A (NewtonScript) Inheritance Primer

Paul R. Potts
potts@oit.itd.umich.edu

Many developers have a difficult time understanding the notion of inheritance. Inheritance is a set of rules which are specific to individual languages. However, in addition to the language, developers using a particular language tend to use a given inheritance idiom. Particular class libraries tend to favor one idiom over another. Developers using a language such as C++ tend to think of inheritance differently than developers using NewtonScript, Scheme, Eiffel, Smalltalk or other languages with inheritance features. In addition, languages such as C++ evolve over time, adding new features.

All of this means that inheritance is really a tower of Babel for developers. In the first section of this article I discuss some aspects of inheritance using C++ as an example. Then, in part two, I look at exactly how the inheritance rules operate in NewtonScript and how the Newton's view system serves as an idiom (and rationale) for these rules. If you think like the designers of NewtonScript and its view system, you should be able to remember the NewtonScript inheritance rules and use the system to its best advantage.

INHERITANCE IN GENERAL

Real-World Thinking About Inheritance

In biological terms, the term inheritance is generally used to refer to genetic inheritance that children share with parents. What are some of the things we think about when we think about inheritance?

First, there are some obvious things:

- Inheritance takes place from one or more (in the real world, typically two) parents.
- Characteristics of the parents, or ancestors of the parents, occur in the children.

Next, there some that aren't so obvious:

- Inheritance from two parents is governed by a complex set of rules and there is a significant element of randomness involved.
- Inheritance alone is not sufficient to completely determine an organism's form.

In computer programs, we usually don't want any nondeterministic behavior, so we hope there is no randomness involved in the inheritance rules. We also hope that our program's behavior is not unpredictable (programs that do this are generally said to contain bugs). So, let's leave the latter two statements out of our thinking about inheritance in computer programs.

This leaves us with the first two statements. Since they are so simple to express, they seem like good starting points for modeling inheritance in programs.

Inheritance in Computer Programs

In the history of innovations in the design of computer programs and languages, inheritance is a natural evolution of the notions of functional decomposition and information hiding that are familiar to users of structured programming techniques. Functional decomposition and information hiding help users to manage complexity; inheritance is another tool to help achieve the same goal.

In (Booch), Grady Booch uses the following definition of *object-oriented programming:*

> *"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships* (Booch, p. 38).

Booch uses the notion of classes, an extension of the type system that many languages such as Pascal and C already implement. I won't talk about this distinction further, since it is irrelevant in NewtonScript.

NewtonScript's design is partially modeled after that of SELF, a language with no classes, but only objects. For more information on Self, see (Ungar) and (Chambers). You can simulate classes using NewtonScript: see Appendix C of *The NewtonScript Programming Language* which is included with the Newton Toolkit.

The Inheritance Idiom

In (Booch chapter 2), Booch goes on to describe two hierarchies in complex systems:

> *"The two most important hierarchies in a complex system are its class structure (the 'is a' hierarchy) and its object structure (the 'part of' hierarchy).*
>
> *"Semantically, inheritance denotes an 'is-a' relationship. For example, a bear 'is a' kind of mammal, a house 'is a' kind of tangible asset, and a quicksort 'is a' particular kind of sorting algorithm. Inheritance thus implies a generalization / specialization hierarchy...* (Booch, p. 59).

Note that this definition of inheritance is an idiom, and not enforced by either C++ or NewtonScript. This kind of inheritance is the recommendation of many developers after years of thinking about object-oriented design. There is no technical reason why the "part of" hierarchy Booch speaks about cannot be handled by an inheritance relationship; it just isn't traditionally done that way. In NewtonScript, as we shall see, an inheritance link can be and is used for just that.

Booch also limits his thinking when he states that a subclass always specializes the superclass. This may be one of the most effective and efficient uses of inheritance as it exists in a language like C++, but it is not the only one. (Jacobson) shows that this general/specific relationship is useful because subclasses should only contain that which is different from the superclass. However, the subclass can also be thought to extend a class (Jacobson, p. 61) – if you think of a class in terms of its additional functionality (methods) and new data members, you extend it; if you think of a class in terms of its behavior in your hierarchy, or in the way it models categories of real-world classes, you may be thinking in terms of specialization.

While there is no technical reason why you can't use inheritance rules in other ways, keep in mind that if you do so, other programmers may have a hard time understanding what you are doing. You will be speaking the same language, but using your own idiom.

Single Inheritance

Let's look at some classes. Figure 1 shows examples of "is-a" relationships between classes. An `apple` "is a" `fruit`, which "is a" `food`. The `pear` and `apple` are both `fruits`; `veggies` and `fruits` are both `foods`. (Class names borrowed from Sphar, p. 37).

