[incr tcl] – Object-Oriented Programming in TCL

Michael J. McLennan

AT&T Bell Laboratories
1247 S. Cedar Crest Blvd.
Allentown, PA 18103
michael.mclennan@att.com

ABSTRACT

TCL was designed as an interactive command language, not a structured programming environment. Small projects flourish in TCL, thereby enticing programmers to build larger systems. But large systems quickly become unwieldy as the growing body of TCL procedures becomes more and more difficult to maintain or extend. What is needed is a way of better organizing procedures and data into packages with well-defined interfaces. **`[incr tcl]`** is a set of TCL extensions designed to provide such support. In this paper, I describe these extensions and show how they can be used to manage complexity.

INTRODUCTION

`[incr tcl]` provides object-oriented extensions to TCL, much as C++ provides object-oriented extensions to C. The emphasis of this work, however, is not to create a whiz-bang object-oriented programming environment. Rather, it is to support more structured programming practices in TCL without changing the flavor of the language. More than anything else, **`[incr tcl]`** provides a means of encapsulating related procedures together with their shared data in a local namespace that is hidden from the outside world. It encourages better programming by promoting the object-oriented "library" mindset. It also allows for code re-use through inheritance.

The fundamental construct in **`[incr tcl]`** is the class definition. Each class acts as a template for actual objects that can be created. Each object contains a unique bundle of data and provides a number of procedures or "methods" that operate on the data.

Class definitions are expressed with the following syntax:

```
class className {
    inherit superClass
    constructor args body
    destructor body
    method name args body
    public varname ?init? ?check?
    private varname ?init?
    ?TCL-commands?
}
```

The **`class`** command registers *className* as a new command in the main interpreter. Objects are then created using the *className* command as follows:

className objName ?args . . . ?

This command creates a new object with the name *objName* and registers *objName* as a new command in the main interpreter. The object is then manipulated by invoking methods with the following syntax:

objName methodName ?args . . . ?

As an example, suppose we create a class to represent a toaster. Before we begin writing the class definition, we must decide what features of the toaster we will need to use in our programming. Many toasters, for example, have a heat setting that allows the user to adjust the darkness of the toast. Toasters also have a tray that catches crumbs from the toast. This tray must be emptied from time to time, or it could become so full that the crumbs would touch the heating elements and catch fire. Notice that both of these toaster features are related: If the heat control is set toward the dark side, the toast will tend to leave more crumbs in the tray, and the tray will have to be emptied more often. Our hypothetical toaster could be encapsulated in the class definition shown in Figure 1.

The class definition can be thought of as a set of "blueprints" for a toaster, while the object (or "instance") itself is a particular toaster that has been manufactured. Methods are common to all objects in a particular class, but data is local to the object instance. Thus, each toaster that is created can have its own heat setting and its own crumb count.

Note that all of the usual TCL commands are recognized within the body of the class definition. This makes it possible to define procedures which are local to the scope of the class definition. Procedures are nearly identical to methods, except that they do not have access to public and private data. In our example, procedures are used to provide default behavior that the user of a toaster could override by simply by supplying his own TCL procedures.

We could create and destroy an instance of a toaster with the following code:

```
toaster SerialNum00001 -heat 1
set c [SerialNum00001 info class]
puts stdout "class = $c"
SerialNum00001 destroy
```

The first line of this example creates a new instance of a toaster with the name "**SerialNum00001**" and passes the remaining arguments ("**-heat 1**") to the constructor. The constructor is a special method which is automatically executed whenever a new object is created. Its purpose is to ensure that the object data has been properly initialized. Likewise, the special destructor method is invoked whenever an object is destroyed, to clean up after it. In

```
class toaster {
    constructor {config} { }
    destructor {
        if {$crumbs > 0} {
            eval $spillproc $crumbs
        }
    }
    method toast {nslices} {
        if {$nslices < 1 || $nslices > 2} {
            error "improper number of slices"
        }
        set crumbs [expr $crumbs+$heat*$nslices]
        if {$crumbs > 1273} {
            eval $fireproc
        }
        return $crumbs
    }
    method clean {} {
        set crumbs 0
    }

    public heat 3 {$heat >= 1 && $heat <= 5}
    public fireproc {defaultfire} {$fireproc != ""}
    public spillproc {defaultspill} {$spillproc != ""}

    private crumbs 0

    proc defaultfire {} {
        puts stdout "fire! fire!"
    }
    proc defaultspill {x} {
        puts stdout "$x crumbs ... what a mess!"
    }
}
```

FIGURE 1 – Definition for a class of "toaster" objects.

addition to these special methods, each class also has a number of built-in methods which provide access to class information. These methods are summarized in the Appendix to this paper.

