLoop self)
          (if closeSnd
               (closeSnd doit:)
          )
     )
)
```

When either the EndLoop or BegLoop cycle type initiated by open or close is done, the cycle class will cue the door. The cue method must thus handle the change from doorOpening to doorOpen and from doorClosing to doorClosed:

```
(method (cue)
     (= doorState
          (if (== doorState doorOpening) doorOpen else doorClosed)
     )
     (self stopUpd:)
)
```

Note that we stop updating the door when it is no longer cycling in order to reduce the load on the animation system.

Locking and unlocking the door may be done by anyone who has the key to it. Though not in the class system yet, each Actor will have a has: method which will test to see if the Actor has a certain inventory object. We use this to define the lock and unlock methods, which take the Actor who is trying to do the action as a parameter. Note the use of a common procedure to exploit the similarities in code:

```
(method (lock who)
     (DoLock who TRUE)
)

(method (unlock who)
     (DoLock who FALSE)
)

(procedure (DoLock who newLockState)
     (if (who has: key)
          (= locked newLockState)
     else
          (Print "You don't have the proper key!")
     )
)
```

Remember that the property key has the ID of the object which is the key to this door.

Since the init: method of the door has added the door to the cast, the door will be sent the doit: message during each animation cycle. This is the ideal place to hook in the check to see if the door should be opened or closed. The door will check to see if any Actor is near enough (as defined by a TRUE return from the code whose ID is in actorNearBy), and if so will invoke the open: method. Otherwise, it will invoke the close: method. Note that since these methods already check to see if the given operation is in progress or is completed, we can invoke them blindly without creating any problems.

```
(method (doit)
     ;If there is no test for a nearby actor, don't try to test.
     (if (== actorNearBy 0) (return))

     ;See if anyone is near.
     (if (cast firstTrue: #perform: actorNearBy)
          (self open:)
     else
          (self close:)
     )
)
```

The test in this method works in the following way: we tell each member of the cast to perform: the code whose ID is in actorNearBy. If any member of the cast returns TRUE from this code, the firstTrue: method will end and return the ID of that member. If no member returns TRUE, firstTrue: will return NULL. Thus the conditional statement will be TRUE if any element of the cast returns TRUE to the code in actorNearBy. The general structure for this code (each door will have its own specific test for nearness) is:

```
(instance nearByTest of Code
     (method (doit theObj)
          (return
               code to test the nearness of the Actor
          )
     )
)
```

Now that the methods have been defined for the AutomaticDoor class, we can use it to define doors in any room in the game. Say we're in a room which has two doors (like the starting room of Space Quest). The first is a face-on door to a closet, which we'll call closetDoor. Since it's face-on and there are no obstructions, we'll use a simple position check to see whether ego is near it:

```
(instance closetDoor of AutomaticDoor
     ;Set the initial position of the object and
     ;say that it opens into the closet.
     (properties
          x:100
          y:60
          view:vClosetDoor
          entranceTo:closet
          actorNearBy:nearCloset
     )
)

(instance nearCloset of Code
     (method (doit theObj)
          (return
               (and
                    (< x (theObj x?))
                    (> (+ x (CelWide view loop cel)) (theObj x?))
                    (< (abs (- y (theObj y?))) 10)
               )
          )
     )
)
```

The code in nearCloset checks to see if the object's x position is within the bounds of the door and whether it is within 10 pixels of the door vertically. If all the above are true, it returns TRUE.

The second door will be an entrance to a secret room. Let's say that the room is so messy near the door that the only easy way to tell if ego is near it is to draw some control into the picture and see if ego is on the control. Also, since the room to which this door leads is secret, we'll need the cardKey object to unlock the door.

```
(instance secretDoor of AutomaticDoor
     (properties
          x:10
          y:80
          entranceTo:secretRoom
          locked:TRUE
          key:cardKey
          actorNearBy:nearSecretDoor
     )
)

(instance nearSecretDoor of Code
     (method (doit theObj)
          (return
               (OnControl theObj secretControl)
```

```
            )
        )
```

In the room code, we add the doors during the initialization phase:

```
    (method (init)
        ...
        (closetDoor init:)
        (secretDoor init:)
        ...
    )


    (if (Said 'lock / door')
        (cond
            ((closetDoor actorNearBy: ego)
                (closetDoor lock:)
            )
            ((secretDoor actorNearBy: ego)
                (secretDoor lock:)
            )
            (else
                (Print "You're not near a door!")
            )
        )
    )


    (if (Said 'unlock / door')
        (cond
            ((closetDoor actorNearBy:)
                (closetDoor unlock:)
            )
            ((secretDoor actorNearBy:)
                (secretDoor unlock:)
            )
            (else
                (Print "You're not near a door!")
            )
        )
    )
```

We use the door's own actorNearBy check to see if ego is close enough to open the door.

Thus concludes our excursion into Object Oriented Programming. The idea behind this style of programming is to create abstractions of the things you are modeling, decide how all these classes are related, and set up a hierarchy of super- and sub-classes which encapsulates these relationships. In the classes are hidden the methods which do things to objects which are instances of the classes, and the properties, which are the defining characteristics of the class. Objects are then instances of a given class which have particular values for the properties and may even have different methods for implementing a certain concept.

What all this setup gives you is the ability in your code to say

```
(secretDoor unlock:)
```

to unlock a door, rather than having to write the code in-line or writing an unlock routine which needs to handle all the possible cases of doors which occur in the game.

Enjoy.

Index

class 6
class statement 8
classdef 9
inheritance 7
instance 6, 11
kindof 8
message 6, 12
messager 12
method 6, 8, 9, 11
    invoking 12
methods 8, 9
naming 7
object 6
procedure 8, 11
properties 8, 11
property 6
    requesting 12
    setting 12
selector 6, 9
self 7
sub-class 6
super 7
super-class 6