Object-Oriented Programming with [incr Tcl]
Building Mega-Widgets with [incr Tk]

Michael J. McLennan
Bell Labs Innovations for Lucent Technologies
1247 S. Cedar Crest Blvd.
Allentown, PA 18104
mmclennan@lucent.com

ABSTRACT

Applications with short development cycles have the best chance for success in today's marketplace. Tcl/Tk provides an interactive development environment for building Graphical User Interface (GUI) applications with incredible speed. Tcl/Tk applications look like they were constructed with the Motif toolkit, but they can be written in a fraction of the time. This is due, in part, to the high-level programming interface that the Tcl language provides. It is also due to the interpretive nature of the Tcl language; changes made to a Tcl/Tk application can be seen immediately, without waiting for the usual compile/link/run cycle. Developers can prototype new ideas, review them with customers, and deliver a finished product within a span of several weeks. The finished product will run on all of the major platforms: Unix, PC Windows, and Macintosh.

But the Tcl language was not designed to support large programming projects. When Tcl/Tk scripts grow larger than a thousand lines, the code complexity can be difficult to manage. [INCR TCL] extends the Tcl language to support object-oriented programming. This allows developers to write high-level building blocks that are more easily assembled into a finished application. The resulting code has more encapsulation, and is easier to maintain and extend. [INCR TCL] is patterned after C++, so for many developers, it is easy to learn.

This memo contains two chapters that will appear in a book published by O'Reilly and Associates. It provides an overview of [INCR TCL], and shows how it can be used to support Tcl/Tk applications. It also describes a special library of base classes called [INCR TK], which can be used to build high-level user interface components called "mega-widgets".

In this Chapter:

- *Objects and Classes*
- *Inheritance*
- *Namespaces*
- *Interactive Development*
- *Autoloading*
- *Adding C code to [INCR TCL] Classes*

1 Object-Oriented Programming with [incr Tcl]

Tcl/Tk applications come together with astounding speed. You can write a simple file browser in an afternoon, or a card game like Solitaire within a week. But as applications get larger, Tcl code becomes more difficult to understand and maintain. You get lost in the mass of procedures and global variables that make up your program. It is hard to create data structures, and even harder to make reusable libraries.

[INCR TCL] extends the Tcl language to support object-oriented programming. It wasn't created as an academic exercise, nor to be buzzword-compatible with the latest trend. It was created to solve real problems, so that Tcl could be used to build large applications.

[INCR TCL] is fully backward-compatible with normal Tcl, so it will run all of your existing Tcl/Tk programs. It simply adds some extra commands which let you create and manipulate objects.

It extends the Tcl language in the same way that C++ extends the base language C. It borrows some familiar concepts from C++,† so many developers find it easy to learn. But while it resembles C++, it is written to be consistent with the Tcl language. This is reflected in its name, which you can pronounce as "increment tickle" or "inker tickle." This is the Tcl way of saying "Tcl++".

† Stanley B. Lippman, *C++ Primer* (2nd edition), Addison-Wesley, 1991; and Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.

This chapter shows how [INCR TCL] can be used to solve common programming problems. As an example, it shows how a tree data structure can be created and used to build a file browser. Along the way, it illustrates many important concepts of object-oriented programming, including encapsulation, inheritance, and composition.

Objects and Classes

I won't go on for pages about object-oriented programming. You have probably read about it in other contexts, and there are some really good texts† that explain it well. But the basic idea is that you create *objects* as building blocks for your application. If you are building a kitchen, for example, you might need objects like toasters, blenders and can openers. If you are building a large kitchen, you might have many different toasters, but they all have the same characteristics. They all belong to the same *class*, in this case a class called `Toaster`.

Each object has some data associated with it. A toaster might have a certain heat setting and a crumb tray that collects the crumbs that fall off each time it toasts bread. Each toaster has its *own* heat setting and its *own* crumb count, so each `Toaster` object has its own variables to represent these things. In object speak, these variables are called *instance variables* or *data members*. You can use these instead of global variables to represent your data.

You tell an object to do something using special procedures called *methods* or *member functions*. For example, a `Toaster` object might have a method called `toast` that you use to toast bread, and another method called `clean` that you use to clean out the crumb tray. Methods let you define a few strictly limited ways to access the data in a class, which helps you prevent many errors.

Everything that you need to know about an object is described in its *class definition*. The class definition lists the instance variables that hold an object's data and the methods that are used to manipulate the object. It acts like a blueprint for creating objects. Objects themselves are often called *instances* of the class that they belong to.

Variables and Methods

Let's see how objects work in a real-life example. Suppose you want to use the Tk canvas widget to build a file browser. It might look something like the one

† For example: Grady Booch, *Object-Oriented Design*, Benjamin/Cummings, 1991; and Timothy Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991.

shown in Figure 1-1. Each entry would have an icon next to the file name to indicate whether the file is an ordinary file, a directory, or an executable program. Aligning each icon with its file name is ordinarily a lot of work, but you can make your job much simpler if you create an object to represent each icon and its associated file name. When you need to add an entry to the file browser, you simply create a new object with an icon and a text string, and tell it to draw itself on the canvas.

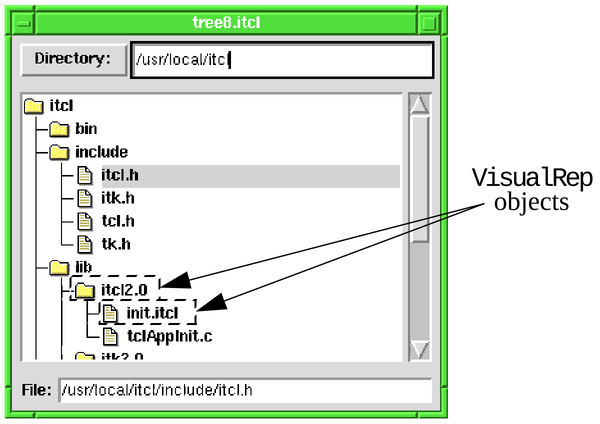


Figure 1-1 Using VisualRep objects to build a file browser.