In our example, the constructor does nothing aside from allowing us to configure public variables (more on this later). But if anyone is foolish enough to destroy a toaster without cleaning it, the destructor automatically invokes a procedure to spill all crumbs in the crumb tray. The public variable **`spillproc`** contains the name of a procedure that is responsible for actually spilling the crumbs, and the private variable **`crumbs`**, containing the current crumb count, is passed as an argument.

All class methods—constructor included—follow the same conventions for matching actual argument values to the names in the formal argument list. These conventions are identical to the ones used for ordinary **`proc`**'s, with one addition: If the last formal argument is named "**`config`**," all remaining arguments are treated as "**–***name value*" assignments, where *name* references any public variable. The value actually assigned to the local variable **`config`** within the method is the list of public variables that were altered for a particular invocation. This makes it easy to recognize changes to public variables and act accordingly. Note that any method can use **`config`** as a formal argument, although the constructor is the most natural place for it to appear.

Returning to our example, the arguments "**`–heat 1`**" are passed to the constructor and interpreted as an assignment to the public variable **`heat`**. The validation statement for this variable ("**`{$heat >= 1 && $heat <= 5}`**") makes sure that improper assignments are recognized as errors. As a rule, validation statements for public variables are automatically checked whenever the variable is altered by a **`config`** assignment. If the statement evaluates as false, an error is returned immediately—without entering the body of the method.

The second line of our example uses an object instance as a command. Specifically, the built-in **`info`** command associated with the new toaster object is invoked to query the class name. The last line of this simple example invokes another built-in method to destroy the toaster object. This action automatically invokes the destructor, which in our example spills the crumb tray.

Most of the useful work associated with a toaster is accomplished by using its methods. Methods provide a clean interface to an object by providing ways to manipulate its data without giving direct access to the data itself. We recall that in our toaster class definition, we defined a method **`toast`** that takes a number of slices as its argument. This method performs the action of toasting bread. In our view of toasters, the only important consequence is that toasting adds crumbs to the crumb tray; we use a simple formula that multiples the heat setting by the number of bread slices, and adds this many crumbs to the tray. Whenever the user of a toaster wants to clean it, he simple invokes the **`clean`** method, and the number of crumbs is reset to zero.

This example, foolish as it may seem, illustrates a simple solution for many of the problems facing TCL programmers. Many programs have clumps of related procedures that share the same data. Typically, this sharing is accomplished through global variables, and related procedures are kept together in source files. The object-oriented paradigm helps to organize this kind of programming better by explicitly grouping procedures and associated data into a class definition. It keeps private data hidden from the global namespace, and provides well-defined access mechanisms for public variables.

Object-oriented programming also provides a mechanism for code re-use through inheritance. Suppose that we wanted to design a slightly better toaster that rang an alarm whenever the crumb tray was getting full. This new toaster would be largely identical to the original model, with the addition of a monitoring device. Using inheritance, we might implement this new toaster as shown in Figure 2.

The first statement ("**`inherit toaster`**") effectively copies the definition of class toaster, causing class **`smartToaster`** to inherit all public and private variables and all of the methods except the constructor and destructor, which are local to a class definition and cannot be inherited. Any method or variable in an inherited class can be referenced explicitly using a "*class*::*method*" or "*class*::*variable*" syntax. Similarly, global commands can be referenced explicitly using the "::*globalCommand*" syntax. Whenever the "::" or "*class*::" qualifier is missing, the method is sought first in the current class scope, then in its superclass, and so on up the inheritance chain and out into the global scope. Within the methods of class toaster, therefore, the name "**toast**" refers to "**toaster::toast**," but within the methods of class smartToaster, "**toast**" refers to "**smartToaster::toast**." Note that since the command resolution includes the global scope, widget commands can be invoked transparently within a class, but have their effects at the global scope.

Object-oriented programming buffs may recognized that to have a command act as a "virtual" method within some class scope, the method name should be prefaced by the object name: *e.g.*, "**$this toast**." This forces the command to be executed in the external interpreter, which always resets scope to the most specific class and begins searching upward in the inheritance hierarchy for the appropriate method.

