NeWS Classes; an Update

Owen Densmore
Sun Microsystems
Mountain View, CA 94043
(415) 336-1798
odensmore@eng.sun.com

The use of classes by the NeWS 1.0 toolkit "Lite" was introduced at the 1986 Monterey Graphics Conference as a tentative exploration of packaging for PostScript programs residing in the NeWS Window System. The NeWS Toolkit (TNT) used by the X11/NeWS Window System has greatly extended the use of classes over its precursor. This is a report on these activities.

This paper is divided into three sections. The first is on the basic PostScript class model used in NeWS. It discusses the fundamental notions of PostScript dictionaries as a means for inheritance, objects as dictionaries, how messages are sent, sending to self and super, creation of classes and instances, and the basic class Object.

The second section continues into advanced topics, many of which are new in the X11/NeWS release. These include multiple inheritance, local variables for methods, redefining instance variables, promotion of class variables to instance variables, obsolescence, re-definition of classes, dynamic creation of methods, defaults, and much more. These topics arise from the far greater use of classes by The NeWS Toolkit than its Lite precursor.

The final section looks at applications of classes in The NeWS Toolkit; applying the advanced topics of the second section. These include automatic destruction of obsolete objects, "magic" dictionaries as instances, use of the canvas tree to form a container hierarchy, implementation of an X window manager in TNT, aids in callback management, a look-and-feel independent User Interface abstraction, and more.

1.0 The Basic Class Model

The PostScript dictionary object can be used to implement Smalltalk-80[5] classes and objects. This section presents the dictionary model for objects and the primitives needed to support basic classes in NeWS.

1.1 Introduction to PostScript Dictionaries

The PostScript language[1,2] supports a *dictionary*, an associative lookup table binding a *key* to a *value*. All name references within PostScript are resolved through the current dictionary stack; an ordered array of dictionaries. The lookup proceeds from the most recent dictionary put on the dictionary stack to the initial dictionary. The initial pair of these dictionaries are special: the systemdict contains all system operators and data, and the userdict contains data global to the current application.

The initial dictionary stack for a NeWS application looks like:

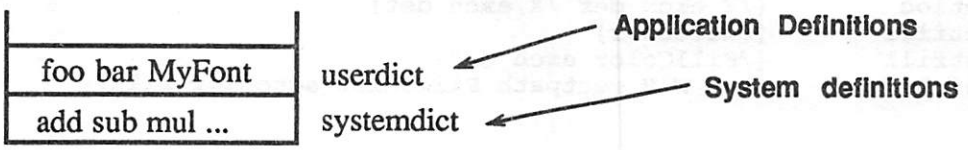


Figure 1: PostScript Dictionary Stack

The key feature of the dictionary stack for object oriented programming is that it provides for name overriding; thereby yielding inheritance.

Name overriding works as follows. Names are looked up from the outermost dict to systemdict. If a system name is found before systemdict, it is used instead. The overriding name can subsequently refer to the system dict object directly by using the **get** operator rather than looking it up by name.

Suppose we want to override the systemdict **add** operator to round down the two values being added. By placing:

```
add { % num1 num2
    floor exch floor exch      % round args down
    systemdict /add get exec   % add them via system operator
def
```

in userdict, we redefine the **add** for the current application.

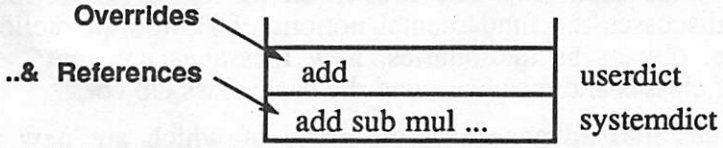


Figure 2: Using Dictionary Stack to Override Names

In object oriented programming parlance, this is an example of the "method" **add** in the "object" userdict invoking the **add** method in its "superclass" systemdict! The object userdict "inherits" the operators **floor, exch, ..** from systemdict when the look-up fails in userdict.

NeWS classes[11] generalize this notion of overriding and inheritance into a Smalltalk-80 style of object oriented programming. This is entirely implemented in roughly 500 lines of PostScript, residing in the server initialization file class.ps.

1.2 What is an Object?

PostScript Objects are an ordered collection of dictionaries treated as a single entity. These dictionaries contain all the data (Class Variables and Instance Variables) and procedures (Methods) needed by the object.

Let's look an a simple rectangle object. It is composed of two dictionaries: the **ClassRect** dictionary that contains the data and methods used by all rectangles, and the **aRectInstance** dictionary that is an instance of a single rectangle whose current location is (100,200) and whose width and height are 50 and 25. The class will contain a single variable, **FillColor**, which is global to all rectangles, and a set of methods specific to rectangles for getting and setting the rectangle location, size and color; and for painting the rectangle.

```
ClassRect:
    /SuperClasses  []
    /FillColor     <red>
    /getsize       {W H}
    /setsize       {/H exch def /W exch def}
    /getloc        {X Y}
    /setloc        {/Y exch def /X exch def}
    /getfill       {FillColor}
    /setfill       {/FillColor exch def}
    /paint         {X Y W H rectpath FillColor setcolor fill}
```

```
aRectInstance:
    /SuperClasses   [ClassRect]
    /X              100
    /Y              200
    /W              50
    /H              25
```

When **aRectInstance** is in use, the dictstack will have the four dictionaries **aRectInstance, ClassRect,** userdict and systemdict

When **getsize** is executed for this object, the names are found in these dicts:

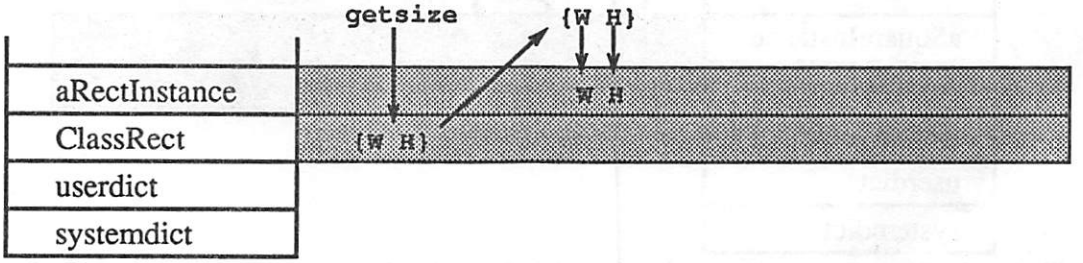


Figure 3: Class and Instance Name Lookup

The method **getsize** is found in **ClassRect.** It executes a procedure looking up the two names **W** & **H** which are found in **aRectInstance.**

The variables **X, Y, W** and **H** are called instance variables; they vary among instances of the class. The variable **FillColor** is called a class variable; it is global to all instances of the class. The procedures **getsize, setsize,** ... are called methods of the class.

Note that classes and instances are both types of objects; they can be put on the dictstack and manipulated. For example, the global class variable **FillColor** can be changed for the class by executing the method **setfill** while **ClassRect** is in use:

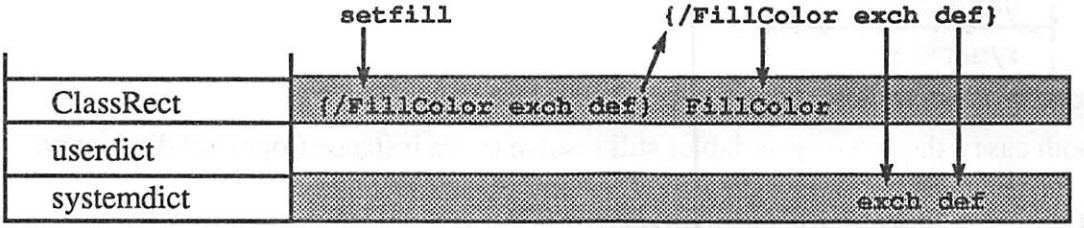


Figure 4: ClassRect Name Lookup for "setfill"

Here the object **ClassRect** is put on the stack, the method setfill is executed and is resolved to the **ClassRect** dict. The procedure executes and uses two names resolved in systemdict which change the value of the **FillColor** class variable. This will change, dynamically, the color for all rectangles immediately.

1.3 Multiple classes: SubClasses

The above example shows only a single dictionary for the class of an instance. Generally an instance has several classes associated with it, each additional class dictionary being a subclass of the initial class.

Let's consider a new class, **ClassSquare,** which adds two new methods to **ClassRect** for setting and getting the "edge" of the square.

```
ClassSquare:
    /SuperClasses   [ClassRect]
    /setedge        {dup /H exch def /W exch def}
    /getedge        {H}
```

```
aSquareInstance:
    /SuperClasses   [ClassSquare ClassRect]
    /X              100
    /Y              200
    /W              50
    /H              50
```

When **getedge** or **setedge** is executed for squares, the method name lookups resolve to the new class, **ClassSquare:**

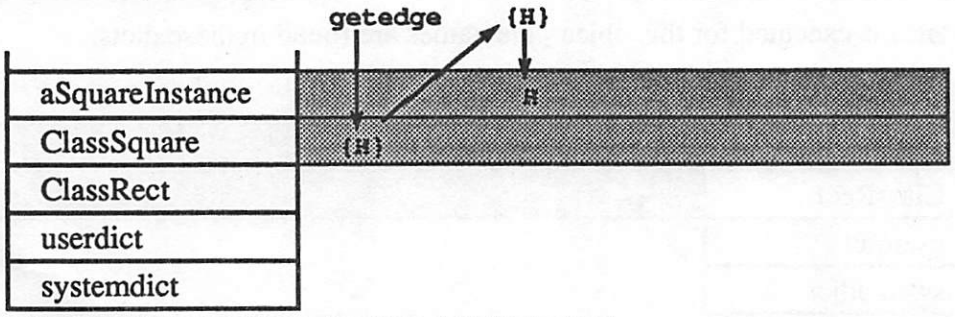


Figure 5: ClassSquare Name Lookup for "getedge"

.But when **getsize** is executed for this object, the method name lookups resolve to the initial class, **ClassRect:**

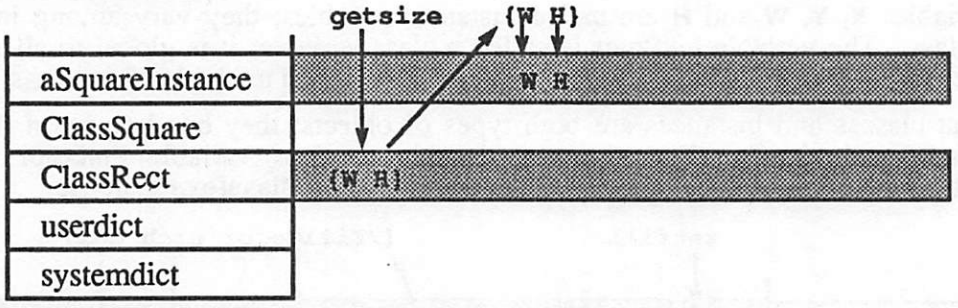


Figure 6: ClassSquare Name Lookup for "getsize"

In both cases, the instance variables still resolve to the instance (topmost) dictionary.