We will create a class `VisualRep` to characterize these objects. The class definition is contained in the file *itcl/tree/visrep.itcl* on the CD-ROM that accompanies this book, and it appears in Example 1-1.

Example 1-1 The class definition for VisualRep objects.

```
image create photo default -file default.gif

class VisualRep {
    variable canvas
    variable icon
    variable title

    constructor {cwin ival tval} {
        set canvas $cwin
        set icon $ival
        set title $tval
    }
    destructor {
        erase
    }

    method draw {x y} {
        erase
        $canvas create image $x $y -image $icon -anchor c -tags $this
        set x1 [expr $x + [image width $icon]/2 + 4]
```

Example 1-1 The class definition for VisualRep objects.

```
        $canvas create text $x1 $y -text $title -anchor w -tags $this
    }
    method erase {} {
        $canvas delete $this
    }
}
```

All of the [INCR TCL] keywords are shown above in bold type. You use the `class` command to define a new class of objects. Inside the class definition is a series of statements that define the instance variables and the methods for objects that belong to this class. In this example, each `VisualRep` object has three variables: `canvas`, `icon` and `title`. The `canvas` variable contains the name of the canvas widget that will display the object. The `icon` variable contains the name of a Tk image used as the icon. And the `title` variable contains a text string that is displayed next to the icon. Each object also has a built-in variable named `this`, which you don't have to declare. It is automatically defined, and it contains the name of the object.

Each `VisualRep` object responds to the two methods listed in the class definition. You can ask the object to `draw` itself at an (`x`,`y`) coordinate on the canvas, and the icon will be centered on this coordinate. You can also ask the object to `erase` itself. Notice that all of the canvas items created in the `draw` method are tagged with the name of the object, taken from the built-in `this` variable. This makes it easy to erase the object later by deleting all canvas items tagged with the object name.

The `constructor` and `destructor` are special methods that are called automatically when an object is created and destroyed. We'll talk more about these later.

The methods and variables in one class are completely separate from those in another. You could create a `Book` class with a `title` variable, or a `Chalkboard` class with `draw` and `erase` methods. Since these members belong to different classes, they will not interfere with our `VisualRep` class. It is always obvious which methods and variables you can use if you think about which object you are manipulating. Because classes keep everything separate, you don't have to worry so much about name collisions, and you can use simpler names in [INCR TCL] code than you would in ordinary Tcl code.

Methods look a lot like an ordinary Tcl procedures. Each method has a name, a Tcl-style argument list, and a body. But unlike procedures, methods automatically have access to the variables defined in the class. In the `draw` method, we talk to the canvas widget using the name stored in the `canvas` variable. We access the icon using `$icon`, and the title string using `$title`. There is no need to declare these variables with anything like the Tcl `global` statement. They have been declared once and for all in the class definition.

The same thing holds true for methods. Within one method, we can treat the other methods as ordinary commands. In the destructor, for example, we call the `erase` method simply by using the command `erase`. If effect, we are telling this object (whichever one is being destroyed) to erase itself. In the code outside of a class, we have to be more specific. We have to tell a particular object to erase itself.

Having defined the class `VisualRep`, we can create an object like this:

```
VisualRep vr1 .canv default "Display this text"
```

The first argument (`vr1`) is the name of the new object. The remaining arguments (`.canv default "Display this text"`) are passed along to the constructor to initialize the object. This might look familiar. It is precisely how you would create a Tk widget:

```
button .b -background red -text "Alert"
```

Here, the first argument (`.b`) is the name of the new widget, and the remaining arguments (`-background red -text "Alert"`) are used to configure the widget. This similarity is no accident. [INCR TCL] was designed to follow the Tk paradigm. Objects can even have configuration options just like the Tk widgets. We'll see this later, but for now, we'll stick with simple examples.

Once an object has been created, you can manipulate it using its methods. You start by saying which object you want to manipulate. You use the object name as a command, with the method name as an operation and the method arguments as additional parameters. For example, you could tell the object `vr1` to draw itself like this:

```
vr1 draw 25 37
```

or to erase itself from the canvas like this:

```
vr1 erase
```

Again, this might look familiar. It is precisely how you would use a Tk widget. You might tell a button to configure itself like this:

```
.b configure -background blue -foreground white
```

or to flash itself like this:

```
.b flash
```

Putting all of this together, we can use `VisualRep` objects to create the drawing shown in Figure 1-2.

We need to create five `VisualRep` objects for this drawing. The first object has a directory folder icon and the message "`[incr Tcl] has:`". The remaining objects have file icons and various message strings. We can create these objects

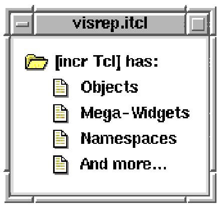


Figure 1-2 Simple drawing composed of VisualRep objects.

and tell each one to draw itself on the canvas using the handful of code in Example 1-2.