In this example, again, our constructor does nothing aside from handling assignments to public parameters. The destructor, however, should still perform the **spill** function implemented in class toaster. To account for this, we must define a new destructor that executes the old destructor method; the old method is referenced explicitly using **toaster::destructor**.

A new definition for the **toast** method is provided to perform the usual toasting operation via **toaster::toast**, then automatically check the crumb level and signal when it is full. Again, we use a public variable **alarmproc** to represent the name of a callback procedure that will actually ring the alarm.

If we were to instanciate a particular toaster in this class, we might query it for hierarchy information:

```
smartToaster SerialNum00002 -heat 5

SerialNum00002 info class
# returns: "smartToaster"
```

```
class smartToaster {
    inherit toaster

    constructor {config} { }
    destructor { toaster::destructor }

    method toast {nslices} {
        toaster::toast $nslices
        if {$crumbs > 1200} {
            eval $alarmproc $this
        }
        return $crumbs
    }

    public alarmproc {defaultalarm} {$alarmproc != ""}

    proc defaultalarm {name} {
        puts stdout "warning: clean toaster $name to avoid a fire"
    }
}
```

FIGURE 2 – Definition for a class of "smartToaster" objects.

```
SerialNum00002 info lineage
# returns: "smartToaster toaster"

SerialNum00002 isa toaster
# returns: 1

SerialNum00002 isa blender
# returns: 0
```

Without inheritance, the much of the code needed to implement the basic toaster would have to be duplicated for each new toaster class. This makes a program larger and more difficult to maintain. If a bug were found in the original code, the fix would have to be propagated to all of the clones. Another alternative is to develop one grand "supertoaster" that can account for all possible toaster behaviors; but this leads to complex code that is also difficult to maintain. Inheritance provides a cleaner solution, allowing for code to be shared among classes.

APPLICATIONS

One application area that this paradigm addresses particularly well is the construction of "mega-widgets." It is often useful to package a collection of primitive widgets together as a new widget—using buttons and entries and a listbox to create a file browser, for example. Until now, it has been difficult to do this and make the result look like a widget, and not a collection of procedures. Using **[incr tcl]**, however, new widget classes can be defined entirely in TCL code, and the resulting widget objects behave like normal widgets.

I have created a class **attColorEditor**, for example, that packages a canvas, a scale and an entry widget together to form a color editor, shown in Figure 3, with a Hue-Saturation-Value (HSV) color model. Any color name typed into the entry widget is automatically displayed in terms of its hue, saturation and value components. Hue and saturation are represented by a position within the circle drawn on the canvas; value is shown directly by the scale widget. As the user adjusts these components by clicking on the canvas or adjusting the scale, the entry is updated to display the current color value in "**#rrggbb**" format. The entry widget is also configured to support drag&drop[1], so the user can transport a color value to other parts of his application.

A new **attColorEditor** widget is created like any other Tk widget:

```
attColorEditor .editor -size 2i \
  -borderwidth 3
```

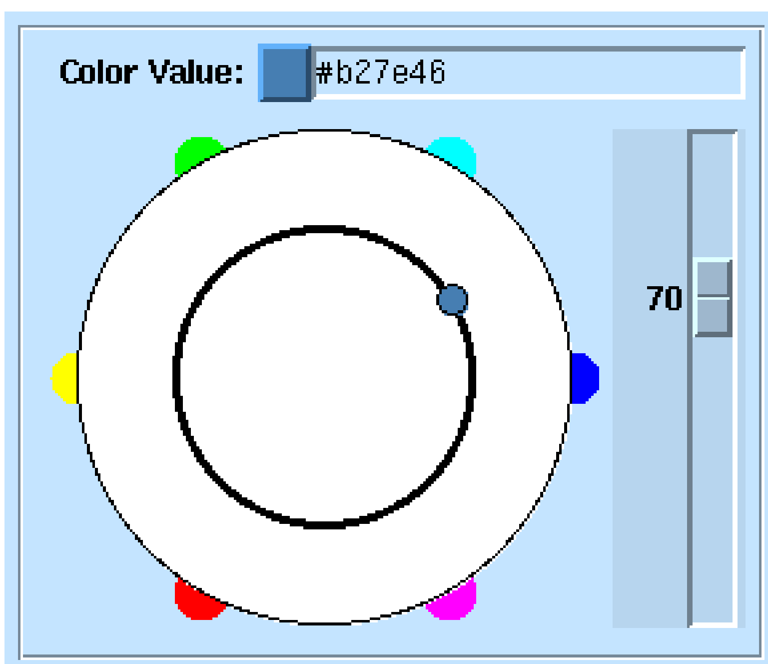

FIGURE 3 – **attColorEditor** mega-widget implemented using **[incr tcl]**.