1.4 Sending Messages

Thus far we have not mentioned how objects are taken on and off the dictstack; and how the methods are executed. This is called sending a message to an object and is handled by the **send** primitive.

A message is an optional set of arguments, followed by a method name, followed by the object being sent to, followed by **send:**

```
arg1 .. argN /method obj send
```

The method may return results by leaving them on the operand stack.

The steps to sending a message are:

- pop off the current object's dictionaries from the dictstack if there is an object currently in effect.
- push on the dictionaries for the object being sent the message
- execute the message, optionally returning results
- pop off the object's dictionaries
- re-install the prior object's dictionaries if there was a pending send.

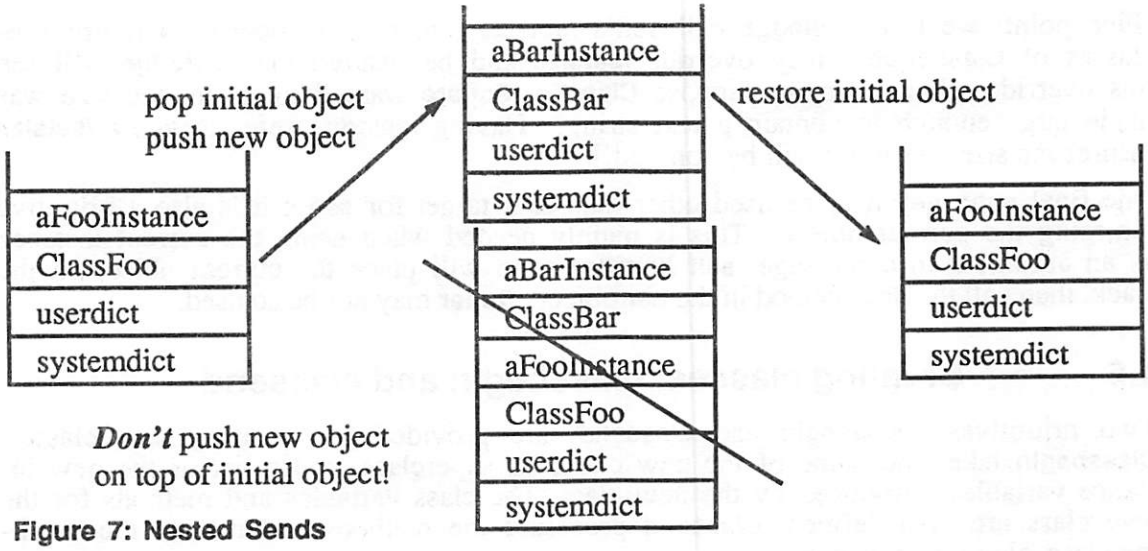


Figure 7: Nested Sends

1.5 Self and Super

Within the context of a method, Smalltalk-80 uses two pseudo variables, **self** and **super**, which may be used as targets for the **send** primitive.

The **self** pseudo variable refers to the current object. Use of **self** within a method directs send to restart name resolution at the object that caused the method to be invoked.

The **super** pseudo variable refers to superclasses of the current method's class. Use of the **super** pseudo variable tells **send** to begin looking for the method in the immediate superclass of this class.

In the following example, **ClassFoo**'s **/zot** method sends the **/foo** message to **self.** This instructs send to begin the name search for **/foo** over again at the top of the dictstack. This resolves to **ClassFooBar**'s **/foo** method. This in turn sends the **/foo** method to **super.** This instructs **send** to start the search for the **/foo** method not at the top of the dictstack, but at the immediately lower class **ClassFoo. Super,** therefore avoids cyclic references; and **self** allows a class to refer to its subclasses not yet defined

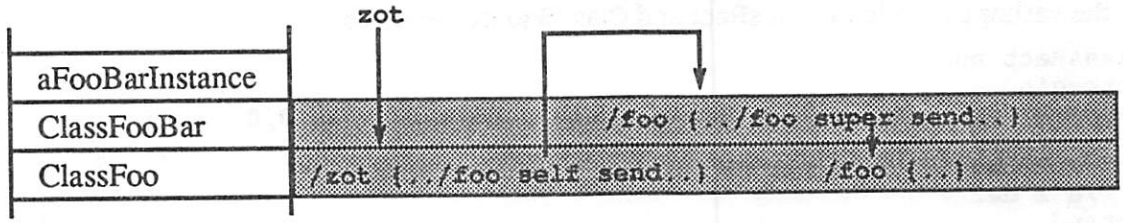


Figure 8: Self and Super Name Resolving

We'll use **self** and **super** to make two changes in the earlier **ClassSquare** example. First, we did not override **/setsize** to enforce both the height and width to be the same. This is corrected by overriding **/setsize.** Next, we change **/setedge** to call **/setsize** rather than repeating the code already in ClassRect.

```
ClassSquare:
    /SuperClasses [ClassRect]
    /setsize      {2 copy ne {<error>} {/setsize super send} ifelse}
    /setedge      {dup /setsize self send}
    /getedge      {H}
```

[Fine point: we have **/setedge** call **self**'s **/setsize** rather than **super**'s so future subclasses of **ClassSquare** may override **/setsize** and be insured that **/setedge** will see this override. For example, suppose **ClassTextSquare** wanted to insure the size was made large enough to contain a text string. Having **/setedge** refer to **self**'s **/setsize** insures the size constraint will be honored!]

One final note: **self** may be used other than as a target for **send**; it is also a primitive returning the current object. This is mainly needed when using the current instance as an argument in a message: **self /foo bar send** will place the current object on the stack, then call the **/foo** method in the **bar** object. **Super** may not be so used.

1.6 Creating classes: classbegin and classend

Two primitives, **classbegin** and **classend**, are provided for creating new classes. **Classbegin** takes the name of the new class, its superclass, and a list of the new instance variables introduced by the new class. The class variables and methods for the new class are then defined. **Classend** processes the methods and returns the resulting class, along with its name.

The list of class variables may be given either as an array or as a dictionary with initial default definitions. In the array case, the variables are initialized to **null**. For a simple class with no superclass, **null** may be used for the superclass to **classbegin**.

The use is typically:

```
/NewClass SuperClass
dictbegin
    /InsVar1 val1 def
        ...
    /InsVarN valN def
dictend
classbegin
    /ClassVar1 val1 def
        ...
    /ClassVarN valN def
    /method1 {..} def
        ...
    /methodN {..} def
classend def
```

For the earlier example of **ClassRect** and **ClassSquare** we'd use:

```
/ClassRect null
dictbegin
    /X 0 def    % initial defaults are a unit rect at 0,0
    /Y 0 def
    /W 1 def
    /H 1 def
dictend
classbegin
    /FillColor  1 0 0 rgbcolor def
    /getsize    {W H} def
    /setsize    {/H exch def /W exch def} def
    /getloc     {X Y} def
    /setloc     {/Y exch def /X exch def} def
    /getfill    {FillColor} def
    /setfill    {/FillColor exch def} def
    /paint      {X Y W H rectpath FillColor setcolor fill} def
classend def
```

```
/ClassSquare ClassRect [] % no new instance variables
classbegin
    /setsize
        {2 copy ne {pop pop (!) print}{/setsize super send} ifelse} def
    /setedge    {dup /setsize self send} def
    /getedge    {H} def
classend def
```

The processing by **classbegin** includes:

- Create the dictionary for the class
- Save the class name
- Create the SuperClasses array for the class
- Create a dictionary of instance variables that includes all the instance variables of this class and its superclasses.

Between the **classbegin .. classend** pair, the dictionary stack is set up to have all the superclasses for the class on the stack, along with the new dictionary for the class itself.

The processing by **classend** includes:

- Method compile all the class methods
- Check the **UserProfile** dictionary for overrides.
- Return the class name and dictionary

Method compilation processes each method to replace **self send** and **super send** with the appropriate PostScript fragments. In particular, **/foo self send** is converted to simply **foo**. This is because sending a message to the current object simply means executing the message with no change in the dictstack; the object is already in place!

UserProfile is a dictionary in which the user may place procedures of the same name as classes which they want to modify. The procedure is passed the class name and class dictionary to modify as desired. Thus, if we were to desire the default color for rectangles to be blue instead of red, this would be placed in the **UserProfile**:

```
/ClassRect {dup /FillColor 0 0 1 rgbcolor put} def
```

We'll return to this in the defaults discussion below.

1.7 Creating instances: /new

New instances of classes are created by sending the message **/new** to the desired class. Thus to create a new rectangle one would use: **/new ClassRect send.** The new object is typically immediately defined as a name:

```
/aRectInstance /new ClassRect send def
```

But we didn't define a **/new** method for **ClassRect**, so how does **/new** work? The answer is simple; we define a root class, class **Object**, which is used by all our other classes; and which has the **/new** method defined in it.

Class **Object** defines **/new** to:

- Allocate a dictionary object
- Install the initial instance variables
- Install the SuperClasses array for the instance
- Perform other initialization

Class **Object** factors these tasks into two other methods: **/newobject** performs all but the final initialization step which is performed by **/newinit**. Subclassers override **/newinit** to perform initialization specific to their class. Typically **/newinit** is used to process arguments used by **/new** for the given class. Suppose, for example, that **ClassFoo** needed initialization for its **/Foo** instance variable. It would use **/newinit**:

```
/newinit { % foo => -
    /newinit super send
    /Foo exch def
def
```

The earlier definition for **ClassRect** must be changed to include class **Object**:

```
/ClassRect Object
dictbegin
    /X 0 def
    ...
```

ClassRect needs no **/newinit** because it has no special initialization other than that gotten for free by defaulting the instance variables to a unit rectangle at the origin.