Example 1-2 Code used to produce Figure 1-2.

```
canvas .canv -width 150 -height 120 -background white
pack .canv

image create photo dir1 -file dir1.gif
image create photo file -file file.gif

VisualRep title .canv dir1 "\[incr Tcl\] has:"
title draw 20 20

VisualRep bullet1 .canv file "Objects"
bullet1 draw 40 40

VisualRep bullet2 .canv file "Mega-Widgets"
bullet2 draw 40 60

VisualRep bullet3 .canv file "Namespaces"
bullet3 draw 40 80

VisualRep bullet4 .canv file "And more..."
bullet4 draw 40 100
```

Constructors and Destructors

Let’s take a moment to see what happens when an object is created. The following command:

```
VisualRep bullet1 .canv file "Objects"
```

creates an object named “`bullet1`” in class `VisualRep`. It starts by allocating the variables contained within the object. For a `VisualRep` object, this includes the variables `canvas`, `icon`, and `title`, as well as the built-in `this` variable. If the class has a `constructor` method, it is automatically called with the remaining arguments passed as parameters to it. The constructor can set internal variables, open files, create other objects, or do anything else needed to initialize an object. If an error is encountered within the constructor, it will abort, and the object will not be created.

Like any other method, the constructor has a Tcl-style argument list. You can have required arguments and optional arguments with default values. You can even use the Tcl `args` argument to handle variable-length argument lists. But whatever arguments you specify for the constructor, you must supply those arguments whenever you create an object. In class `VisualRep`, the constructor takes three values: a canvas, an icon image, and a title string. These are all required arguments, so you must supply all three values whenever you create a `VisualRep` object. The constructor shown in Example 1-1 simply stores the three values in the instance variables so they will be available later when the object needs to draw itself.

The constructor is optional. If you don't need one, you can leave it out of the class definition. This is like having a constructor with a null argument list and a null body. When you create an object, you won't supply any additional parameters, and you won't do anything special to initialize the object.

The `destructor` method is also optional. If it is defined, it is automatically called when an object is destroyed, to free any resources that are no longer needed. An object like `bullet1` is destroyed using the "`delete object`" command like this:

```
delete object bullet1
```

This command can take multiple arguments representing objects to be deleted. It is not possible to pass arguments to the destructor, so as you can see in Example 1-1, the destructor is defined without an argument list.

Instance variables are deleted automatically, but any other resources associated with the object should be explicitly freed. If a file is opened in the constructor, it should be closed in the destructor. If an image is created in the constructor, it should be deleted in the destructor. As a result, the destructor usually looks like the inverse of the constructor. If an error is encountered while executing the destructor, the "`delete object`" command is aborted, and the object remains alive.

For the `VisualRep` class, the destructor uses the `erase` method to erase the object from its canvas. Whenever a `VisualRep` object is deleted, it disappears.

Pointers

Each object must have a unique name. When we use the object name as a command, there is no question about which object we are talking to. In effect, the object name in [INCR TCL] is like the memory address of an object in C++. It uniquely identifies the object.

We can create a “pointer” to an object by saving its name in a variable. For example, if we think of the objects created in Example 1-2, we could say:

```
set x "bullet1"
$x erase
```

The variable x contains the name “`bullet1`”, but it could just as easily have the name “`bullet2`” or “`title`”. Whatever object it refers to, we use the name `$x` as a command, telling that object to erase itself.

We could even tell all of the objects to erase themselves like this:

```
foreach obj {title bullet1 bullet2 bullet3 bullet4} {
    $obj erase
}
```

One object can point to another simply by having an instance variable that stores the name of the other object. Suppose you want to create a tree data structure. In ordinary Tcl, this is extremely difficult, but with [INCR TCL], you simply create an object to represent each node of the tree. Each node has a variable `parent` that contains the name of the parent node, and a variable `children`, that contains a list of names for the child nodes. The class definition for a `Tree` node is contained in the file *itcl/tree/tree1.itcl*, and it appears in Example 1-3.

Example 1-3 The class definition for a simple Tree data structure.

```
class Tree {
    variable parent ""
    variable children ""

    method add {obj} {
        $obj parent $this
        lappend children $obj
    }
    method clear {} {
        if {$children != ""} {
            eval delete object $children
        }
        set children ""
    }
    method parent {pobj} {
        set parent $pobj
    }

    method contents {} {
        return $children
    }
}
```

Notice that when we declared the `parent` and `children` variables, we included an extra `""` value. This value is used to initialize these variables when an object is first created, before calling the constructor. It is optional. If a variable does not have an initializer, it will still get created, but it will be undefined until the constructor or some other method sets its value. In this example, we do not have a constructor, so we are careful to include initializers for both of the instance variables.

The `Tree` class has four methods: The `add` method adds another `Tree` object as a child node. The `parent` method stores the name of a parent `Tree` object. The `contents` method returns a list of immediate children, and is used to traverse the tree. The `clear` method destroys all children under the current node.

Notice that in the `clear` method, we used the Tcl `eval` command. This lets us delete all of the children in one shot. The `eval` command flattens the list `$children` into a series of separate object names, and the `delete object` command deletes them. If we had forgotten the `eval` command, the `delete object` command would have misinterpreted the value `$children` as one long object name, and it would have generated an error.

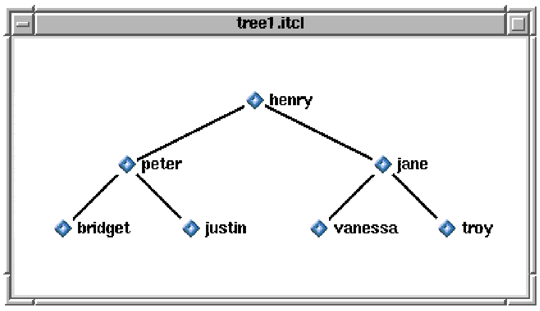


Figure 1-3 Diagram of a family tree.

We can create a series of `Tree` objects to represent any tree information that exists as a hierarchy. Consider the tree shown in Figure 1-3. We can create the root object "`henry`" like this:

```
Tree henry
```

This allocates memory for the object and initializes its `parent` and `children` variables to the null string. If effect, it has no parent and no children. Since there is no constructor for this class, construction is over at this point, and the object is ready to use.