(Some of you may be wondering whether the tree on the right still represents single inheritance, if their are several arrows pointing to one class. Yes, it does, because single inheritance means that each class has at most one parent. Parents can have multiple children.)

In the example above, we say that `fruit` and `apple` are descendants of class `food`; `food` is an ancestor of its two subclasses. (Some textbooks use child and parent, subclass and superclass; it really doesn't matter as long as the meaning is understood). The use of arrows to point from a child class back to the parent is traditional and used in many different textbooks independent of the actual language used to implement the inheritance relationships.

Polymorphism

Polymorphism is the ability of one object to take on many shapes (poly = many, morph = shapes). In computer programming, polymorphism means that the type of an object can be determined at runtime. In C++, this means that a variable which is of a particular class can also be used to hold objects who are descendants of this class. Let's look at some uses to which this can be put.

- Polymorphism can be used to send the same message to a variety of different objects; the exact class of each need not be known at compile time.

Let's assume that `apple`, `pear`, and `food` all have an `eat` method (see Figure 2). If `fruit` and `apple` are children of `food`, and we have a variable of type `food`, in C++ we could assign a descendant of type `food` (such as `apple`) to this variable. (See Sphar, p. 37). Sending the `eat` message to this variable then calls a different method, depending on the class of the object, which is determined at runtime. One variable can take on different possible behaviors (many shapes).

- Polymorphism is extremely useful for collections. For example, an array of type `food` can contain objects whose classes are children of class `food`.

In C++, the chief feature that allows polymorphism is the virtual method. A virtual method is a method which may be overridden at runtime in a child class. A pure virtual method is a virtual method that must be overridden – calling it directly is an error. This kind of method is useful for creating abstract base classes that can't "live" on their own. For example, it makes no sense to provide a method called `cook` for the class `food`, since the cooking method depends on the kind of `food` (subclass). Abstract base classes can be simulated in NewtonScript the same way they are in Smalltalk – just write your base class method so that it gives an error message if called.

In C++, inheritance is based on the notion of class and subclass. While objects can be polymorphic, they cannot alter their behavior beyond the predefined set of possible methods in themselves or their parent classes. In NewtonScript, inheritance is dynamic, and can be changed at runtime. We examine an example of this later.

Multiple Inheritance

In multiple inheritance, not only can one parent have multiple children, but one child can have multiple parents. Where any combination of single inheritance can be represented by a tree structure, multiple inheritance is represented by a directed acyclic graph.

In newer class libraries, the use of multiple inheritance allows the implementation of mix-in classes. In theory, this means you can add the necessary behavior to a class by simply mixing in the classes you want. For example, if you have a generic `array` class, and you want to make it streamable, so that you can save it to disk and restore it easily, you could define

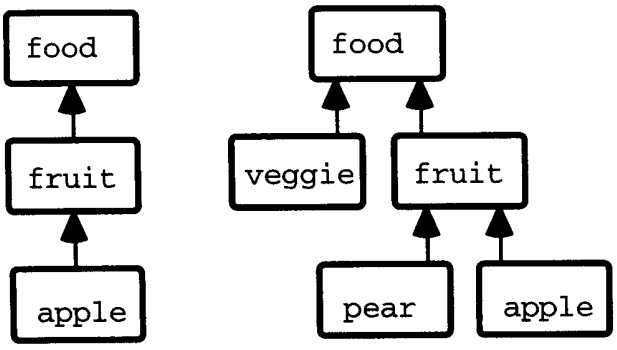


Figure 1 - "is-a" relationships.

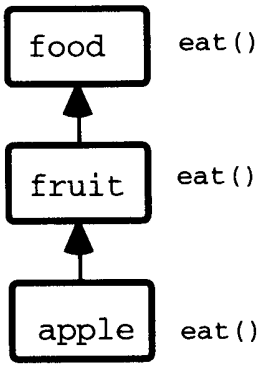


Figure 2

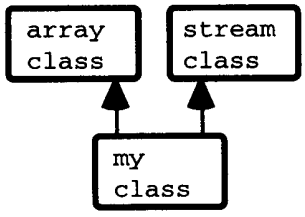


Figure 3

a class that inherits from both the `array` and the `stream` class (see Figure 3). Your objects would then be able to receive messages which are directed to either parent. Several of the newer commercial class libraries, such as the Booch Components and Code Warrior's PowerPlant, use multiple inheritance.

In C++, multiple inheritance can be very complicated. According to (Waldo, 102)

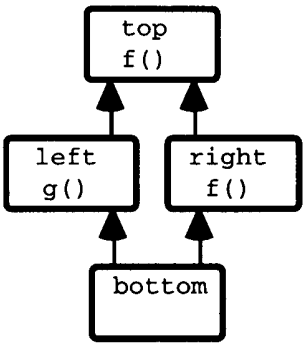
"...the real expressive power of inheritance is delivered by just one of its six variants... yet we must learn the complexity of all six variants' interactions with other language features, such as initialization, virtual functions, overloading, and conversions.