The instance variables **X, Y, W, H**, the class variable **Color**, and the methods **getsize, setsize**, .. are laid out as follows in their dictionaries for an instance of **ClassSquare**:

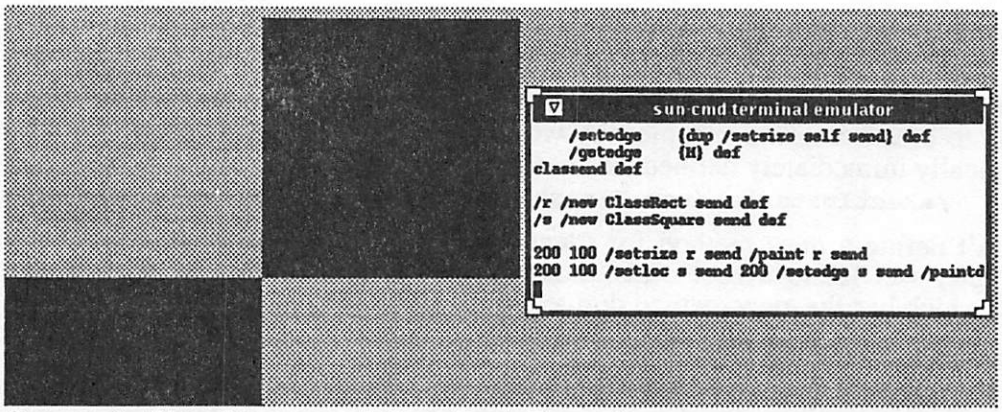
aSquare	**X Y W H**
ClassSquare	**setsize setedge getedge**
ClassRect	**FillColor getsize setsize getloc setloc getfill setfill paint**
Object	**new newobject newinit ...**

Figure 9: Name Layout in a Square Instance

To make and use a square:

```
/r /new ClassRect send def
/s /new ClassSquare send def
200 100 /setsize r send /paint r send
200 100 /setloc s send 200 /setedge s send /paint s send
```

The resulting screen looks like:



Screen 1: ClassSquare and ClassRect Example

1.8 Class Basics Review

We've now covered the fundamentals of NeWS classes:

- The PostScript dictionary stack provides a simple name overriding technique.
- This forms the basis for defining objects as an ordered set of dictionaries composed of the superclass dictionaries and an optional instance dictionary.
- The **send** primitive is used to automate the dictionary stack manipulation and method execution. A message has the form:

```
arg1 .. argN /method object send => results
```

- The pseudo variables **self** and **super** are special targets for send which refer to the current object and the superclass for the current method. **Self** may also be used as a primitive returning the current object.
- **Classbegin** and **classend** are primitives used for constructing classes. The method compilation step in **classend** resolves **self** and **super** sends.
- **/new** and **/newinit** are means for constructing new instances of classes. They are provided in a special root class, class **Object**.

Most of this was present in the NeWS 1.0 release with the Lite toolkit as presented in the earlier paper on NeWS classes [3]. [There the entire class implementation was given as a two page Appendix A!] We'll now go on to further class topics brought about by the increased use of classes in TNT.

2.0 Advanced Class Topics

The NeWS Toolkit makes far greater use of classes than its Lite precursor. [One joke has it that TNT should have been called Classic!]

The flavor of TNT is that most NeWS objects (canvases, processes, events) have a class based counterpart. In particular, all user interface objects (windows, menus, buttons, ...) are based on a **ClassCanvas** subclass. In addition, the notion of a "container hierarchy" is supported via a **ClassBag** which introduces the notion of nesting of **ClassCanvas** instances. Both support event management through the notion of activation/deactivation.

X11/NeWS itself makes far greater use of classes, even implementing the X Window Manager in subclasses of **ClassCanvas!**

To support this, several enhancements were added to the basic class primitives, and to the base class, class **Object**. The following section will enumerate these changes, subsequent sections enlarging on the purpose and use of these changes.

In the following examples, we will be executing small code fragments using the NeWS P-Shell (psh) which simply pipes its **stdin** to the NeWS server, and pipes the output from NeWS to its **stdout**.

2.1 X11/NeWS Enhancements to PostScript Classes

These changes are of two kinds: changes to the class primitives and changes to the base class, class **Object**.

The class primitives are the class creation primitives **classbegin** and **classend**, the method compiler, **send**, and miscellaneous utilities in the class implementation file class.ps.

The first major change is that **classbegin** can take an array of superclasses: supporting multiple inheritance. Other changes to classbegin are:

- Dynamic reinstallation of classes
- More flexible instance variable redefinition and protection
- Support for enumeration of classes
- SuperClasses pushed onto dictstack between **classbegin/classend**

The first major change to **classend** was to migrate much of its code up to **classbegin**, thereby allowing us to push the SuperClasses onto the dictstack between **classbegin/classend**. The other major change was to allow users to modify the newly created class via the **UserProfile** dictionary.

The method compiler was rewritten to be easily enhanced; it simply keeps a dictionary of tokens it will modify, along with the code to do the modification. One such enhancement is to make sure **self send** and **super send** work correctly when local dictionaries have been pushed onto the dictstack.

The **send** primitive was rewritten in C for performance reasons. Access to the send stack was provided. Nested sends were popped off the dictstack.

The changes to class Object include:

- **/installmethod** for creating on-the-fly methods
- **/new**: factored into **/newobject, /newinit**
- **/newmagic**: method for creating instances from existing dictionaries
- **/new**: now copies compound instance variables
- **/newdefault, /defaultclass** methods for use with abstract classes
- **/sendtopmost**: access to the send history stack
- **promote, promoted? unpromote ?promote**: utilities for promoting class variables to instance variables
- **/destroy** & **/obsolete**: methods for object cleanup.

2.2 Classbegin: Multiple Inheritance

Classbegin was changed to allow the superclass to be an array of classes. The array may be empty, indicating a root class. The resulting class is in no way special; it simply has a SuperClasses array that contains more class dictionaries than if it was a subclass of a single superclass. Thus, for NeWS, multiple inheritance conceptually differs little from simple inheritance. In fact, the initial rewrite for multiple inheritance was smaller than the single inheritance version due to fewer special cases!

The rules for the resulting SuperClasses list given the initial superclass array are:

- If ClassA precedes ClassB in **classbegin**'s superclass array, then it will proceed ClassB in the resulting SuperClasses list.
- If ClassA precedes ClassB in any of the SuperClasses lists for a class in **classbegin**'s superclass array, then it will proceed ClassB in the resulting SuperClasses list.

Two fundamental changes to **classbegin** and **classend** were required:

- The incoming array had to be converted to a SuperClasses list using the above rules
- A more sophisticated reduction of **super send** by the method compiler was needed.

The primary use of multiple inheritance is to mix-in a common capability into a set of classes that do not descend from each other. As an example, consider the following class whose only purpose is to add a text string to graphical classes:

```
/ClassLabel Object
dictbegin
    /Label      () def
dictend
classbegin
    /LabelColor 0 0 1 rgbcolor def
    /LabelX     {X 10 add} def
    /LabelY     {Y 10 add} def

    /paint      {/paint super send /paintlabel self send} def
    /paintlabel {LabelColor setcolor LabelX LabelY moveto Label show}def
    /getlabel   {Label} def
    /setlabel   {/Label exch def} def
classend def
```

Note that **ClassLabel** is a subclass of **Object** (which has no **/paint** method), but **ClassLabel**'s **/paint** calls super! **ClassLabel** would fail if sent **/paint**, but instead must be mixed in with another class which does have a **/paint** method. Although we could simply give **ClassLabel** no superclass, we decide to descend from **Object** so that some messages will work, such as **/classname** below, even though others will not.

Now let's define a button class:

```
/ClassSimpleButton [ClassLabel ClassRect]
dictbegin
    % New button instance variables
dictend
classbegin
    % New button class variables & methods
classend def
```

To see its superclasses:

```
    /superclasses ClassSimpleButton send
    [exch {/classname exch send} forall] ==
[/Object /ClassRect /ClassLabel]
```

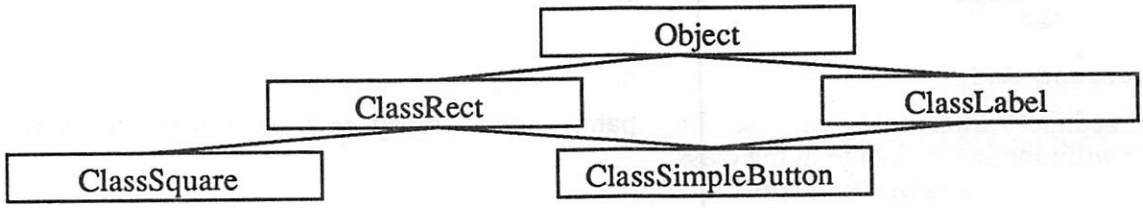
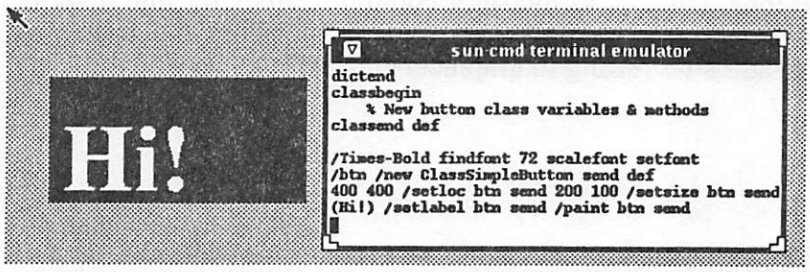


Figure 10: Multiple Inheritance

To use **ClassSimpleButton** & **ClassLabel**:

```
/Times-Bold findfont 72 scalefont setfont
/btn /new ClassSimpleButton send def
400 400 /setloc btn send 200 100 /setsize btn send
(Hi!) /setlabel btn send /paint btn send
```



Screen 2: Simple Button

[Historic note: the initial idea for multiple inheritance, as well as several (at least 7!) other changes in the X11/NeWS class implementation, was from the LispScript re-implementation of class.ps done at Schlumberger by David Singer and Rafael Bracho[9]. This project, like TNT, had far greater need for sophisticated classes. Although we were driven to modify the original Schlumberger implementation, their initial implementation was invaluable!]