We can add children to this node by creating them:

```
Tree peter
Tree jane
```

and by adding them in:

```
henry add peter
henry add jane
```

Each of these calls to the add method triggers a series of other statements. We could draw the flow of execution as shown in Figure 1-4. Each object is drawn with a piece broken off so that you can see the parent and children variables hidden inside of it. When we call "henry add peter", we jump into the context of the henry object (meaning that we have access to its variables), and we execute the body of the add method. The first statement tells peter that its parent is now henry. We jump into the context of the peter object, execute its parent method, and store the name henry into its parent variable. We then return to henry and continue on with its add method. We append peter to the list of henry's children, and the add operation is complete. Now henry knows that peter is a child, and peter knows that henry is its parent.

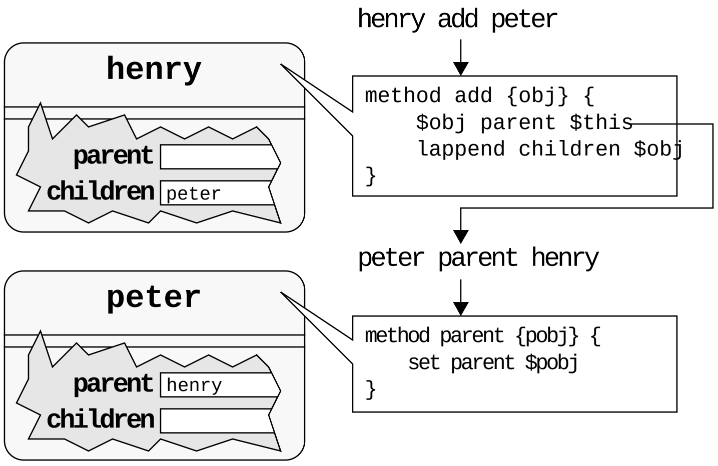


Figure 1-4 Execution can flow from one object context to another.

This simple example shows the real strength of [INCR TCL]: *encapsulation*. The variables inside each object are completely protected from the outside world. You cannot set them directly. You can only call methods, which provide a controlled interface to the underlying data. If you decide next week to rewrite this class, you can change the names of these variables or you can eliminate them entirely. You will have to fix the methods in the class, but you won't have to fix any other code. As long as you don't change how the methods are used, the programs that rely on this class will remain intact.

We can create the rest of the tree shown in Figure 1-3 as follows:

```
peter add [Tree bridget]
peter add [Tree justin]
```

```
jane add [Tree vanessa]
jane add [Tree troy]
```

We have shortened things a bit. The `Tree` command returns the name of each new `Tree` object. We capture the name with square brackets and pass it directly to the `add` method.

Generating Object Names

If you are creating a lot of objects, you may not want to think of a name for each one. Sometimes you don't care what the name is, as long as it is unique. Remember, each object must have a unique name to identify it. [INCR TCL] will generate a name for you if `#auto` is included as all or part of the object name. For example, we could add 10 more children to the `jane` node like this:

```
for {set i 0} {$i < 10} {incr i} {
    jane add [Tree #auto]
}
```

Each time an object is created, [INCR TCL] replaces `#auto` with an automatically generated name like `tree17`. If you use a name like "x**#auto**y", you will get a name like "x**tree17**y". The `#auto` part is composed of the class name (starting with a lower-case letter) and a unique number.

If we use the `Tree` class together with `VisualRep`, we can write a procedure to draw any tree on a canvas widget. We simply traverse the tree, and at each node, we create a `VisualRep` object and tell it to draw itself on the canvas. Of course, we also draw some lines on the canvas connecting each parent to its children. We will be creating a lot of `VisualRep` objects, so having automatically generated names will come in handy. A complete code example is in the file *itcl/tree/tree1.itcl*, but the drawing part appears in Example 1-4.

Example 1-4 A recursive procedure draws the tree onto a canvas widget.

```
proc draw_node {canvas obj x y width} {
    set kids [$obj contents]
    if {[llength $kids] == 1} {
        set x0 $x
        set delx 0
    } else {
        set x0 [expr $x-0.5*$width]
        set delx [expr 1.0*$width/([llength $kids]-1)]
    }
    set y0 [expr $y+50]

    foreach o $kids {
        $canvas create line $x $y $x0 $y0 -width 2
        draw_node $canvas $o $x0 $y0 [expr 0.5*$delx]

        set x0 [expr $x0+$delx]
    }
    set visual [VisualRep #auto $canvas default $obj]
    $visual draw $x $y
```

Example 1-4 A recursive procedure draws the tree onto a canvas widget.

```
}

canvas .canv -width 400 -height 200 -background white
pack .canv

draw_node .canv henry 190 50 200
```

We create the canvas and pack it, and then we call `draw_node` to draw the tree starting at node `henry`. Inside `draw_node`, we use the `contents` method to get a list of children for the current node. If there is only one child, we draw it directly below the current node. Otherwise, we divide up the available screen width and place the children starting at the x-coordinate `$x0`, with `$delx` pixels between them. We draw a line down to each child's position, and we draw the child by calling `draw_node` recursively. This will draw not only the child, but all of the children below it as well. We finish up by creating a `VisualRep` for the current node. The `default` argument says to use the default (diamond) icon, and the `$obj` argument sets the title string to the object name. We need to tell this `VisualRep` to draw itself on the canvas, so we capture its automatically generated name in the `visual` variable, and we use this as a pointer to the object.