Take a look at the example below, which is borrowed from (Waldo), p. 103:

```
class top {
  public: virtual void f() { printf("top::f()"); }
};
class left: public virtual top {
  public: void g() { f(); }
};
class right: public virtual top {
  public: void f() { printf("right::f()"); }
};
class bottom: public left, public right {
};
main() {
  bottom x;
  x.g();
  return 0;
}
```

This is a deliberately contrived example, but it demonstrates a potential problem in multiple inheritance: ambiguity. C++ exacerbates the situation because instead of disallowing the potential name conflict between the two `f` methods in `top` and `right`, C++ allows the potential ambiguity, which must be resolved by the compiler. The upshot of this is that classes which use multiple inheritance must be very carefully written to avoid situations like the one shown in Figure 4.

I bring up this example to illustrate that there exists a debate over whether the complete, complex inheritance mechanism in C++ is justified by its usefulness in creating real-world applications. In NewtonScript we have an example of a language that supports a particular, simple type of multiple inheritance which has already shown its usefulness for building Newton applications.




```
top has method f

left has method g,
which makes a call to
method f.

right has method f.

Call class bottom's
method g. Which
instance of f is
eventually called?

In C++ the surprise
answer is right::f.
```

Figure 4 - multiple-inheritance confusion.

Mangled Inheritance

Suppose you have some objects that inherit from each other in a circular order (see Figure 5). This is an invalid inheritance chain. Some environments may be able to detect such a construct, but if you ask NewtonScript to search an object like this, the system goes into an endless loop. I bring this up only to show one of the possible error conditions you can produce when experiments with inheritance go wrong.

Software Design Using Inheritance in C++ and NewtonScript

When object-oriented languages began to appear, it was widely thought that these languages were perfectly suited to programming graphical user interface systems such as the Macintosh. After my initial exposure to THINK C with Objects, I was skeptical.

The THINK Class Library uses single inheritance as a powerful way to structure the behavior of classes, but this behavior hierarchy is distinct from the system's visual hierarchy, where objects such as buttons and check boxes are contained within other objects such as dialog boxes and windows. It doesn't make sense for a button to share most of the behavior of a window. With only single inheritance, this leads to a complex scheme of inserting pointers to objects into other objects. Other class libraries such as Borland's TurboVision for C++ use similar techniques.

Now, the developer community is beginning to get its hands on class libraries that use multiple inheritance. I use C++ daily and have great appreciation for some of the features, such as the stream library, that multiple inheritance can make possible. I don't quite feel competent, however, to write a reusable class library of useful components. Very few users of C++ do. Copy constructors, default base class destructors, exception handling, the many complex uses of the `const` keyword, name spaces – few compilers can handle it all well, and few users can, either. Even now when reading other developers' code, I find myself reaching for my C++ reference books more times than I care to admit.

With only a few months of experience with NewtonScript under my belt, I do not feel nervous sharing reusable NewtonScript objects with the developer community. Many users are finding that C++ is just too complex a world to comfortably inhabit, especially given the latest set of new features. Although I find C++ to be very powerful, I doubt that I would find too many users who would not agree that the language has grown to the point where using a well-chosen subset is nearly essential to retain one's sanity.

NewtonScript, on the other hand, was designed to be a simpler alternative. Let's take a look at what I consider some of the key differences.

- *At runtime, C++ objects are aggregates of all the data objects in their inheritance paths.* This means that if an object comes from several classes, each of which has a number of data members, at runtime the object takes up a large amount of memory. Inheritance in NewtonScript is based on *difference*.

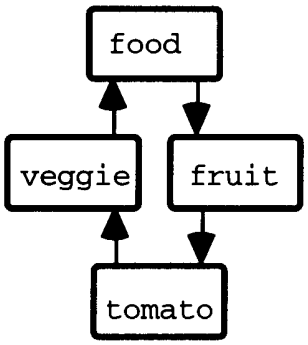


Figure 5 - mangled inheritance.

- *In C++, polymorphism is complex because of the type system.* In NewtonScript, polymorphism is effortless, because of the ability of slots to hold any kind of object and for objects to alter themselves and the contents.

For building large systems, NewtonScript also has some definite disadvantages over the C++ inheritance model. One of these is security. In C++, classes can contain methods and data which are public, private or protected – inherited classes cannot violate the security placed on these objects. This is useful for large-scale programming projects in which multiple developers are working on components of a system, or for commercial classes in which only compiled libraries and header files are provided. The NewtonScript programmer is free to interfere with other classes, either accidentally or purposefully, and little can be done to guard against this kind of behavior.