2.3 Classbegin: Reusable Class Dictionaries

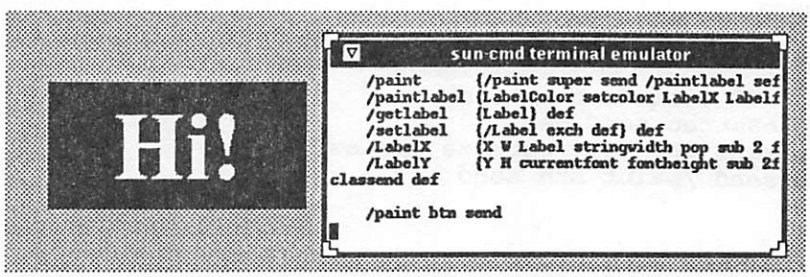
In PostScript, compound objects such as strings, arrays, and dictionaries are shared: the objects point to a piece of virtual memory. NeWS added the capability to PostScript dictionaries to grow in size. These two capabilities combine to allow classbegin to reuse a class dictionary when the class is re-defined.

This has interesting ramifications. If I rebuild a class, all its existing subclasses will see these changes. Let's rebuild the **LabelClass** defined above to center the text in its bounding box. We do this by redefining **LabelX** and **LabelY** to calculate the correct position:

```
/ClassLabel Object
    ...
    /LabelX {X W Label stringwidth pop sub 2 div add} def
    /LabelY {
        Y H currentfont fontheight sub 2 div add
        currentfont fontdescent add
    } def
    ...
classend def
```

Immediately after doing this, sending **/paint** to the already existing **/btn** in the example will inherit the change in the class:

```
/paint btn send
```



Screen 3: Simple Button with Centered Label

2.4 Classbegin: Redefinition of Instance Variables

In earlier versions of classes, clients could inadvertently reuse an instance variable name without intending to do so. It was decided to warn clients when they introduce an instance variable whose name has already been used by a superclass.

Unfortunately there was good reason for knowledgeable clients to redefine instance variables as an efficient way for their particular class to have a different initial value for that variable. A good example would be for **ClassRect** and **ClassSquare**; especially if **ClassRect** had decided to use non-square values for the height and width! Another example would be to allow **ClassSimpleButton** to have its label initially be (Button!):

```
/ClassSimpleButton [ClassLabel ClassRect]
dictbegin
    /Label (Button!) redef
dictend
classbegin
    % New button class variables & methods
classend def
```

To solve this problem, a new primitive was introduced: **redef**. When instance variables are given to **classbegin** in the dictionary form, use of **redef** will allow the variable to be redefined with no warning; and without changing the initial value of the superclass's instance variable. Using **redef** on a new name results in a warning.

Fine point: a redefined instance variable is redefined *in the new class only*, not in the class that initially defined the variable. Thus, in the above, **ClassLabel** has its **Label** still initialized to the empty string, while **ClassSimpleButton**, and its descendents, will have **Label** initialized to (Button!):

```
    /new ClassLabel send /getlabel exch send ==
()
    /new ClassSimpleButton send /getlabel exch send ==
(Button!)
```

In sum, then, we issue warnings when a name is reused as an instance variable; use of **redef** allows redefinition of an existing instance variable's initial value; and a warning will be issued when **redef** is used with a name that does not already exist.

2.5 Classbegin: SubClasses list

In NeWS 1.0, each class kept a list of the names of its subclasses. This was to allow enumeration of the class tree, and to support browsers. Class names, rather than direct references to the class objects, were used to avoid garbage collection problems. Because X11/NeWS has features to solve the garbage collection problems, **classbegin** now installs references to the objects, not simply their names.

Two utility methods of class **Object** may be used to enumerate the class hierarchy: **/superclasses** and **/subclasses**, both of which return arrays of objects.

To print out the names of the superclasses of **ClassMenu**, for example:

```
    /superclasses ClassMenu send
    [exch {/classname exch send} forall] ==
[/Object /ClassTarget /ClassCanvas /ClassSelList]
```

and to print out the names of the subclasses of **ClassCanvas**:

```
    /subclasses ClassCanvas send
    [exch {/classname exch send} forall] ==
[/ClassFramebuffer /ClassSelList /ClassControl /ClassBag /ClassXWindow]
```

2.6 Classend: UserProfile

As part of the initialization of X11/NeWS, a user provided file .startup.ps is read. This file may add entries to the **UserProfile** dictionary created just prior to reading the file. **Classend** looks to see if there is a procedure of the same name as the class. If so, it hands the class to the procedure for modification.

If I preferred my rectangles to start out green, I would place this in .startup.ps:

```
UserProfile begin
    /ClassRect { % class => class'
        0 1 0 rgbcolor /setfill 2 index send
    } def
end
```

This is part of a general defaulting scheme discussed below.

2.7 Object: /newinit

In NeWS 1.0, every class wanting to initialize new instances by executing code in the **/new** call had to override **/new** in a standard cliche:

```
/new { % <args> => instance
    /new super send begin
        <instance initialization code>
        currentdict
    end
} def
```

This, though clever, began to be very unwieldy for complex argument handling. It also was difficult to explain to new programmers: the **/new super send begin** simply happened to fake a **send** to the new instance. Lastly, it became a performance problem for classes that had many superclasses.

The solution was to recognize that **/new** really does two separate things: it sends a message to a class to create an empty instance of that class, then it sends a second message to the newly created instance to initialize itself. We achieve this by factoring **/new** into two methods: **/newobject** to do the former, and **/newinit** to do the latter:

```
/new { % <args> => instance
    /newobject self send                    % <args> instance
    {/newinit self send self} exch send % instance
} def
```

The default **/newobject** in class **Object** creates a new instance dictionary, copies the instance variables into that dictionary, and sets the SuperClasses array. **/newinit** defaults to a no-op; expecting superclasses to override it. The second line in **/new** is a clever way to send **/newinit** to the new object, and to return the new object without keeping it on the operand stack.

2.8 Object: /newmagic

NeWS extensions to PostScript are done as "magic" dictionaries similar to PostScript font dictionaries. The three primary extensions are canvases, events, and processes. Not surprisingly, there is a corresponding class object for each of these.

In NeWS 1.0 these classes had an instance variable corresponding to the NeWS magic dictionary. Thus **ClassCanvas** required a **/Canvas** instance variable.

The factoring of **/new** into **/newobject** and **/newinit**, and X11/NeWS allowing magic dicts to be extended by users (not allowed in NeWS 1.0) combine to make possible converting an existing magic dict (canvas, event, process) into an instance of an object. This is formalized by the method **/newmagic** in Object.

The **/newmagic** method converts a dictionary argument into an un-initialized instance of the class to which the message is sent. Thus ClassCanvas creates a canvas using the NeWS primitive newcanvas, then sends **/newmagic** to self to convert it into an instance of the subclass of **ClassCanvas** being created. This is done by having **ClassCanvas** supply an override for **/newobject**. Similarly, **ClassInterest** and **ClassEventMgr** convert events and processes into class instances.

To see how it works, we'll create a simple dict with one field. We'll then create a new **ClassRect** object from that dict. The resulting dict is the same as the initial dict but enhanced to be an instance of **ClassRect**.

```
    /d 10 dict def
    d /Magic (Magic) put
    [d {pop} forall] ==
[/Magic]
    /dd d /newmagic ClassRect send def
    [dd {pop} forall] ==
[/Magic /X /Y /W /H /SuperClasses]
    d dd eq ==
true
```

2.9 Object: Destruction and Obsolescence

Four new methods have been added to class **Object** to assist in managing object destruction. To better understand these, we need to review NeWS memory management.

PostScript objects are either simple or compound. Simple objects (ints, reals, bools, ..) require no additional storage, while compound objects (strings, arrays, and dictionaries) are collections of other objects and reference additional memory in PostScript's virtual memory. When compound objects are replicated by being pushed onto a stack or put into a dictionary or array, only the object, not the associated virtual memory, is replicated. This virtual memory segment contains a reference count equal to the number of references to it. When this count becomes zero, the memory is released.

Two difficulty with reference counted garbage collection arise:

- Cycles: Memory cycles may easily occur in which one compound object contains a reference to another object which in turn refers back to it. If the first reference to the object is removed, the cyclical reference prevents garbage collection.
- Managers: Other external agencies outside the applications control may need to reference a compound object. Examples are selection and keyboard focus managers. The reference they maintain can cause an object to not be garbage collected.

X11/NeWS introduces the notion of "soft" references: when the references to an object are all soft, a special **/Obsolete** event is generated. A process may listen for these, performing an appropriate action.

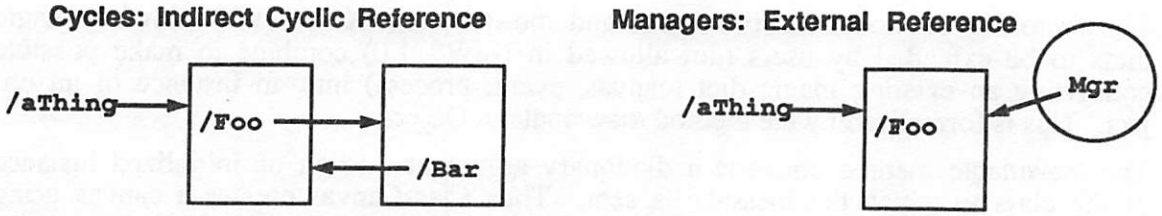


Figure 11: Cycles and Managers

To help manage instance destruction, four methods have been added to class **Object**:

- **/destroy**: Called when an object is to be no longer used. Typically this will kill processes spawned by the object, remove themselves from managers referencing them, and break cyclic references. Defaults to a no-op in **Object**.
- **/destroydependent**: When **/destroy** is sent to an instance containing references to other instances, **/destroydependent** rather than **/destroy** is sent to these other instances. If an instance contains a menu, for example, sending the menu **/destroydependent** allows the menu to not destroy itself if it is shared among other instances. This allows all **OpenLookFrames**, for example, to use a single menu. Defaults to **destroy self send** in **Object**.
- **/obsolete**: Sent to an object when all remaining references are soft. Defaults to **destroy self send** in **Object**.
- **/cleanoutclass**: Prepare a class dictionary for re-use. Typically this simply calls **undef** for each key in the class dictionary.

Finally, there is a **classdestroy** primitive added to the **classbegin classend** pair. It removes the class from any SubClasses lists that contain it, then calls **/cleanoutclass**.

2.10 Object: /installmethod

Because **UserProfile** modifications of classes required installing new methods into existing classes, and because certain instances need to override their class's methods, we decided to make method compilation generally available through a method in class **Object**.