A Real Application

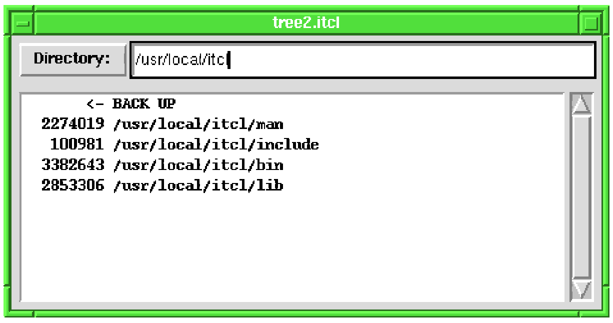
We can use our `Tree` class to build a real application, like a file browser that helps the user find wasted disk space. The Unix `du` utility reports the disk usage for a series of directories, given a starting point in the file system. Its output is a long list of sizes and directory names that looks like this:

```
    $ du -b /usr/local/itcl
    29928    /usr/local/itcl/lib/tcl7.4
    ...
    36343    /usr/local/itcl/man/man1
    812848   /usr/local/itcl/man/man3
    1416632  /usr/local/itcl/man/mann
    2274019  /usr/local/itcl/man
    11648898 /usr/local/itcl
```

The `-b` option says that directory sizes should be reported in bytes.

It is much easier to understand this output if we present it hierarchically, as shown in Figure 1-5. If we are looking at the */usr/local/itcl* directory, for example, we can see that it has four subdirectories, and of these, *bin* is the biggest. We could double-click on this directory to see a listing of its contents, or double-click on *BACK UP* to move back to the parent directory.

We can use a tree to organize the output from the `du` command. Each node of the tree would represent one directory. It would have a parent node for its



```
        <- BACK UP
 2274019 /usr/local/itcl/man
  100981 /usr/local/itcl/include
 3382643 /usr/local/itcl/bin
 2853306 /usr/local/itcl/lib
```

Figure 1-5 A hierarchical browser for the "du" utility.

parent directory and a list of child nodes for its subdirectories. The simple `Tree` class shown in Example 1-3 will handle this, but each node must also store the name and the size of the directory that it represents.

We can modify the Tree class to keep track of a name and a value for each node as shown in Example 1-5.

Example 1-5 Tree class updated to store name/value pairs.

```
class Tree {
    variable name ""
    variable value ""
    variable parent ""
    variable children ""

    constructor {n v} {
        set name $n
        set value $v
    }
    destructor {
        clear
    }

    method add {obj} {
        $obj parent $this
        lappend children $obj
    }
    method clear {} {
        if {$children != ""} {
            eval delete object $children
        }
        set children ""
    }
    method parent {pobj} {
        set parent $pobj
    }

    method get {{option -value}} {
        switch -- $option {
            -name   { return $name }
            -value  { return $value }
            -parent { return $parent }
        }
```

Example 1-5 Tree class updated to store name/value pairs.

```
        error "bad option \"$option\""
    }
    method contents {} {
        return $children
    }
}
```

We simply add `name` and `value` variables to the class. We also define a constructor, so that the name and the value are set when each object is created. These are required arguments, so when we create a `Tree` node, the command must look something like this:

```
    Tree henry /usr/local/itcl 8619141
```

Actually, the name and value strings could be anything, but in this example, we are using `name` to store the directory name, and `value` to store the directory size.

We have also added a destructor to the `Tree` so that when any node is destroyed, it clears its list of children. This causes the children to be destroyed, and their destructors cause their children to be destroyed, and so on. So destroying any node causes an entire sub-tree to be recursively destroyed.

If we are moving up and down the tree and we reach a certain node, we will probably want to find out its name and its value. Remember, variables like `name` and `value` are kept hidden within an object. We can't access them directly. We can tell the object to do something only by calling one of its methods. In this case, we invent a method called `get` that will give us access to the necessary information. If we have a `Tree` node called `henry`, we might use its `get` method like this:

```
    puts "directory: [henry get -name]"
    puts "     size: [henry get -value]"
```

The `get` method itself is defined in Example 1-5. Its argument list looks a little strange, but it is the standard Tcl syntax for an optional argument. The outer set of braces represents the argument list, and the inner set represents one argument: its name is `option`, and its default value (if it is not specified) is "`-value`". So if we simply want the value, we can call the method without any arguments, like this:

```
    puts "     size: [henry get]"
```

The `get` method merely looks at its `option` flag and returns the appropriate information. We use a Tcl `switch` command to handle the various cases. Since the `option` flag will start with a "`-`", we are careful to include the "`--`" argument in the `switch` command. This tells the switch that the very next argument is the string to match against, not an option for the `switch` command itself.

With a new and improved `Tree` class in hand, we return to building a browser for the Unix `du` utility. If you are not used to working with tree data structures, this code may seem complicated. But keep in mind that it is the example itself—not [INCR TCL]—that adds the complexity. If you don't believe me, try solving this same problem without [INCR TCL]!

We create a procedure called `get_usage` to load the disk usage information for any directory. This is shown in Example 1-6.

Example 1-6 Disk usage information is stored in a tree.

```
set root ""
proc get_usage {dir} {
    global root

    if {$root != ""} {
        delete object $root
    }
    set parentDir [file dirname $dir]
    set root [Tree #auto $parentDir ""]
    set hiers($parentDir) $root

    set info [split [exec du -b $dir] \n]
    set last [expr [llength $info]-1]

    for {set i $last} {$i >= 0} {incr i -1} {
        set line [lindex $info $i]

        if {[scan $line {%d %s} size name] == 2} {
            set hiers($name) [Tree #auto $name $size]

            set parentDir [file dirname $name]
            set parent $hiers($parentDir)
            $parent add $hiers($name)
        }
    }
    return $root
}
```

We simply pass it the name of a directory, and it runs the `du` program and creates a tree to store its output. We use the Tcl `exec` command to execute the `du` program, and we split its output into a list of lines. We traverse backward through this list, starting at the root directory, and working our way downward through the file hierarchy because the `du` program puts the parent directories after their children in its output. We `scan` each line to pick out the directory name and size, ignoring any lines have the wrong format. We create a new `Tree` object to represent each directory. We don't really care about the name of the `Tree` object itself, and we don't want to make up names like "henry" and "jane", so we use `#auto` to get automatically generated names. Once each `Tree` node has been created, we add it into the node for its parent directory.