C++ is also type-safe. This means that the compiler can check for many kinds of type violations at compile time (for example, assigning a long integer to a data member of a class which is supposed to hold a floating-point value). In NewtonScript, this is a perfectly legal activity, and the user does not discover the error until they notice incorrect behavior at runtime, perhaps when they try to send the contents of the slot to a function that is expecting a floating-point value.

NEWTONSCRIPT INHERITANCE

NewtonScript uses a simple type of multiple inheritance between frames, in which inheritance links are maintained via two particular reserved slot names, `_parent` and `_proto`. There is nothing magical about the slots themselves; for frames in RAM, you can create the slots yourself, modify them, remove them and use them to link frames in any manner you choose. The special part comes from the way NewtonScript automatically uses these slots to look up methods or slots using its inheritance rules.

The Slots and Nothing But the Slots

In NewtonScript, inheritance takes place *only* via the `_parent` and `_proto` slots. This may seem obvious, but programmers accustomed to other languages may need to make some adjustments. If you are accustomed to a language like Pascal, that lets you declare a function inside another function, where the innermost function then can access the containing function's variables, you may be initially tempted to write something like this in NewtonScript:

```
outerFrame := {yin: 1, yang: 2, innerFrame:
       {z: func () print (yin)}};

call outerFrame.innerframe.z with ();
```

This does not produce the desired results. The function `z` can't "see" the slot `yin`. NewtonScript doesn't have any mechanism for following a trail from `innerFrame` to `outerFrame`, except via the special slots named `_parent` and `_proto`, which we don't provide in this example.

Although the constructor call that creates `innerFrame` is inside the constructor call that creates `outerFrame`, `innerFrame` isn't really "inside" `outerFrame` in any real sense. All that is happening here is that, at runtime, a frame constructor function is called which builds `innerFrame`, then puts a reference to `innerFrame` in the slot named `innerFrame` in the frame called `outerFrame`, which it also builds at runtime. The same results can be achieved by the following:

```
z := func () print ("Test received");
innerFrame := {z:z};
outerFrame := {yin: 1, yang: 2, innerFrame:innerFRAme};
```

Lexical scoping, like that found in Pascal, is provided only by the function environment mechanism (see *The NewtonScript Programming Language* for more information on function environments).

Are You Receiving Me?

In the examples in this article, we create some frames and link them up via the `_parent` and `_proto` slots. When we speak of starting an inheritance lookup at a particular frame, what we really mean is that the frame in question is the current receiver. Our examples don't always show this because they aren't fully executable programs. In the case of looking up an inherited function, the function that is eventually found and executed is referred to as the implementor. For more information on these terms, take a look at *The NewtonScript Programming Language*.

These terms don't strictly apply to the examples, since I execute functions that simulate the inheritance mechanism indirectly. In my examples, the receiver is represented by the frame that supposedly contains the currently executing function. In the inheritance diagrams, it is always located at the bottom left. It is the starting point, in each case, for the inheritance search.

NewtonScript's Inheritance Rules

NewtonScript's inheritance rules were designed around the view system, although they also work with frames that aren't part of the view system. Understanding the view system helps you understand the justification for these rules. In the synopsis below "view" means a frame that is part of a NewtonScript program using the view system.

- The `_parent` slot is used to represent containment; innermost views inherit from their containing views.
- The `_proto` slot is used to inherit behavior. Most of the prototype chain for a view is located in ROM, in order to achieve the design goal of minimizing the runtime use of RAM.

I go into more detail as we look at each set of behaviors.

Getting a Slot Using a Frame Name

When you access a slot using a frame name, such as `myFrame.dot`, NewtonScript's inheritance mechanism only traverses the `_proto` chain. This guarantees that if you use an explicit frame name to look for a slot, you only find it in your own frame or in one of the prototype frames, which are typically in ROM. This is not altered by changing the containment hierarchy (the `_parent` chain). This is written incorrectly on the quick reference card in *The NewtonScript Programming Language*, which indicates that only the `_parent` chain is searched.

Let's look at how this traversal works. Put the following in your Project Data file:

```
Apple := {label: "Apple"};
Fruit := {label: "Fruit"};
Food := {label: "Food", target: "Our target!"};

// Food is our top-level pseudo-class, and has
// no _proto slot.

Apple._proto := Fruit;
Fruit._proto := Food;
```

This builds a simple hierarchy using the `_proto` slot (see Figure 5).

Let's pretend that `Apple` is the receiver. Let's write a function to enumerate the frames that are in our class hierarchy, by printing the `label` slots. (We want a general function, not one that is specific to our particular set of frames). Think for a

moment about how you would implement such a function. If you are familiar with recursion, try thinking of a recursive solution. If you aren't familiar with recursion and you would like to be, I highly recommend taking a look at the language Scheme, which is one of NewtonScript's closest ancestors. See (Springer).