The **/installmethod** method takes a procedure as its argument and sends it to the class for which the procedure should be methodcompiled. This resolves **self send** and **super send** correctly for that class. The resulting method is then defined in that class.

Although **/installmethod** is generally sent to a class, we also allow it to be sent to instances of a class. These "instance methods" resolve **super** to be the class of the instance, not the superclass of the instance's class. Allowing instance methods is a pragmatic way to avoid almost-empty classes that override only one method.

As an example, suppose we decided not to create a separate class for squares, but I, as a client of **ClassRect**, had to insure a particular rectangle was square. I'd do this as follows:

```
/square /new ClassRect send def
/setsize {
    2 copy ne {
        pop pop (Please make me square.\n) print
    } {
        /setsize super send
    } ifelse
} /installmethod square send
```

Resulting in this behavior:

```
    10 100 /setsize square send
Please make me square.
    45 45 /setsize square send
    [/getsize square send] ==
[45 45]
```

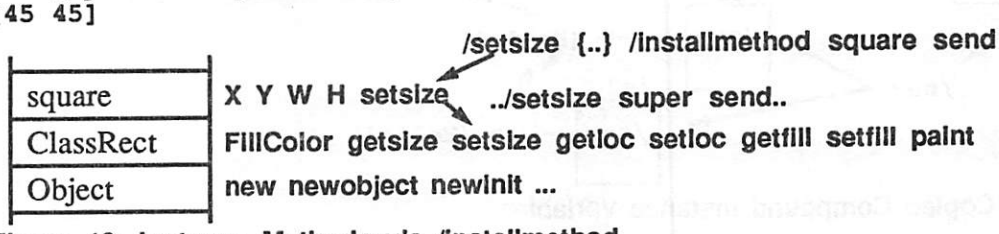


Figure 12: Instance Methods via /installmethod

2.11 Object: /new Copies Compound Instances

Recall that PostScript compound objects are shared; they return pointers to dynamically allocated memory. In NeWS 1.0, when a pre-initialized instance variable was a compound object, each new instance shared this object. In X11/NeWS, **/new** makes separate copies of these objects. This avoids having one instance modifying a compound object being propagated to all instances. The correct way to achieve this shared behavior is to use class variables.

To see this in action, notice that modifying one of two strings referring to the same data changes both strings:

```
    /s1 (Hello) def
    /s2 s1 def
    [s1 s2]==
[(Hello) (Hello)]
    s1 0 104 put % 104 = 'h'
    [s1 s2]==
[(hello) (hello)]
```

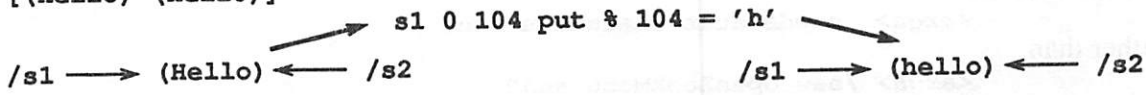


Figure 13: Shared Strings

Now make the beginnings of a string class with a string (compound) instance variable:

```
/SimpleString Object
dictbegin
    /Str (Hello!) def
dictend
classbegin
    /getstring {Str} def
    % .. & more stuff
classend def
```

The analogous tests with string instances will not modify both strings because of the copying of compound instance variables:

```
    /s1 /new SimpleString send def
    /s2 /new SimpleString send def
    [/getstring s1 send /getstring s2 send] ==
[(Hello!) (Hello!)]
    /getstring s1 send 0 104 put % 104 = 'h'
    [/getstring s1 send /getstring s2 send] ==
[(hello!) (Hello!)]
```

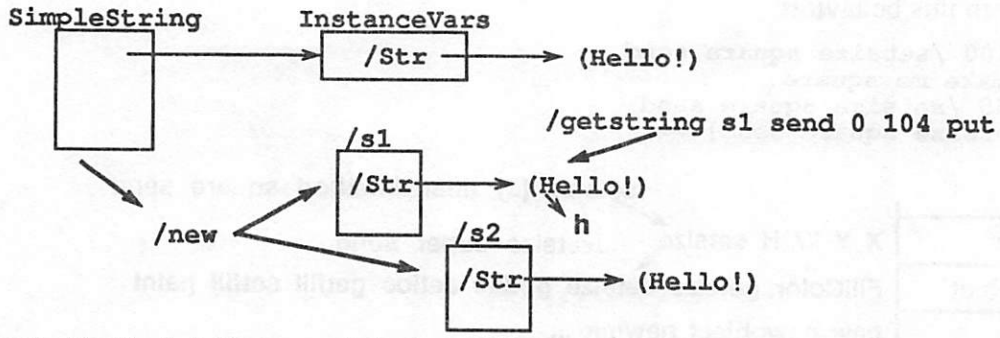


Figure 14: Copied Compound Instance Variables

2.12 Object: /newdefault & /defaultclass

The Smalltalk-80 notion of abstract superclasses is supported in NeWS. An abstract superclass is simply a class that requires one or more methods to be defined, but does not implement them; expecting a subclass to do so. When such a method is not implemented by subclasses, the **SubClassResponsibility** error is raised. An abstract superclass is therefore incomplete and thus should never receive the **/new** method.

It was decided to complement this notion with a special method, **/newdefault**, that would redirect the **/new** message to the correct subclass of the abstract superclass. It is implemented by having a **/DefaultClass** class variable for each class. This is initially **self** for all classes unless overridden. Abstract superclasses always override **/DefaultClass**.

This is used to provide Look & Feel (user interface) independence in TNT. All UI elements have an abstract superclass. **ClassMenu** is such an abstract class with **OpenLookMenu** being its default implementation. Thus clients are encouraged to create new menus via:

```
<args> /newdefault ClassMenu send
```

rather than

```
<args> /new OpenLookMenu send
```

One oddity of **/DefaultClass** is that the class cannot be defined until the subclass has been defined. This is solved by setting **/DefaultClass** to be a procedure returning the desired subclass, deferring looking for the subclass until after the abstract class is defined. Thus, in **ClassMenu** we find:

```
/DefaultClass {OpenLookMenu} def
```

There are times when a client wants to make a subclass of one of these default classes. This is done using the **/defaultclass** method. Thus, if I needed to modify **ClassMenu**'s default class, I'd use:

```
/MyMenu /defaultclass ClassMenu send
dictbegin
...
```

This is a generalization of a concept that started in NeWS 1.0 as **DefaultWindow** and **DefaultMenu** which were simply pointers to the desired default classes.

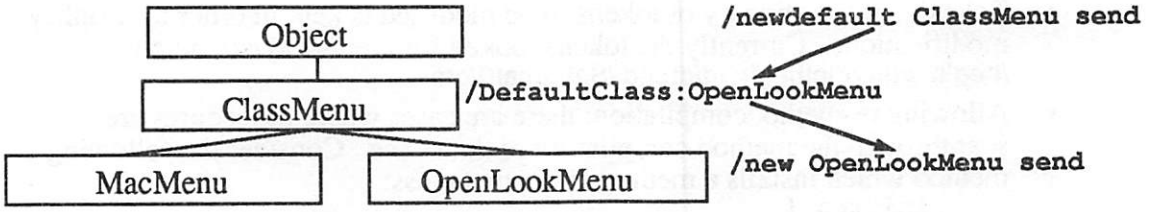


Figure 15: /newdefault and /defaultclass

2.13 Object: Promotion of Class Variables

Because instances are simply dictionaries, a class variable may be overridden by defining the same name in the instance dictionary. Consider rectangle class and instance variables:

`aRectInstance`	`X Y W H`	Instance Variables
`ClassRect`	`FillColor`	Class Variables

Now assume we **def** into **aRectInstance** the name **FillColor**. This is called *promoting* the ClassVariable **FillColor** to be an Instance Variable. Its value will override the class variable whenever referenced in methods for that one instance.

Class **Object** provides four utilities for handling promotion:

- **promote**: promote a class variable to an instance variable.
- **promoted?**: check if the variable is an instance variable.
- **?promote**: promote variable if it differs from the class version.
- **unpromote**: remove variable from instance variables.

We'll see how TNT uses these in the discussion of defaults below.

2.14 send: Change Summary

The discussion on send in the first section presented the X11/NeWS behavior. Changes since NeWS 1.0 include:

- Nested sends: NeWS 1.0 did not pop off the current send; it simply pushed the object on top of the current object.
- Access to the popped off sends is provided. This is useful when an instance wants to know what object caused it to be sent a message. [This is a generalization of the **ThisWindow** hack of NeWS 1.0]
- C implementation: For performance reasons, the **send** utility, and the support for **super send** is implemented in C.
- PostScript versions of the C implements are provided to allow overriding the C implementations. This is useful for testing and debugging.

2.15 methodcompile: Change Summary

The **methodcompile** primitive in class.ps is used by both **classend** and class **Object**'s **/installmethod** discussed above. The method compiler is almost trivial: it simply scans an array looking for **self send** and **super send**, converting them into the appropriate code fragments. Changes since NeWS 1.0 include:

- Tokenized: a dictionary of tokens to be modified is kept in order to simplify modifications. Currently the tokens looked for are: **/send /supersend /begin /end /dictbegin /dictend /SetLocalDicts**
- Allowing re-methodcompilation: there are cases where procedures are sent through the method compiler more than once. Consider the following method which installs a method in another class:

```
/initfoo {
    /mumble {...} /installmethod Foo send
} def
```

The code {...} will be multiply method compiled: once during the method compilation of **/initfoo** at the **classend** of its class definition, and once when the **/installmethod** is executed.

- Local Variables: The method compiler now notices when dictionaries are added to the dictstack within a method; generating the correct code for **self** and **super** sends for this situation. It also provides a compiler directive to override the count of these local dicts.

The local variable dictionary problem is subtle: when a method needs to define local variables, it has to add a dictionary to the dictstack. Suppose **ClassSquare** had:

```
/foo {
    10 dict begin
        /a .. def
            ...
        a /setedge self send
            ...
    end
} def
```

This results in a dictstack of:

```
<10 dict>
aSquare
ClassSquare
ClassRect
...
```

But the method compiler would usually convert the **a /setedge self send** to **a setedge.** Alas, the setedge method will cause **def** to be used to set **W** and **H** ..but now in the wrong dict: the temporary local variable dict! The solution is to count the **begin/end** pairs (and **dictbegin/dictend** pairs) to determine whether the conversion of **/foo self send** to **foo** is valid. Clearly this is heuristic at best, so there is an explicit override available: **/SetLocalDicts** is used as a directive to the method compiler to override the **begin/end** counter.

