Originally published in *PIE Developers* magazine, January 1994. For information about PIE Developers, contact Creative Digital Systems at CDS.SEM@applelink.apple.com or 415-621-4252.

Class-based NewtonScript Programming

Walter Smith
Apple Computer, Inc.
wrs@apple.com

NewtonScript is often described as an "object-oriented" language. However, even if (or *especially* if) you have some experience with other object-oriented languages, such as Smalltalk or C++, you may be a bit confused by it. NewtonScript does have many features that will be familiar to you, but it has one important difference: NewtonScript is *prototype-based*, rather than *class-based*. That is, rather than dividing the world into classes and instances for the purposes of inheritance, NewtonScript objects inherit directly from other objects.

Don't forget everything you know about class-based programming, though. It is possible, and even desirable, to *simulate* classes in NewtonScript. Even though the language contains no explicit features to support classes, you can use a simple set of stylistic conventions to gain the familiar advantages of classes when you need them, without losing the flexibility of prototypes.

What are classes good for?

Newton programming puts great emphasis on the view system. The structure of your application is based around the views that make up the user interface. Newton Toolkit reflects this strong view orientation, making it very easy to create views with attached data and methods. However, it's not necessarily appropriate to use the view system alone to organize your program.

Most applications of any complexity use various independent, fairly complicated data structures. A standard programming technique is to implement these structures as "abstract data types" that encapsulate the functionality. In an object-oriented program, these take the form of classes.

Classes let you divide your program's functionality into manageable pieces. By combining a data structure with the functions that operate on it, classes make the program more understandable and maintainable. A well-written class can be reused in other applications, which saves you effort. There are plenty of reasons why classes are a good idea; you can look in any book on object-oriented programming for more.

You should use the view structure of your application to divide it into parts according to the user interface. It's a good idea to implement some or all of the internal data structures as classes.

Classes: a brief reminder

Let's start by reviewing the traditional class-based model of object programming. I use Smalltalk concepts and terminology in this article; C++ folks will need to translate the discussion slightly to fit their frame of reference.

The principal concept in the class-based model is, not surprisingly, the *class*. A class defines the structure and behavior of a set of objects called the instances of the class. Each instance contains a set of *instance variables*, which are specified in the class. Instances can respond to *messages* by executing the *methods* defined in the class. Every instance has the same instance variables and methods. In addition, the class can define *class variables* and *class methods*, which are available to all the instances of the class.

Inheritance is also determined by classes. Each class can have a *superclass*, from which it inherits variable and method definitions. In some languages, a class can have multiple superclasses, but there's no easy way to simulate that in NewtonScript, so I won't consider that here.

An object is created by sending a message, usually called "New" or something similar, to its class. It may also be created by sending a "Clone" message to an instance of the class. When a message is sent to an instance, the corresponding method is located in the class (or superclasses) and executed. The method can refer directly to instance variables from that particular instance, and to class variables.

Inheritance in NewtonScript

The NewtonScript object model is prototype-based. Frames inherit directly from other frames; there are no classes. A frame may be linked to other frames through its _proto and _parent slots. These slots define the inheritance path for the frame. When you send a message to a frame, the method that executes can use the slots of that frame (the *receiver*) as variables. If a variable or method reference cannot be resolved in the receiver, the proto chain is searched. If the desired slot still isn't found, the search moves one step up the parent chain, searching the parent and its proto chain, and so forth.

These rules came about because they are a good fit for the Newton programming environment, which is oriented around the view system. Parent inheritance provides inheritance of variables and messages through the view hierarchy: you can define a variable in a view for all its subviews to access. Proto inheritance allows views to share common templates, and also lets most of the data stay out of RAM.

Even though the inheritance system (and all of NewtonScript) is closely integrated with the view system, it is really just a set of rules that can be applied in whatever way you find useful. You can send messages to any frame, not just a view frame, and non-view frames can take advantage of the inheritance rules as well. As this article will demonstrate, the same rules are suitable for a form of class-based programming.

The basic idea

I will now describe the basic class-based NewtonScript technique. Remember that there is no built-in idea of a class or an instance in NewtonScript; this is just a set of conventions for using NewtonScript that will let you create structures similar to those you would create in a class-based language. Thus, although I will use the terms "class", "instance", and so forth, they are all just frames being used in specific ways.