The args "**-size 2i -borderwidth 3**" are passed to the constructor and treated as assignments to the public variables **size** and **borderwidth**. The class also includes a **config** method which can be used to change widget attributes (*i.e.*, public variables), an **install** method which loads a color value, and a **get** method which returns the current color value. Note that although any number of **attColorEditor** objects can be created, each will have its own private variables containing the H, S and V color components.

Inheritance is an added advantage, allowing mega-widgets to be specialized with new features. I have created a class **attListBox**, for example, that displays a list of items in a text widget, and automatically manages its own scrollbar. When the list is longer than the display area, the scrollbar is packed into the widget frame; when the list is shorter, it is removed. I then derived the class **attSelectBox**, which inherits the display capability from **attListBox**, and adds facilities for selecting items in the list. I then derived the class **attFilteredSelectBox**, which inherits the selection capability from **attSelectBox**, and adds a filter entry that restricts the display to items matching a string pattern. A simple application illustrating this family of widgets is shown in Figure 4.

With no C code and very little TCL code, I have created an array of useful widgets. **[incr tcl]** supports a clean separation between the main application and the special-purpose code that drives these widgets. I believe that these same principles can be applied in a variety of different contexts, allowing programmers to encapsulate many pieces that join together in their application.

CONCLUSIONS

The object-oriented paradigm has become popular because it supports the construction of complex applications. **[incr tcl]** represents a simple set of object-oriented extensions designed to help programmers write applications that are easier to maintain. Instead of relying on a single global namespace, programmers can package related procedures together with their shared data into a class definition. These procedures provide a well-defined interface for manipulating the data that is otherwise hidden from the outside world. Classes can inherit characteristics from one another, allowing for code reuse through specialization. Each class has transparent access to its own local namespace, but can also access the namespace of inherited classes and the global namespace, when needed.

As it stands, **[incr tcl]** supports only single inheritance. That is, a class can inherit characteristics from a single class and its ancestors. With just a little more effort, it could be modified to support multiple inheritance. The current work, however, is simply the first step in exploring a paradigm which programmers could use to write better applications with TCL.

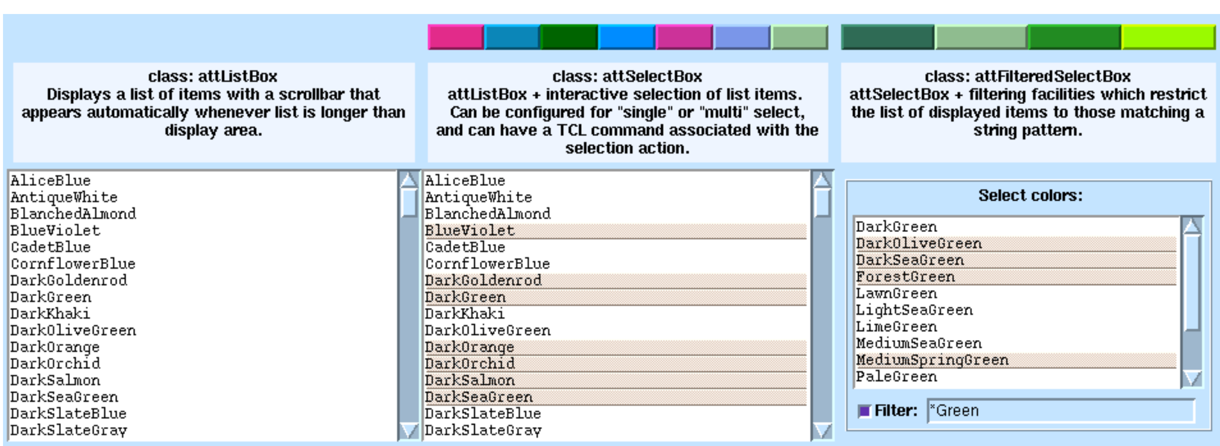

FIGURE 4 – Inheritance is used to build a family of listbox mega-widgets.