3.0 Application of Classes

In the two previous sections we have seen the basic classing system in NeWS, and have looked at advanced topics brought about by increased use of classes in X11/NeWS and The NeWS Toolkit. The rest of the paper will show how these are applied.

3.1 Defaults

Unix is a very adaptable environment. Users expect to be able to modify the behavior of their keyboard (stty) and terminal emulations (termcap) easily, and to specify default behavior of programs with "*.rc" files.

Unix window systems have tried to provide flexibility through use of files containing default settings for constant parameters such as text fonts, text sizes, and colors for foreground, background and text. Difficulties arise, however, when trying to specify changing just the text size of menus, or the background color for only the text editor. Typically every program has to anticipate each parameter the user may want to override, and check some sort of defaults database for an entry. These systems tend to be limited by their database implementation, have serious performance problems, and generally have a bad fit to the underlying window semantics.

Classes provide a simple formalism to easily solve these problems. Defaults are simply class variables. Subclasses may override these variables to yield a hierarchy, differentiating Button fonts from TextEditor fonts. Individual instances may override a class variable by promoting it to be an instance variable. Users may override a class's choice of an instance variable by use of a **UserProfile** entry. Default implementations in abstract superclasses provide a formalism for user-interface intrinsics.

Simple as this appears, it is an entirely general formalism for the defaults problem! The model is this:

- Each object in the system has an associated class.
  Example: UI objects are based on **ClassCanvas** which contains general drawing parameters: **FillColor, StrokeColor, TextFont, TextColor**, etc.
- These objects are subclassed to form more differentiated objects.
  Example: **ClassMenu** is based on **ClassCanvas.**
- Each new application defined in the system defines a basic class for itself shared among all instances of that application.
  Example: **TextEditor** is a subclass of **ClassCanvas.** It contains a **TextEditorMenu** menu that is a subclass of **ClassMenu.**
- Users override these general facilities as their classes are defined using the **UserProfile.**
  Example: I can change general **FillColors** to 50% gray, the menu **FillColor** to white, and the **TextEditMenu** to red by placing in .startup.ps:

```
UserProfile begin
    /ClassCanvas { % name class => name class
        dup /FillColor .5 .5 .5 rgbcolor put
    def
    /ClassMenu { % name class => name class
        dup /FillColor 1 1 1 rgbcolor put
    } def
    /TextEditMenu { % name class => name class
        dup /FillColor 1 0 0 rgbcolor put
    } def
 end
```

- Finally, individual instances can have their own, individual, values of the default class variable by promoting that variable to be an instance variable.
  Example: To color the currently selected window's clients yellow, execute this via a psh:

```
/selectedframes ClassFrame send {
    client exch send
    /FillColor 1 1 0 rgbcolor /promote 3 index send
    /damage exch send
} forall
```

  Note: This typically would be done by supplying the invocation of the application with an argc/argv argument. There is not currently a general argument passing technique in TNT that would automate this. On the other hand, use of the selected frames is a reasonable UI style.

A more subtle form of defaulting is changing the default implementation of an abstract super class. These classes typically act as so called "intrinsics", classes that define basic behavior but with the details to be filled in by specific implementations. Again, we use the class hierarchy to manage intrinsics and user control over them:

- Assume there are two styles of window frames available: **MacFrame**'s and **OpenLookFrame**'s The user chooses between them by:

```
UserProfile begin
    /ClassFrame { % name class => name class
        dup /DefaultClass {MacFrame} put
    def
end
```

The application insures this default is used by:

```
    /win ... /newdefault ClassFrame send def
```

3.2 Magic Dict Instances

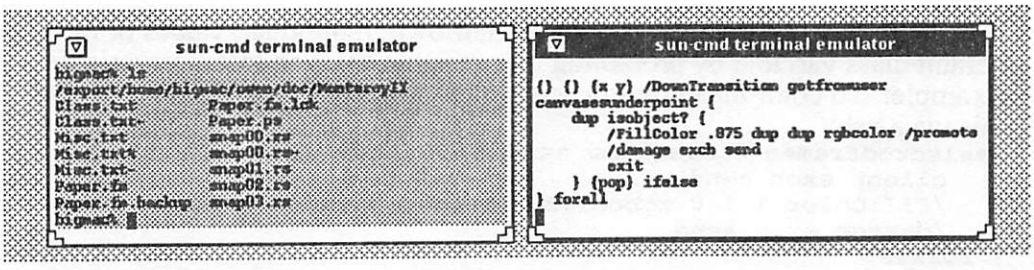
Through use of **/newmagic**, classes may choose to package instances in the dictionaries NeWS itself uses for the corresponding objects. The three most pervasive examples of this in TNT are:

- **ClassCanvas** is the common ancestor class for all UI objects. It overrides **/newobject** to use **newcanvas** to create a NeWS canvas object, turning this into the instance dictionary using **/newmagic.**
- **ClassEventMgr** forks a process which expresses interest in receiving certain events. The resulting process object is converted into the instance dictionary with **/newmagic.**
- **ClassInterest** uses the NeWS **createevent** primitive and **/newmagic** to return an instance that is both an object and a NeWS interest.

This style allows a single object to be a simple NeWS primitive and also to receive messages, and conversely, to allow a method to give **self** to a NeWS operator expecting a NeWS primitive.

As an example, the following changes the **FillColor** of the clicked-on canvas:

```
{} {} {x y} /DownTransition getfromuser
canvasesunderpoint {
    dup isobject? {
        /FillColor 0 1 0 rgbcolor /promote 3 index send
        /damage exch send
        exit
    } {pop} ifelse
} forall
```



Screen 4: Coloring Magic Dict Canvases

It works by:

- Calling the TNT **getfromuser** utility to get a mouse click, returning the x,y location.
- Calling the NeWS **canvasesunderpoint** utility to get an array of the canvases under that point.
- Using **forall** to enumerate this list looking for the first canvas that is also an object (instance of a subclass of **ClassCanvas**) via the **isobject?** utility.
- Using this canvas as an object, promoting its **FillColor** to be green, then damaging it to cause it to repaint.

3.3 Inheriting Through The Container Hierarchy

As mentioned above, all user-interface objects in TNT are subclasses of **ClassCanvas.**

Because **ClassCanvas** and its subclasses are created from NeWS canvas magic dicts, the NeWS canvas fields automatically become instance variables for instances of these classes. Several of these magic instance variables may be used to enumerate the canvas tree: **ParentCanvas, BottomCanvas, TopCanvas, CanvasAbove, CanvasBelow,** and **TopChild.**

The **ParentCanvas** field points to the canvas's immediately containing canvas in the canvas tree. We use this to form an alternative form of inheritance: inheritance through the container hierarchy rather than inheritance through the class hierarchy. Two examples of this are colors and fonts: **ClassCanvas** subclasses inherit their initial font, both family and point size, by enquiring their ParentCanvas:

```
/TextFamily{ % - => name
    /TextFamily Parent send
} def
```

Notice that the buck has to stop somewhere! The solution is to define a **ClassFrameBuffer** that is the root of the canvas tree, use **/newmagic** to bind its instance to the top of the container hierarchy, and to initialize its **TextFamily** to a standard font. Thus the framebuffer is to the container hierarchy what class Object is to the class hierarchy: the root of an inheritance tree.

3.4 Container Class: ClassBag

ClassBag provides these facilities:

- It provides named access to its children
- It provides recursive descent painting and damage repair
- It provides inherited event management
- It provides primitive layout and validation
- It manages non-canvas children; subclasses of **ClassGraphic,** which is similar to the X Gadget object.

To see these things in action, let's play with canvases and bags.

First, let's make a simple instance of **ClassCanvas.** It will be the default style: a rectangle that paints itself by stroking and filling itself using the **/FillColor** and **/StrokeColor** class variables.

```
/can framebuffer /new ClassCanvas send def
50 50 100 100 /reshape can send
/map can send
/activate can send
```

Why don't I see it initially? It's been activated, thus should paint itself when it receives damage when initially mapped. But it needs to be made opaque to enable its receiving **/Damaged** events causing it to repaint itself:

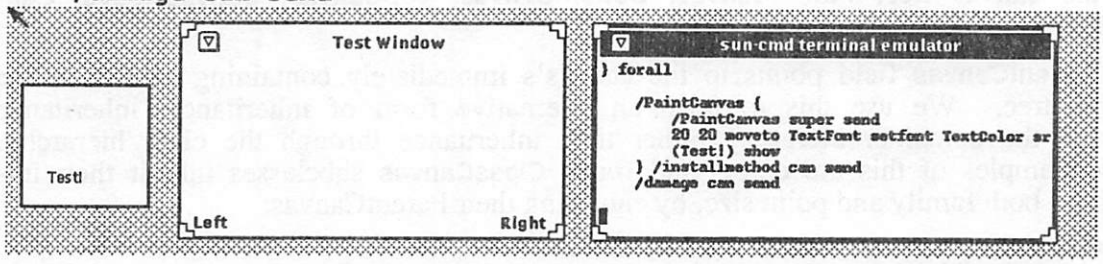
```
false /settransparent can send
```

Now move some windows over it and watch it repair itself. In fact, let's make our own window, with no client (application) installed yet:

```
/win null [] framebuffer /newdefault ClassBaseFrame send def
200 200 300 200 /reshape win send
(Test Window) /setlabel win send
(Left) (Right) /setfooter win send
/activate win send
/map win send
```

Move **win** on top of **can** to see its damage repair work. Play some. To personalize our canvas, let's replace its painting procedure:

```
/PaintCanvas {
    /PaintCanvas super send
    20 20 moveto TextFont setfont TextColor setcolor
    (Test!) show
} /installmethod can send
/damage can send
```



Screen 5: A ClassCanvas and ClassBaseFrame Pair

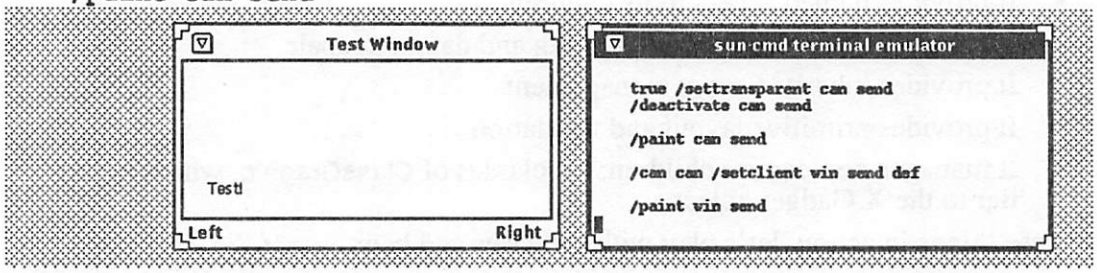
This installs the **PaintCanvas** method (called by **/paint** and the damage repair procedures) in the instance, having it call **super** to paint the stroke and fill.