The main idea is to use parent inheritance to connect instances with classes. An instance is a frame whose slots make up the instance variables, and whose _parent slot points to the class (another frame). The class's slots make up the methods.

As a simple example, consider a class "Stack" that implements a push-down stack. It has the standard operations Push and Pop that add and remove items, and a predicate IsEmpty that determines if there are any items on the stack. The representation is very simple: just an array of items and an integer giving the index of the topmost item.

The class frame looks like this:

```
Stack := {
  New:
       func (maxItems)
         {_parent: self,
          topIndex: -1,
          items: Array(maxItems, NIL)},

  Clone:
       func () begin
         local newObj := Clone(self);
         newObj.items := Clone(items);
         newObj
       end,

  Push:
       func (item) begin
         topIndex := topIndex + 1;
         items[topIndex] := item;
         self
       end,

  Pop:
       func () begin
         if :IsEmpty() then
           NIL
         else begin
           local item := items[topIndex];
           items[topIndex] := NIL;
           topIndex := topIndex - 1;
           item
         end
       end,

  IsEmpty:
       func ()
         topIndex = -1
};
```

The class frame begins with the New method. This is a "class method" that is intended to be

used as message of the Stack class itself, as in Stack:New(16). It consists simply of a frame constructor that builds an instance frame. An instance always has a _parent slot that refers to the class frame; note that because New is a message intended to be sent to the class, it can just use self to get the class pointer. The rest of the slots contain the instance variables: a topIndex slot for the index of the topmost item (-1 if the stack is empty) and an items slot for the array of items. New takes an argument that determines the maximum number of items on the stack, but it would be easy to make this dynamic (if it didn't have to fit in an article like this).

It's usually a good idea to provide a Clone method for a class. This lets you make a copy of an object without having to know how deep the copy has to go (such knowledge would violate the encapsulation that is one of the reasons to have a class in the first place). In the case of Stack, a simple Clone would leave the items array shared between two instances, which would result in confusing and incorrect behavior. A DeepClone, on the other hand, would copy the entire class frame along with the instance, because of the pointer in the _parent slot. That would actually work in this case, although it would waste huge amounts of space--watch out for this sort of mistake. The correct way to clone a Stack is to clone the instance, then give the clone a clone of the items array, which is what the Clone method above does.

After the New and Clone methods, which are usually present in any class, come the methods particular to this class. The Push method increments topIndex and adds the item to the end of the items array. Note that instance variables such as topIndex and items are accessed simply by their names, because they are slots of the receiver. The Pop method calls the IsEmpty method to see if the stack is empty. If so, it returns NIL; if not, it returns the topmost item and decrements topIndex. It assigns NIL to the former topmost slot so it won't prevent the item from being garbage collected.

Code to use a class is similar to code you would write in a language like Smalltalk. You create an object by sending the New message to its class. You use the resulting instance by sending messages to it. Of course, you do all this in NewtonScript syntax, which uses the colon for sending a message.

```
s := Stack:New(16);
s:Push(10);
s:Push(20);
x := s:Pop() + s:Pop();
```

At the end of this code, the value of x will be 30.

Practical issues

Before getting into more advanced topics, here's some practical information about doing class-based programming with current tools. Newton Toolkit as it exists today doesn't include any features that specifically support class-based programming; for example, the browser only shows the view hierarchy. Nevertheless, it's not too hard to get classes into your application.

You need to build the class frame itself into your package, and you need to make it accessible from NewtonScript code. You can do both at once by putting the class directly into an evaluate slot of your application's main view. For the above example, you could add a slot called Stack to the main view and type the class frame just as it appears (but not including the "Stack :="

line) into the browser.

If you prefer, you could make the class frame a global compile-time variable by putting it (including the assignment this time) into Project Data. That won't make it available to your runtime code, however; you still have to create the Stack slot in the main view, but you can just enter "Stack" as its value. You have to put the class in Project Data if you want to use superclasses (more on this later).

Class variables

You can create "class variables", that is, variables that are shared by all instances of a class, by adding slots to the class frame. This is the same way you add variables to a view to be shared by the subviews, but it's a bit more tricky because the view system does something automatically for views that you have to do manually for classes.