REFERENCES

[1] The **drag&drop** command supports drag-and-drop operations for Tk widgets. It is available for anonymous ftp from the archive site on **harbor.ecn.purdue.edu**, among the contributed extensions in **/pub/tcl/extensions**.

APPENDIX

Following is a summary of the commands used in **[incr tcl]**.

`class` *className definition*

Provides the definition for a class named *className*. If *className* is already defined, then the class definition is ignored; this allows for "multiple includes." *className* becomes a command in the interpreter, handling the creation of new object instances. The class definition is evaluated as a series of TCL statements that configure the interpreter managing a particular class scope. In addition to the usual commands, the following class definition commands are recognized:

`inherit` *className*

Declares *className* as a superclass, causing the current class to inherit its characteristics.

`constructor` *args body*

Declares the argument list and body used for the constructor. If defined, the constructor is automatically invoked whenever a new object instance is created. A constructor definition cannot be inherited.

`destructor` *body*

Declares the body used for the destructor. If defined, the destructor is automatically invoked whenever an object is destroyed. A destructor definition cannot be inherited.

`method` *name args body*

Declares a method called *name* with an argument list *args* and a body of TCL statements. A method is similar to the more familiar **proc**, except that it has access to public/private variables. Within the class scope, a method can be invoked just as a **proc** would be invoked—simply by using its name. A method name can also be explicitly scoped using the syntax "*class*`::`*method*." Commands in the external interpreter can be explicitly scoped using the syntax "`::`*globalCommand*." In the external interpreter, the method name must be prefaced by an object name.

`public` *varName ?init? ?check?*

Declares a public variable named *varName*. If the optional *init* is specified, it is used as the initial value for the public variable when a new object is created. If the optional *check* is specified, it is tested whenever a public variable is accessed via the **config** argument. If the check expression evaluates to zero, the assignment is treated as an error.

`private` *varName ?init?*

Declares a private variable named *varName*. If the optional *init* is specified, it is used as the initial value for the private variable when a new object is created. All objects have a built-in private variable named "**this**" which is initialized to the instance name for the object.

className objName ?args. . . ?

Creates a new object in class *className* with name *objName*. Remaining arguments are passed to the constructor. The object is manipulated using its *objName* as a command, as shown below:

objName method ?args. . . ?

Invokes a method named *method* to operate on the specified object. Remaining arguments are passed to the method. The method name can be "**constructor**," "**destructor**," any method name appearing in the class definition, or any of the following built-in methods:

objName `info` *?option? ?args. . . ?*

Returns information related to the class definition, or information concerning the interpreter that implements the class scope. The option parameter includes the options recognized by the usual TCL **info** command, as well as the following things:

objName `info class`

Returns the most-specific class name for a particular object instance.

objName `info lineage`

Returns a list of class names in order from most to least specific. This list represents the derivation history for a particular object, and is formed starting with the most specific class and following the chain of **`inherit`** statements upward through the hierarchy of class definitions.

objName `info methods` *?methodName?*

Returns a list containing the definitions of all methods definined in a class. Each list contains three elements: the method name, the argument list, and the body. If the optional *methodName* is specified and that method name is recognized, then a list is returned containing two elements: the argument list and the body. If the *methodName* is not recognized, an empty string is returned.

objName `info publics` *?varName?*

Returns a list containing the definitions of all public variables defined in a class. Each list contains four elements: the variable name, its inital value, its validation statement, and its current value. If the optional *varName* is specified and that variable name is recognized, then a list is returned containing three elements: the initial value, the validation statement, and the current value. If the *varName* is not recognized, an empty string is returned.

objName `info privates` *?varName?*

Returns a list containing the definitions of all private variables defined in a class. Each list contains three elements: the variable name, its inital value, and its current value. If the optional *varName* is specified and that variable name is recognized, then a list is returned containing two elements: the initial value and the current value. If the *varName* is not recognized, an empty string is returned.

objName `isa` *className*

Returns non-zero if the given *className* can be found in an object's lineage, and zero otherwise.

objName `destroy`

Invokes the destructor associated with an object and frees the object data. After this command, *objName* is no longer recognized as a command in the main interpreter.