OK, let's make **can** hop into **win** by making it become the client! First make **can** transparent again and deactivate it to show that it can't take care of itself anymore:

```
true /settransparent can send
/deactivate can send
```

Note that the canvas disappeared! This is because changing the state of the transparency of a canvas propagates damage to the framebuffer, which repaints itself, hiding the canvas. To see it:

```
/paint can send
```



Screen 6: Reparenting the ClassCanvas into the ClassBaseFrame

Now drag **win** over it; no damage repair. We've successfully deactivated it. To make **can** the client of win, call **/setclient** which returns the old client which will be null in this case.

```
    /can can /setclient win send def
    can ==
null
```

It won't paint automatically; you'll have to ask **win** to paint itself. Note we no longer have a handle on our canvas!

```
    /paint win send
```

Now reshape **win** and move it around; **can** really did get glued in and reactivated correctly.

We can access **can** by the name **/Client**. The method **/sendclient** sends messages to named clients, allowing us to indirectly manipulate **can** through **win**:

```
    /Times-Bold 24 null /settextparams /Client /sendclient win send
    /paint win send
```

We can pop can back onto the framebuffer by:

```
    /can null /setclient win send def
    /paint win send
    framebuffer /reparent can send
    50 50 100 100 /reshape can send
    /map can send
    /paint can send
```

We've now shown all the advertised features of bag except use of **ClassGraphic**. This is left as an exercise for the reader!

3.5 pswm: TNT Class Based X Window Manager!

The pswm X window manager[7] is implemented in PostScript using the TNT toolkit! This is done by creating two sets of classes:

- **XClientWindow** & **ClassXWindow**: **XClientWindow** is a subclass of **ClassXWindow** which is in turn a subclass of **ClassCanvas**. An instance of **ClassXWindow** is created by using **/newmagic** on an existing X window. Because X windows are canvas objects like any other NeWS canvas, we can make an instance out of X window magic dicts as we would with any other NeWS canvas.
- **XFrame**: a subclass of **ClassFrame** which is specialized for X window manager functions. There are five additional subclasses which map onto the basic frame types: **XBaseFrame, XCommandFrame, XHelpFrame, XNoticeFrame**, and **XIconFrame**.

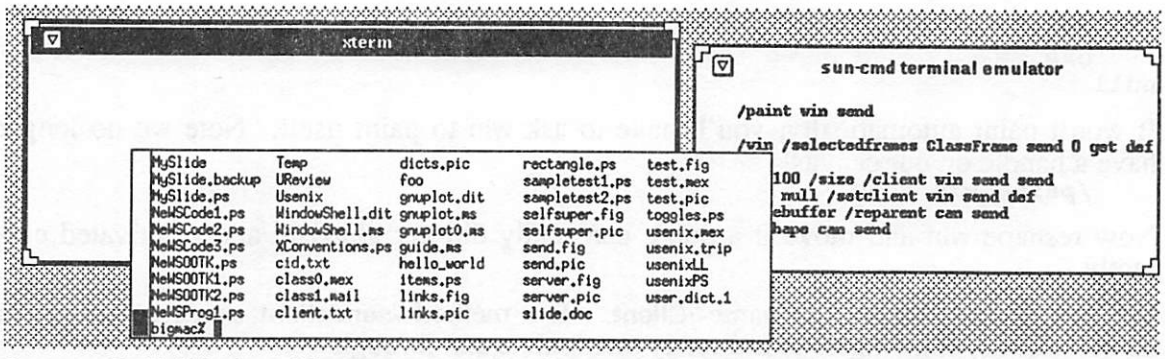
We can manipulate these clients just as we did above with simpler clients:

First, start an XTerm, select its frame, and get a reference to it. (To make it interesting, give it a scrollbar and fill it up with listings or something else.)

```
    /win /selectedframes ClassFrame send 0 get def
```

Now pop the XTerm client onto the framebuffer:

```
    100 100 /size /client win send send
    /can null /setclient win send def
    framebuffer /reparent can send
    /reshape can send
```



Screen 7: X Window Management with ClassBag and ClassCanvas

The first line is to get the location and size set up for the following **/reshape** so that it will be the same size. No repainting was required because the X clients are opaque.

Now play with the X client typing at it and scrolling. Finally put it back into its window:

```
/can can /setclient win send def
```

3.6 Obsolete Object Management

Class **Object** maintains a process which looks for all **/Obsolete** events. It then sends **/classdestroy** to classes, or **/obsolete** to instances. It does nothing for non-class objects.

To demonstrate how obsolete management works; we'll look at how canvases get handled.

First, make a simple canvas as before, installing a message to be printed if **/obsolete** is encountered:

```
/MakeCan {
    /can framebuffer /new ClassCanvas send def
    50 50 100 100 /reshape can send
    /map can send
    /activate can send
    false /settransparent can send
    /obsolete {
        (I'm going away!\n) print
        /obsolete super send
    } /installmethod can send
} def
```

Now make one, then destroy it. The **/destroy** method will remove all indirect references to the object by deactivating the canvas, thus having no process point to the canvas.

```
MakeCan
/destroy can send
```

Note that it didn't go away. The reason is that there is still a hard reference to it, **/can**. NeWS has a side effect of garbage collection of canvases: it removes them from the display. If we destroy our reference, the canvas disappears, but with no message:

```
/can null def
```

There is no message because there are no soft references outstanding, thus the object goes through standard NeWS garbage collection. Now make another canvas, and then simply remove our reference.

```
    MakeCan
    /can null def
I'm going away!
```

This results in the **/Obsolete** message. This occurs because the canvas, being still active, has an event manager which has a soft reference to the canvas. By our setting our hard reference to null, the only remaining references are soft event manager references, and the **/Obsolete** event is generated, causing **/obsolete** to be sent to the canvas.

3.7 Class Interfaces to Client Side Services

One of the more creative uses of classes in TNT has been using classes to provide an object oriented interface to services far too heavy-weight to be implemented in PostScript. This is done by reversing the typical window system client-server model: the NeWS display server invokes a client-side "daemon" process that listens for requests from NeWS using the NeWS Wire Service. The Wire Service package is a TNT library using the CPS client-server communication package for NeWS[10]

The client process is not a typical window system application; rather it is designed to listen for client requests, responding by managing client canvases. A simple PostScript class is built with methods which communicate with this service. This provides a light weight PostScript class-based interface to the service. The first instance created for the class causes the client side daemon to initialize and start up. The implementation of the client service may be any language; typically C and C++, but any language with a NeWS Wire Service interface may be used.

The first application of this idea is Jot[8], a C based text package. Jot evolved from the Ched editor described in the NeWS Book[6]. It originally was packaged as a C callable library. This proved to be clumsy for rapid prototyping within the TNT group, however; there were three machine architectures to compile for and it was awkward to re-compile programs for simple applications. The solution was to provide a second interface to the Jot library: a textserver client using the Jot library, but as a text daemon listening for requests from **ClassText**, a subclass of **ClassCanvas** which implements its methods as Wire Service calls to the textserver textserver.

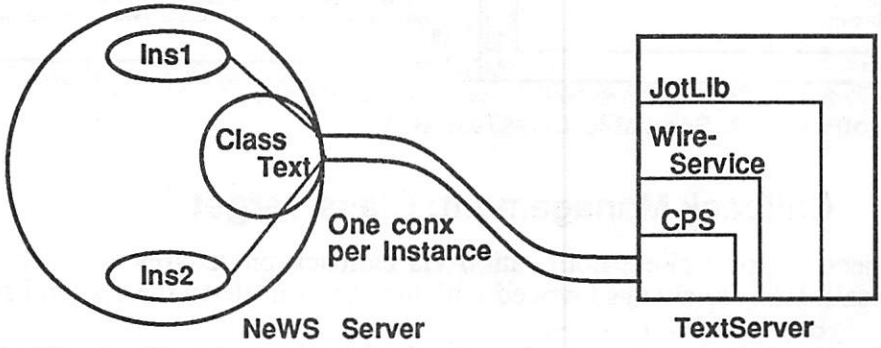
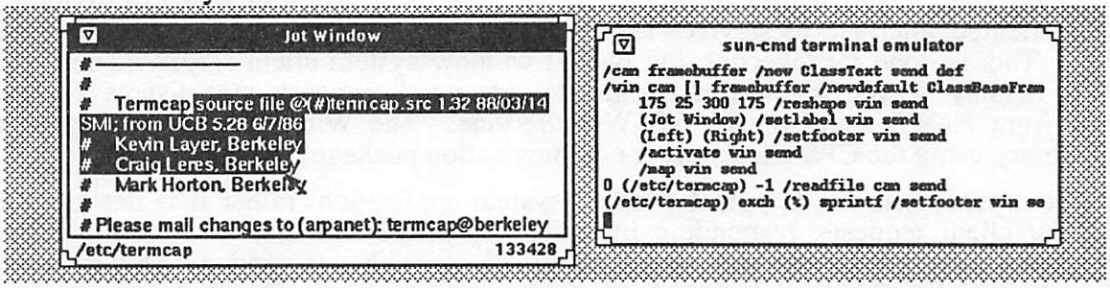


Figure 16: The Jot Model

The use of Jot from PostScript is similar to other **ClassCanvas** instances:

```
/can framebuffer /new ClassText send def
/win can [] framebuffer /newdefault ClassBaseFrame send def
    100 100 400 200 /reshape win send
    (Jot Window) /setlabel win send
    (Left) (Right) /setfooter win send
    /activate win send
    /map win send
0 (/etc/termcap) -1 /readfile can send
(/etc/termcap) exch (%) sprintf /setfooter win send
```

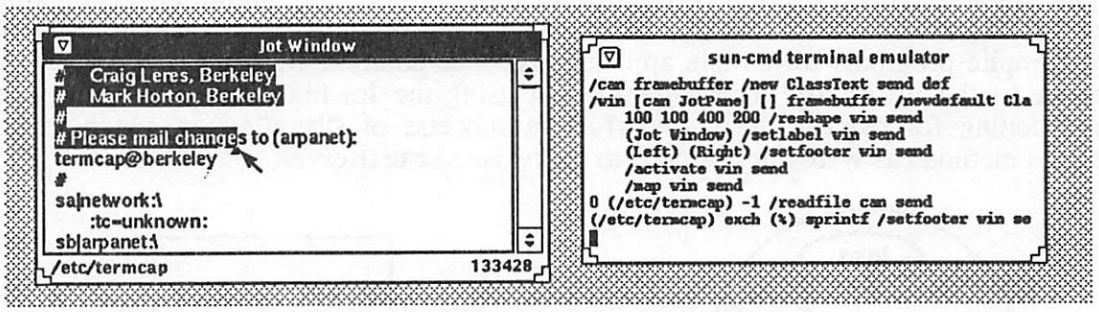
The first statement creates a new instance of a Jot canvas, **ClassText**. The next few lines create and initialize a **ClassBaseFrame** instance with the **ClassText** instance as its client. The last two lines read in /etc/termcap and set the frame's footer to be the file name and byte size.