Finding the node for the parent directory is a little tricky. We can use the Tcl "`file dirname`" command to get the name of the parent directory, but we must figure out what `Tree` object represents this directory. We could scan through the entire tree looking for it, but that would be horribly slow. Instead, we create a lookup table using an array called hiers that maps a directory name to its corresponding Tree object. As we create each object, we are careful to store it in this array so it can be found later when its children are created. Figure 1-6 shows the array and how the values relate to the directory structure we started with.

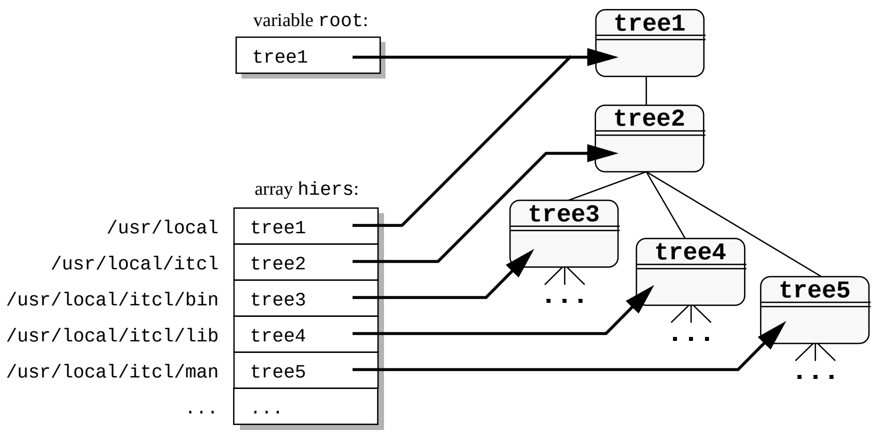


Figure 1-6 Finding directories in a tree of disk usage information.

Since we traverse backward through the du output, parent Tree nodes will always be created and entered into the hiers array before their child nodes. The only exception is the parent for the very first node. It will not appear in the output from du, so we have to create it ourselves to get everything started. We call this the *root node*, and we save its name in a global variable called root. The next time we call get_usage, we can destroy the old tree simply by destroying the root node, and then start a new tree by creating a new root node.

We can put all of this together in an application like the one shown in Figure 1-5. A complete example appears in the file *itcl/tree/tree2.itcl*, so I will not show all of the code here. But it works something like this. When the user types a directory name at the top of this application, we call the procedure get_usage to execute du and build a tree containing the output. We then call another procedure show_usage to display the root object in a listbox. The code for show_usage appears in Example 1-7.

We start by clearing the listbox and clearing any elements that might have been selected. If this node has a parent, we add the *BACK UP* element at the top of the listbox. Double-clicking on this element will invoke show_usage for the parent directory, so you can move back up in the hierarchy. We use the

Example 1-7 The contents of any Tree node can be displayed in a listbox.

```
proc show_usage {obj} {
    global root lbox

    catch {unset lbox}
    .display.lbox delete 0 end
    .display.lbox selection clear 0 end

    set counter 0

    if {[$obj get -parent] != ""} {
        .display.lbox insert end "       <- BACK UP"
        set lbox($counter) [$obj get -parent]
        incr counter
    }

    foreach kid [$obj contents] {
        set name [$kid get -name]
        set size [$kid get -value]
        .display.lbox insert end [format "%9d %-50s" $size $name]
        set lbox($counter) $kid
        incr counter
    }
}
```

`contents` method to scan through the list of child nodes, and for each of these nodes, we add an element showing the directory size and its name. Double-clicking on any of these elements will invoke `show_usage` for their node, so you can move down in the hierarchy. We use a constant-width font for the listbox, and we format each line with the Tcl `format` command, to make sure that size and name fields align properly as two columns.

Notice that as we create each element, we are careful to build an array called `lbox` which maps the element number to a `Tree` node. Later on when we get a double-click, we can use this array to figure out which `Tree` node to show. We simply add a binding to the listbox like this:

```
bind .display.lbox <Double-ButtonPress-1> {
    set index [.display.lbox nearest %y]
    show_usage $lbox($index)
    break
}
```

When the double-click occurs, the `%y` field is replaced with the y-coordinate of the mouse pointer, and the listbox `nearest` operation returns the number of the element nearest this position. We convert this to the corresponding `Tree` object using the `lbox` array, and then use `show_usage` to display the information for that node. Normally, the double-click would also be handled as another ordinary button press event, but we are careful to avoid this by breaking out of any further event processing.

Without the `Tree` class, this application would have been considerably more difficult to write. [INCR TCL] solves the problem by providing a way to create new data structures. Data structures are encapsulated with a well-defined set of methods to manipulate them. This naturally supports the creation of libraries. A generic component like the Tree class can be written once, and reused again and again in many different applications.

Interface versus Implementation

As classes get more complicated, and as method bodies get longer, the class definition becomes more difficult to read. Finding important information, like the method names and argument lists, is like looking for a needle in a haystack of [INCR TCL] code. But a method body does not have to be included with the method declaration. Class definitions are much easier to read if the bodies are defined elsewhere, using the body command. For example, our Tree class can be rewritten as shown in Example 1-8.