Now that you've thought about it, here is a function that does it by cheating:

```
familyTree1 := func (frame)
  foreach slot, value deeply in frame do
    if slot = 'label then print (value);
```

If you put the above code in your project's Project Data file, you can execute it at build time by adding the following code:

```
afterScript := call familyTree1 with (Apple);
```

This very short function is sneaky and takes advantage of the fact that `foreach` deeply searches the `_proto` chain on its own so we don't have to. Next, let's do it on our own.

Here's a version that does it recursively:

```
familyTree2 := func(frame)
  if frame then
  begin
    print (frame.label);
    call familyTree2 with (frame._proto);
  end;
```

Here's a method that does the same thing iteratively instead of recursively, and prints the results in a formatted form in the Inspector window.

```
familyTree3 := func (frame)
begin
  local ourself := frame;
  write (ourself.label);
  while ourself do
  if ourself := ourself._proto then
    write (" is a child of" && ourself.label);
    write ("\nEnd of family tree\n");
  return NIL;
end;
```

You can call these functions from the `afterScript` using the same method as the `familyTree1` function.

Getting a Slot

When NewtonScript comes across a symbol in a function that is used as a value (for example, `return target`), here is what it does:

- First, it checks to see if the symbol is a local variable that is defined in the current function. Note that this also checks the lexical environments of any enclosing functions (see *The NewtonScript Programming Language* for more information on function environments).
- Second, it looks for a slot with a matching name. This search starts with the receiver, and continues on using `_proto` and `_parent` inheritance as described below.
- Finally, NewtonScript looks for a global slot with a matching name.

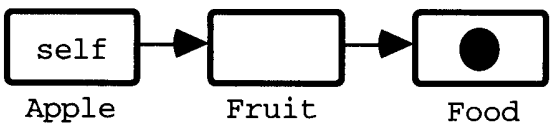


Figure 5

If the slot is not found after trying all three of these methods, an exception is generated.

Let's describe how the slot lookup works using the `_proto` and `_parent` inheritance chains. In Figure 6, our target is represented by the black dot – it's a slot called `target`. Arrows pointing to the right represent `_proto` links, arrows pointing up are `_parent` links. Our target is located at `self._parent._parent._proto`. That's the shortest route there, but then, we can see the whole graph. NewtonScript can't. NewtonScript uses its inheritance rules.

What happens when we refer to `target`? (Let's assume that it isn't a local variable in our function environment). Slot lookup begins. `Target` isn't in the receiver, so NewtonScript tries to search the receiver's `_proto` chain. Our first frame has no `_proto` slot, so we've exhausted the `_proto` chain. The search goes up to the `_parent` slot, and we start again. According to *The NewtonScript Programming Language* "prototype inheritance takes precedence over parent inheritance; all prototype frames on one level are searched before moving up to search a parent frame and its prototypes on another level" (p. 5-5). The route that NewtonScript takes is illustrated in Figure 7.

NewtonScript tries the following order:

```
1. self
   (NewtonScript sees the proto chain is exhausted,
   since self has no proto chain).
2. self._parent
3. self._parent._proto
4. self._parent._proto._proto
   (NewtonScript sees the proto chain is exhausted,
   so it backs up and starts again with the parent)
5. self._parent._parent
6. self._parent._parent._proto
   (Success!)
```

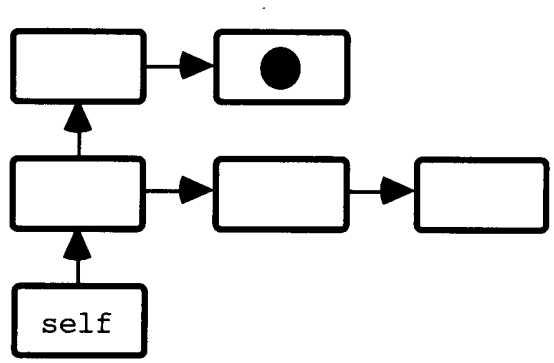


Figure 6

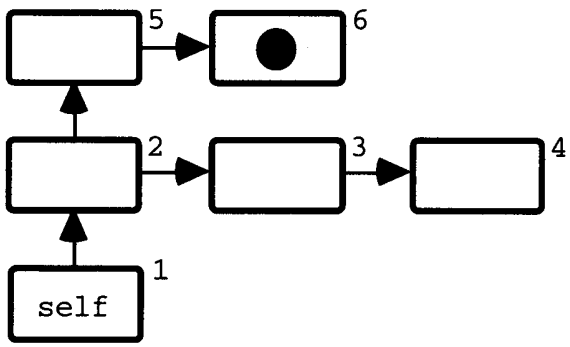


Figure 7 - NewtonScript's search route.