Remember that your class frame, like all other data built at compile time, is in read-only space when your application is running. It's not possible to change the values of the slots; thus, it's impossible to assign a new value to a class variable. The view system gets around this problem by creating a heap-based frame whose _proto slot points to the view and using that frame as the view. The original slots are accessible through proto inheritance, and assignments create and modify slots in the heap-based frame, overriding the initial values in the original. You can use the same trick for your class frame.

For example, let's say you want to have a class variable x whose initial value is zero. The class frame, defined in the Project Data file, contains a slot named "x":

```
TheClass := { ... x: 0 ... }
```

The base view has a slot called TheClass whose value is defined simply as "TheClass". At some point early in your application's execution, perhaps in the viewSetupFormScript of the view where your class frame is defined, create a heap-based version of the class and assign it to the variable TheClass:

```
viewSetupFormScript:
    func () begin
       ...
       if not TheClass._proto exists then
           TheClass := {_proto: TheClass};
       ...
    end
```

Now you can use and assign the variable x in methods of TheClass. The instances will inherit it via their _parent slot, and the first time x is assigned, an "x" slot will be created in the heap-based part of the class, shadowing the initial value of zero. Note that you only want to do this setup once--otherwise you'll end up with a chain of frames, one for each time your application has been opened. Checking for the _proto slot, as above, is one way to ensure this; you could also set TheClass := TheClass._proto in your main view's viewQuitScript.

Superclasses

It's easy to get the close equivalent of a superclass: just give the class a _proto slot pointing to its superclass. This requires the class definitions to be in Project Data so their names are available at compile time. For example, if you have a SortedList class that should have Collection as its superclass:

```
Collection := { ... };
SortedList := {_proto: Collection, ...};
```

Of course, you have to define Collection before SortedList to do it this way. If you prefer to reverse their definitions for some reason, you can add the _proto slot later:

```
SortedList := { ... };
Collection := { ... };
SortedList._proto := Collection;
```

If you override a method in a subclass, you can call the superclass version using the inherited keyword. If you have class variables, note that because assignments take place in the outermost frame in the _proto chain (that is, the wrapper you create at initialization time for each class), each class gets its own version of the class variable.

Using classes to encapsulate soup entries

One use to consider for classes is encapsulating the constraints on soup entries. Normally, entries in a soup are "dumb" frames: simple data records with no _parent or _proto slots to inherit behavior. The reason is obvious: if they did, each entry would contain a copy of the inherited frame. (Actually, _proto slots are not followed in soup entries anyway, for various reasons.) Thus, soup entries are not normally used in an object-oriented fashion.

Unfortunately, soup entries generally have somewhat complicated requirements, such as a set of required slots, so it would be nice to give them an object interface. You can do this by defining a class as a "wrapper" for a particular kind of soup entry. In addition to a New method, it can have class methods to retrieve entries from the soup and create objects from existing entries, and instance methods to delete, undo changes, and save changes to the entry. Each instance has a slot that refers to its soup entry.

Given such a wrapper class, you can treat the soup and its contents as objects, sending messages to entries to make changes and retrieve data. The class is then a central location for the code that implements the requirements for entries in the soup.

ROM instance prototypes

If your instances are fairly complicated and have a lot of slots whose values aren't all likely to change, you can save some space by using the same _proto trick as classes and views. That is, in your New method, create the instance as a little frame that just refers to the class and an initial instance variable prototype that stays in application space:

```
New: func ()
       {_parent: self,
        _proto: '{ ...initial instance vars... }}
```

Leaving instances behind

Because so much is contained in the application, it's very difficult to make instances that can survive card or application removal. The only way to do this is to copy the entire class (and its superclasses, if any) into the heap, which would probably take up too much space to be practical.

Conclusion

This technique doesn't exactly simulate any existing class-based object system, but it gives you what you need to encapsulate your data types into class-like structures. I find it to be very useful (stores, soups, and cursors all essentially follow this model), and I hope it will help you create better Newton applications. Have fun!

Walter Smith joined the Newton group in 1988. He is the principal designer and implementor of NewtonScript and the Newton object store.

wrs@apple.com, 8 July 1994