Example 1-8 Separating the Tree class interface from its implementation.

```
class Tree {
    variable name ""
    variable value ""
    variable parent ""
    variable children ""

    constructor {n v} {
        set name $n
        set value $v
    }
    destructor {
        clear
    }

    method add {obj}
    method clear {}
    method parent {pobj}
    method get {{option -value}}
    method contents {}
}

body Tree::add {obj} {
    $obj parent $this
    lappend children $obj
}
body Tree::clear {} {
    if {$children != ""} {
        eval delete object $children
    }
    set children ""
}
body Tree::parent {pobj} {
    set parent $pobj
}

body Tree::get {{option -value}} {
    switch -- $option {
        -name   { return $name }
        -value  { return $value }
        -parent { return $parent }
```

Example 1-8 Separating the Tree class interface from its implementation.

```
    }
    error "bad option \"$option\""
}
body Tree::contents {} {
    return $children
}
```

Since the body commands appear outside of the class definition, we cannot use simple method names like add. Remember, we could have other classes that also have an add method. Outside of the class, we must use a full name like Tree::add to identify the method. A class name followed by "::" characters is called a *scope qualifier*. You can add this to any method name or variable name to clear up ambiguities.

The class definition establishes once and for all what methods are available, and how they are used. Whatever arguments you give when you declare a method, you must use the same arguments later when you define the method body. For example, when we declared the Tree::add method, we said that it takes one argument named obj. Later, when we defined the body, we used the same argument list. When we declared the Tree::contents method, we gave it a null argument list. Again, when we defined the body, we repeated the null argument list. If you make a mistake and the argument lists do not match, the body command will report the error.

It turns out that the argument lists don't have to match letter for letter, but they must match in meaning. The argument names can change, but the argument lists must have the same number of required arguments, and all optional arguments must have the same default values. For example, when we declared the Tree::get method, we said that it has one argument named option with a default value "-value". When we define the body we must still have one argument with a default value "-value", but its name could be anything, like this:

```
    body Tree::get {{new -value}} {
        switch -- $new {
            ...
        }
    }
```

If you use the special args argument when you declare a method, you can replace it with other arguments when you define the method body. The args argument represents variable argument lists, so it acts like a wildcard when the argument lists are compared by the body command.

If you want to completely suspend this consistency check, you can simply leave the argument list off when you declare the method in the class definition. The body command will have no argument list to compare against, so it will use whatever argument list you give it.

Since the constructor and destructor declarations have a slightly different syntax, their bodies *must* be included in the class definition. However, you can declare them with null bodies, and redefine the bodies later using the `body` command. If you do this, the argument list for the constructor must match whatever appears in the class definition, and the argument list for the destructor must always be null.

The `class` command defines the *interface* for a class, and subsequent `body` commands define the *implementation*. Separating the interface from the implementation not only makes the code easier to read, but as we will see below, it also supports interactive development.

Protection Levels: Public and Private

Usually, the class methods are the public part of an object, and the class variables are kept hidden inside. But what if you want to keep a method hidden for internal use? In our `Tree` class, for example, the `parent` method is used internally to tell a child that it has a new parent. If it is exposed, someone using the `Tree` class might be tempted to call it, and they could destroy the integrity of the tree. Or consider the opposite problem: What if you want to allow access to a variable? In our `Tree` class, the `name` and `value` variables are kept hidden within an object. We added a `get` method so that someone using the class could access these values, but there is a better way to handle this.

You can use the `public` and `private` commands to set the protection level for each member in the class definition. For example, we can use these commands in our `Tree` class as shown in Example 1-9.

Example 1-9 Adding protection levels to the Tree class.

```
class Tree {
    public variable name ""
    public variable value ""

    private variable parent ""
    private variable children ""

    constructor {args} {
        eval configure $args
    }
    destructor {
        clear
    }

    public method add {obj}
    public method clear {}
    private method parent {pobj}

    public method back {}
    public method contents {}
}
```

Any member can be accessed by methods within the same class, but only the public members are available to someone using the class. Since we declared the `parent` method to be private, it will not be visible to anyone outside of the class.

Each class has built-in `configure` and `cget` methods that mimic the behavior of Tk widgets. The `configure` method provides access to an object's attributes, and the `cget` method returns the current value for a particular attribute. Any variable declared to be public is treated as an attribute that can be accessed with these methods. Just by declaring the `name` and `value` variables to be public, for example, we can say:

```
Tree henry
henry configure -name "Henry Fonda" -value "great actor"
puts " name: [henry cget -name]"
puts "value: [henry cget -value]"
```

Just like Tk, the attribute names have a leading "`-`" sign. So if the variable is called `name`, the attribute is `-name`.

You can also set the attributes when you create an object, as long as you define the constructor as shown in Example 1-9. For example, we can say:

```
Tree henry -name "Henry Fonda" -value "great actor"
```

The extra arguments are captured by the `args` argument and passed along to the `configure` method in the constructor. The `eval` command is needed to make sure that the `args` list is not treated as a single argument, but as a list of option/value pairs. It is a good idea to write your constructor like this. It mimics the normal Tk behavior, and it lets someone using the class set some of the attributes, and leave others with a default value.

Now that we know about the built-in `cget` method, our `get` method is obsolete. We have removed it from the class definition in Example 1-9, in favor of a `back` method that can be used to query the parent for a particular node.

Since anyone can change a public variable by configuring it, we need a way to guard against bad values that might cause errors. And sometimes when an option changes, we need to do something to update the object. Public variables can have some extra code associated with them to handle these things. Whenever the value is configured, the code checks for errors and brings the object up to date. As an example, suppose we add a `-sort` option to the `Tree` class, to indicate how the contents of each node should be sorted. Whenever the `-sort` option is set, the code associated with it could reorder the child nodes. We could update the `Tree` class to handle sorting as shown in Example 1-10.