Screen 8: ClassText, a Class based Interface to a Client Service

A scrollable pane is also provided which nests the **ClassText** instance within a canvas with scrollbars. This is achieved by simply replacing the first two lines above with

```
/can framebuffer /new ClassText send def
/win [can JotPane] [] framebuffer /newdefault ClassBaseFrame send def
```



Screen 9: JotPane - A Scrollable ClassText Bag

3.8 Callback Management: Class Target

TNT UI objects support client notification via callback procedures installed in the object. These callbacks are always invoked with the object itself on the operand stack:

```
obj callback => -.
```

Button callbacks, for example, are always called with the button object itself on the stack.

Generally the callback does not want to change the UI object, however. Instead it wants to manage another "target" object. The obvious way to allow this is to install a soft reference to that target in the UI object. Unfortunately softening the target does not completely solve the problem: the **/obsolete** message will be sent to the target, not the UI object. We'd like to avoid requiring the target knowing it is bound to a UI object. In other words, we want obsolescence of the target to send a message to the associated UI object(s).

The solution is **ClassTarget**. **ClassTarget** implements the simplest version of external reference obsolescence management. It does so by creating a separate set of interests in the obsolescence of its target clients. When a target is installed, it is softened and an **/Obsolete** interest is added to a global target event manager. The interest triggers a procedure which tells the UI object to reset its target to **null**.

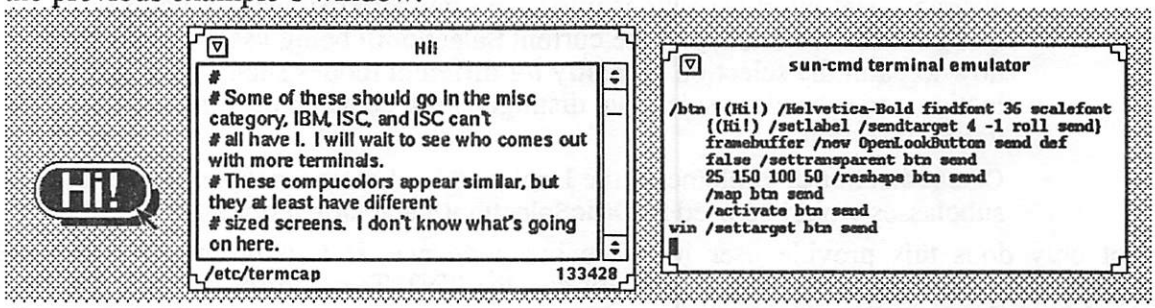
As an example of target usage, we'll place a button on the framebuffer that will cause the Jot window created above to repaint itself.

```
/btn [(Hi!) /Helvetica-Bold findfont 36 scalefont]
    {(Hi!) /setlabel /sendtarget 4 -1 roll send}
    framebuffer /new OpenLookButton send def

    false /settransparent btn send
    25 150 100 50 /reshape btn send
    /map btn send
    /activate btn send

win /settarget btn send
```

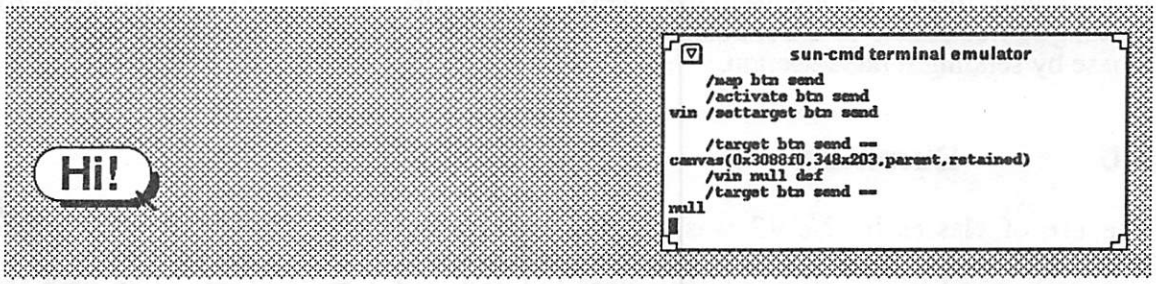
The first three lines create the button with a callback that sends **/damage** to its target. The rest installs the button on the framebuffer, and sets the button's target to be the previous example's window.



Screen 10: ClassTarget Usage

To see the obsolescence management, we'll set our **win** handle to **null**. The window disappears because it receives an **/Obsolete** event. The button's target is also reset to **null**:

```
    /target btn send ==
canvas(0x308820,305x199,parent,retained)
    /win null def
    /target btn send ==
null
```



Screen 11: ClassTarget Garbage Collection

3.9 Look & Feel Independent Selection Model

A prime goal in modern window toolkits is providing a client interface independent of the particular Look and Feel in use. Thus changing from click-to-type to follow-cursor should entail no change in the client application. Converting between left and right handed use of the mouse, and between one, two, and three buttons should also be transparent to the client.

Although complete UI independence is impossible because of potentially new semantic content of user interface, reasonable abstraction is possible in practice. An area in which TNT has been particularly successful is selections[4]. It has successfully built clients that are insensitive to changes between Mac style wipe-through selections and OpenLook style point-and-adjust selections, for example. TNT implements selections as three classes: **ClassSelection**, **ClassSelectable**, and **ClassSelectionUI**.

- **ClassSelection** describes the selected *data*. It manages the selected data content and transfer among clients. It negotiates how clients choose between data types such as ASCII text or PostScript graphics, and whether or not the transfer is by files or strings.
- **ClassSelectable** is concerned with the *rendering* of the selection data. It abstracts user interface actions into seven standard requests. It negotiates between the client and the current **SelectionUI** being used. It knows how to paint the selection correctly for different modes such as primary selection and secondary selection, distinguishing between pending delete versus insertion.
- **ClassSelectionUI** implements the Look and Feel *state machine*. Separate subclasses would be used for **MacSelectionUI** and **OpenLookSelectionUI**.

Not only does this provide user interface independence, it also promotes sharing of code among similar clients. Thus both Jot and the TNT TextControl selections are derived from the same **ClassSelectable**.

The user's preferred **SelectionUI** is activated during NeWS initialization. Clients supporting selections instantiate their subclass of **ClassSelectable**, which in turn registers them with the **SelectionUI**. Subsequent UI-client interaction in making a selection occurs via sends from the **SelectionUI** to the client's **Selectable**. A selection is initialized by the **SelectionUI** sending **/newselection** to the client's **Selectable**, which in turn sends a **/new** to its associated **Selection**. This **Selection** instance lives for the duration of the selection, and is then discarded. The **SelectionUI** will send the commands: **/selectat /adjustto /dragat /dragto /attachinsertionpoint** to the Selectable. It also will make the **/inselection?** query. The selection database is accessed by sending **/getselection** to **ClassSelection**. A **Selection** instance is inserted into the database by sending it **/setselection**.

4.0 Summary

The use of classes by NeWS was, by and large, a defensive posture; it was simply the only way we could see to successfully add packaging to PostScript! The dictionary representation for classes in instances has provided a graceful migration path for adding classes to PostScript. All of the classing system can easily be implemented in PostScript itself; it needs no interpreter enhancement.

We have been quite pleased by the results of increasing the use of classes in TNT. Use of classes in window programming is clearly a good fit. Integrating the container

hierarchy into the class hierarchy provides alternative inheritance styles. Merging the NeWS objects into instances via **/newmagic** vastly simplifies the interaction between the window platform, NeWS, and its toolkit, TNT. We even found it possible and reasonable to implement an X window manager in TNT.

The new directions we are pursuing are printing (both driving printers and application document printing), internationalization, and Jot-like client daemons for other editors. One interesting direction being investigated is an environment not unlike an object oriented HyperCard using PostScript as an interpretive shell to glue the individual components. We'd like to use the P-Shell to do for window components what the Bourne shell does for Unix commands.

The primary measure of success for NeWS classes is that we now spend far more time designing than we do implementing! The final measure of success is simply this: those using the system smile more often than not!

5.0 References

[1] Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison Wesley, (1985)

[2] Owen Densmore, *Networking NeWS* chapter 10 of *Unix Networking*; Kochan and Wood, ed., Haden Books, (1989)

[3] Owen Densmore, *Object Oriented Programming in NeWS*, Monterey USENIX Graphics Workshop (1986)

[4] Jerry Farrell, *Client-UI Separation for Selections in the NeWS Toolkit*, submitted to CHI-90, (1989)

[5] Adele Goldberg, David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley (1983)

[6] James Gosling, David Rosenthal, Michelle Arden, David Lavallee, *The NeWS Book*, Springer-Verlag (1989)

[7] Stuart Marks, *The pswm X Window Manager*, Sun Microsystems internal presentation (1989)

[8] Jonathan Payne, *Jot: Jonathan's Own Text*, private communication, (1989)

[9] Dave Singer, Rafael Bracho, *Schlumberger Rewrite of Class.ps*, private communication (1988)

[10] *NDE Technical Reference Manual*, Sun Microsystems (1989)

[11] *The X11/NeWS Reference Manual Set*, Sun Microsystems (1989)