If you've studied computer science, you might have noticed that if you rotate Figure 6 by 180 degrees so that `self` is at the top, it resembles a binary tree. The analogy is an imperfect one, but it might be encouraging to remember that NewtonScript's inheritance structure is no more complicated than a binary tree. The rules that NewtonScript uses for searching this data structure, however, are quite different, and our frames are not sorted or balanced like binary trees usually are.

If NewtonScript was written in NewtonScript, the code to get a slot might look something like this:

```
recursiveSearchProtoChain := func (frame, slotName)
begin
  local result := nil;
  if frame then
    if HasSlot(frame, slotName) then
      result := frame;
    else
      result := call recursiveSearchProtoChain with
        (frame._proto, slotName);
  return result;
end;
recursiveSearchParentAndProtoChain := func (frame,
slotName)
begin
  local result := nil;
  if frame then
    if not result := call recursiveSearchProtoChain
      with (frame, slotName) then
      result := call recursiveSearchParentAndProtoChain
        with (frame._parent, slotName);
  return result;
end;
```


To use this code, you would call `recursiveSearchParentAndProtoChain` with a reference to a starting frame (representing the receiver in our example) and the name of a slot to look for. In the example above, this would look something like this:

```
if call recursiveSearchParentAndProtoChain with
    (self, 'target) then
  print ("Found it!")
else
  print ("Failed to find target.");
```

The next set of frames shows a working example combining system prototypes and the parent containment hierarchy – envision a radio button inside a radio button group inside a clView, with all of these frames pointing to prototype frames in ROM.

```
fakeROMproto1 := {label: "fake clView proto"};
fakeROMproto2 := {label: "fake radioButton group
                  proto"};
fakeROMproto3 := {label: "fake radioButton proto"};
fakeRadioButton := {label: "fake radio button"};
fakeRadioButtonGroup := {label: "fake radio button
                  group"};
fakeclView := {label: "fake clView"};

// put our target in a slot in a specific frame

fakeROMproto1.target := "our target!";

// now, set up dual inheritance relationship

fakeRadioButton._proto := fakeROMproto3;
fakeRadioButtonGroup._proto := fakeROMproto2;
fakeclView._proto := fakeROMproto1;
fakeRadioButton._parent := fakeRadioButtonGroup;
fakeRadioButtonGroup._parent := fakeclView;
if call recursiveSearchParentAndProtoChain with
    (fakeRadioButton, 'target) then
  print ("Found it!")
else
  print ("Failed to find target.");
```

Feel free to borrow this code, play with it, and add `print` statements along the way to see what is happening. If you understand this simulation, you understand slot lookup in NewtonScript.

For those who might have trouble with recursion, here's a version of the same function that works iteratively instead of recursively:

```
IterativeSearchParentAndProtoChain :=
  func (frame, slotName)
begin
  local parentWalker := frame;
  local protoWalker := frame;
  while parentWalker do
  begin
    while protoWalker do
    begin
      if HasSlot (protoWalker, 'target)
        then return print("found it!");
      protoWalker := protoWalker._proto;
    end;
    protoWalker := parentWalker._parent;
    parentWalker := parentWalker._parent;
  end;
end;
```

A NewtonScript Method Call

A NewtonScript method call, also called sending a message or calling a function, traverses the `_proto` and `_parent` chains. The traversal order is the same as the order for getting a slot. You can easily write some test frames of your own and put functions with the same names in different positions, to see for yourself how this works.

A NewtonScript Method Call Using the Inherited Keyword

The special case of sending a method call using the `inherited` keyword, as in the form `inherited:method()` or `inherited:?method()` also only traverses the calling object's `_proto` chain. Using the `inherited` keyword does not change the receiver. You can't use the `inherited` keyword and supply a specific frame to send the message to.

Why doesn't the form `inherited:method` traverse the `_parent` chain? Consider, for example, a button inside a clView. The button's `viewSetupFormScript` doesn't necessarily inherit default behavior from the `_parent` frame, since this relationship is one of containment, rather than specialization of behavior. The default behavior of a button is in its ROM `_proto`, not in its containing clView.

Setting a Slot

Next, let's set the value of `target`. This could be accomplished by a line of NewtonScript such as `target := 5`. How does NewtonScript search for the place to set `target`? First, I assume that there is no local variable in the function environment with the desired name (if there is one it's used). I also assume there is no global with the same name to use if the inheritance search fails. For a complete explanation of the search rules, see *The NewtonScript Programming Language* p. 5-7. Here's a summary: a slot is never set in any of the frame's `_proto` ancestors or in a frame's parents; it is always set in the frame or the frame's `_parent` that heads up the `_proto` chain in which the slot was found.

This may sound complicated, but its purpose becomes clear if you understand the rules for getting a slot, and understand that members of the proto chain are generally not in RAM (they are in pseudo-ROM, or in real ROM). (McKeehan and Rhodes, p. 154) suggest that if multiple objects refer to the same `_proto` frame, overriding a slot in one of the objects does not automatically affect the search paths of the others. When you set a slot, the rules guarantee that if you get that slot again, you always get the new value, which stands in the way of NewtonScript's search.

Let's take a look at the example shown in Figure 8 (the dot represents the found slot, the box represents where the shadowing slot is placed). Here is the search sequence:

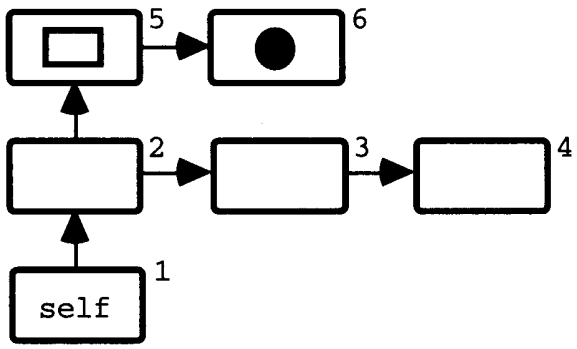


Figure 8

```
1. self
   (NewtonScript sees the proto chain is exhausted,
   since self has no _proto).
2. self._parent
3. self._parent._proto
4. self._parent._proto._proto
   (NewtonScript sees the proto chain is exhausted,
   so it backs up and starts again with the parent)
5. self._parent._parent
6. self._parent._parent._proto
   (Success! NewtonScript never sets a slot in the
   _proto chain, so it sets it back in the chain's
head)
```

If there is no existing local variable, or inherited or global slot to shadow with the set operation, NewtonScript makes a local variable in the currently executing function environment. It is generally considered to be bad style to create a local this way, since it isn't clear what you are doing. I recommend creating local variables explicitly, using the `local` keyword. In addition, NewtonScript has to create an additional data structure at runtime to hold this local, so it is accessed less efficiently then if it was created using the `local` keyword. Mike Engber also warns that NewtonScript may disallow this implicit declaration of locals in the future. (Some environments, such as Visual BASIC, let you state whether you allow implicit declaration of local variables or not; I always set this option to require explicit declaration, since it helps catch a common programming error such as misspelling the name of a variable in an assignment statement).

Altering the Inheritance Structure at Runtime

Since slots in any frames stored in RAM can be modified at runtime, and NewtonScript simply uses slots to create its inheritance links, you can modify any RAM-based object's inheritance links at runtime. Again, there is nothing special about the `_parent` and `_proto` slots, except the way that the runtime system uses them to search for slots.

The following example started out as an accident. I was creating a set of sample frames that looked something like this:

```
wackyFrame :=  {label: "Great-grandparent", _proto:
    {label: "Grandparent", _proto:
    {label: "Parent", _proto:
    {label: "Child"}}}};
```

Oh, wait. That's wrong. I made the proto links point in the wrong direction. Parents shouldn't point to children!.

Never fear, I'll just write a function to reverse the links. After all, being a dynamic language, NewtonScript is flexible to the point of being rubbery. (For additional ideas on what you can do with altering inheritance links dynamically at runtime in NewtonScript, take a look at (Ungar)). If my function is correct, the resulting frame should look like this:

```
{label: "Child",
 _proto: {label: "Parent",
   _proto: {label: "Grandparent",
     _proto: {label: "Great-grandparent"}}}}
```

Here's the transformation function. The return value is a reference to the new chain of frames (which now starts with the frame that used to be the end of the chain).

```
recursiveReverseProtoChain := func(frame)
begin
  if not frame._proto then return frame;
  local oldProto := frame._proto;
  local result := call recursiveReverseProtoChain
       with (oldProto);
  RemoveSlot (frame, '_proto);
  oldProto._proto := frame;
  return result;
end;
```

An iterative version of the above function is left, as they say, as an exercise for the reader. Have fun playing with this code, and if you make any interesting discoveries, drop me a line.

I would like to express my thanks to Mike Engber and Walter Smith for reviewing my code, finding many errors, and even supplying a working function or two when I got stuck. With luck and their assistance, we have caught most of the technical errors. Any remaining bugs are, as usual, my own fault.

ANNOTATED BIBLIOGRAPHY

There are lots of good books available on object-oriented programming. Some of them are specific to individual languages, and most tend to focus on one particular idiom for the use of object-oriented concepts. These are some of the books and papers I have studied, all or in part, in the past few years:

Agesen, Ole, et. al. "Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance." This paper is available in PostScript form via anonymous ftp from self.stanford.edu. It's a scholarly paper, and I couldn't understand much of it, but I did pick up some interesting tidbits. For example, in SELF, the syntax for getting an object from a slot is the same as the syntax for invoking a method in that slot which returns an object. This would be an interesting feature for a future version of NewtonScript, and useful for rapid prototyping and implementing default behaviors for objects. You could even change the contents of a slot from a data object to a function at runtime.

Booch, Grady. *Object-Oriented Analysis and Design with Applications*, second edition. Redwood City, CA: Benjamin/Cummings, 1994. This is probably the most accessible book on object-oriented design available, and is quite comprehensive. Booch provides a specialized notation for designing classes.

Coad, Peter and Nicola, Jill. *Object-Oriented Programming*. Englewood Cliffs, NJ: Prentice Hall (Yourdon Press), 1993. This book shows examples in C++ and Smalltalk, and comes with a source code disk. It is especially fun for its presentation of a number of informal and usable guidelines on how to create reusable classes to model real-world problems.

Jacobson, Ivar, et. al. *Object-Oriented Software Engineering: a Use Case Driven Approach*. Reading, MA: Addison-Wesley (ACM Press), 1993. This is a comprehensive textbook on object-oriented design. It focuses largely on the formal process of software design. It contains an interesting chapter on the design of software components.

McKeehan, Julie and Rhodes, Neil. *Programming for the Newton: Software Development with NewtonScript*. Cambridge, MA: Academic Press, Inc. (AP Professional Imprint), 1994. This is the first major book for Newton developers outside of the Newton Toolkit documentation, and includes a demonstration version of the Newton Toolkit software. Given the relatively high cost of the Newton Toolkit, this bundle is probably the easiest way to get your feet wet in Newton programming.

Meyer, Bertrand. *Eiffel: the Language*. New York, NY: Prentice Hall, 1992. Eiffel is not yet widely used, and lacks native compilers for many platforms, but it looks like a very promising language. It has many interesting features not available in most other languages such as assertions, a dynamic runtime environment, exception handling, and persistent objects.

Shlaer, Sally and Mellor, Stephen J. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, NJ: Prentice Hall (Yourdon Press), 1992. This is a technical overview of how to build complex systems using several models: state models, process models and information models. It differs considerably from other books on object-oriented design. I found it particularly interesting for its descriptions of objects using state machines and data flow diagrams. After using state machines in a recent project, I find that I now rely on them heavily when thinking about object relationships. Class-based inheritance relationships get a very brief discussion as one type of relation between objects.

Sphar, Chuck. *Object-Oriented Programming Power for THINK Pascal Programmers*. Redmond, WA: Microsoft Press, 1991. This is an older book that talks about the use of class libraries in Object Pascal, referring specifically to TCL, MacApp and several small examples built from scratch. If you're a Pascal programmer, this book could serve as a good introduction to object-oriented programming.

Springer, George and Friedman, Daniel P. *Scheme and the Art of Programming*. Cambridge, MA: MIT Press, 1989. If you're interested in making the leap from languages like Pascal and C to LISP and its variants, the easiest way would probably be to get a copy of Gambit Scheme for the Macintosh and work through this book. Learning a little bit about Scheme can give you a great deal of insight into NewtonScript.

Stroustrup, Bjarne. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994. If you've ever wanted to get inside the mind of Bjarne Stroustrup to understand why the language developed in the way it did, here's your chance.

Ungar, David, et. al. "Organizing Programs Without Classes." (This paper is available in PostScript form via anonymous ftp from self.stanford.edu). This paper discusses how SELF can build object-oriented programs in which there are no classes – objects inherit only from other objects. Some of the ideas in this paper are applicable to NewtonScript, which borrowed some of its ideas from SELF. For example, the authors show a way to change the drawing style of a graphical object in a drawing program from boxed to smooth by altering its parent link at runtime.

Waldo, Jim, editor. *The Evolution of C++: Language Design in the Marketplace of Ideas.* Cambridge, MA: MIT Press (A Usenix Association Book), 1993. This book contains a variety of historical articles on various features in C++, including two of Bjarne Stroustrup's original articles. Two of the most interesting are by Tom Cargill and Jim Waldo, presenting two sides of the debate about the inclusion of multiple inheritance in the language.

Paul R. Potts is currently between meaningful work experiences. He is available to entertain at parties by constructing inheritance chains out of thin air. To find him, look for a wild-eyed, bearded man shuffling up and down the sidewalk outside of Border's Books in Ann Arbor, Michigan, wearing a sign around his neck that says "Will build object-oriented applications for food."