We add a `-sort` option simply by adding a public variable called `sort`. Its initial value is `""`, which means that by default, sorting is turned off. We can

Example 1-10 Tree class with a -sort option.

```
class Tree {
    public variable name ""
    public variable value ""

    public variable sort ""
    private variable lastSort ""

    private variable parent ""
    private variable children ""

    constructor {args} {
        eval configure $args
    }
    destructor {
        clear
    }

    public method add {obj}
    public method clear {}
    private method parent {pobj}

    public method back {}
    public method contents {}
    private method reorder {}
}

...

body Tree::add {obj} {
    $obj parent $this
    lappend children $obj
    set lastSort ""
}
body Tree::contents {} {
    reorder
    return $children
}

body Tree::reorder {} {
    if {$sort != $lastSort} {
        set children [lsort -command $sort $children]
    }
    set lastSort $sort
}

configbody Tree::sort {
    reorder
}
```

add some code to this variable in the class definition, right after its default value. Or we can define it later with a `configbody` command. The `configbody` command is just like the `body` command, but it takes two arguments: the name of the variable, and the body of code. There is no argument list for a variable, as you would have for a method. In this example, we use the `configbody` command near the end to define the code for the `sort` variable. Whenever the `-sort` option is configured, we call the `reorder` method to reorder the nodes.

If there are a lot of nodes, reordering them can be expensive. So we try to avoid sorting whenever possible. We have a variable called `lastSort` that keeps track of the last value for the `-sort` option, which is the name of some sorting procedure, as we'll see below. We can call the `reorder` method as often as we want, but it will reorder the nodes only if the `-sort` option has really changed.

We also set things up so that the nodes will be reordered properly if a new node is added. We could just reorder the list each time a node is added, but that would be expensive. Instead, we reorder the list when someone tries to query it via the `contents` method. Most of the time, the list will already be sorted, and the `reorder` method will do nothing. Whenever we add a node in the `add` method, we reset the value of `lastSort` to `""`, so that the next call to `contents` will actually reorder the nodes.

The `configure` method automatically guards against errors that occur when an option is set. For example, if we say:

```
Tree henry
henry configure -sort bogus_sort_proc -value 1
```

the `configure` method finds the public variable `sort` and sets it to the value `bogus_sort_proc`. Then it looks for code associated with this variable and executes it. In this case, it calls the `reorder` method to reorder the nodes using the procedure `bogus_sort_proc`. If this causes an error, the variable is automatically reset to its previous value, and the `configure` command aborts, returning an error message. Otherwise, it continues on with the next option, in this case handling the `-value` option.

Let's take a look at how the `-sort` option is actually used. In the `reorder` method, the `sort` value is given to the Tcl `lsort` command to do the actual sorting. The `lsort` command treats this as a comparison function. As it is sorting the list, it calls this function again and again, two elements at a time, and checks the result. The function should return "+1" if the first element is greater than the second, "-1" if the first is less than the second, and "0" if they are equal. The `lsort` command orders the two elements accordingly.

For example, if we want an alphabetical listing of `Tree` objects, we could write a function like this to compare the `-name` options:

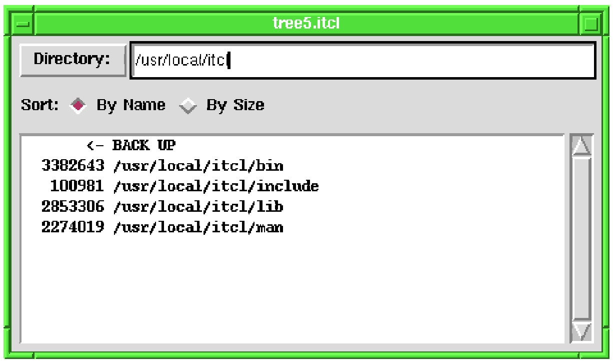
```
proc cmp_tree_names {obj1 obj2} {
    set val1 [$obj1 cget -name]
    set val2 [$obj2 cget -name]
    return [string compare $val1 $val2]
}
```

and we could tell a particular Tree object like `henry` to use this:

```
henry configure -sort cmp_tree_names
```

Its children would then be listed alphabetically. If we wanted a value-ordered list, we could write a function like `cmp_tree_values` to compare the `-value` attributes, and use that function as the `-sort` option.

We can put all of this together in a new and improved `du` browser, as shown in Figure 1-7. A complete code example appears in the file *itcl/tree/tree5.itcl*, but it works like this. When the user clicks on a radiobutton to change the sorting option, we configure the `-sort` option for the node being displayed, query its children, and update the listbox.



```
      <- BACK UP
3382643 /usr/local/itcl/bin
 100981 /usr/local/itcl/include
2853306 /usr/local/itcl/lib
2274019 /usr/local/itcl/man
```

Figure 1-7 An improved "du" browser with radiobuttons to control sorting.

Common Variables and Procedures

Sometimes it is necessary to have variables that do not belong to any particular object, but are shared among all objects in the class. In C++, they are referred to as *static data members*. In [INCR TCL], they are called *common variables*.

We can see the need for this in the following example. Suppose we improve our `du` application to have a graphical display like the one shown in Figure 1-8. Each file name has an icon next to it. We could use a canvas widget in place of a listbox, and draw each entry on the canvas with a `VisualRep` object, as we did in Example 1-2.

In this example, we will take things one step further. We set up the browser so that when you click on a file, it becomes selected. It is highlighted with a gray rectangle, and its usage information is displayed in a label at the bottom of the application.

We can fix up our `VisualRep` class to do most of the work for us. We will add `select` and `deselect` methods, so that each `VisualRep` object will know whether or not it is selected, and will highlight itself accordingly. A complete

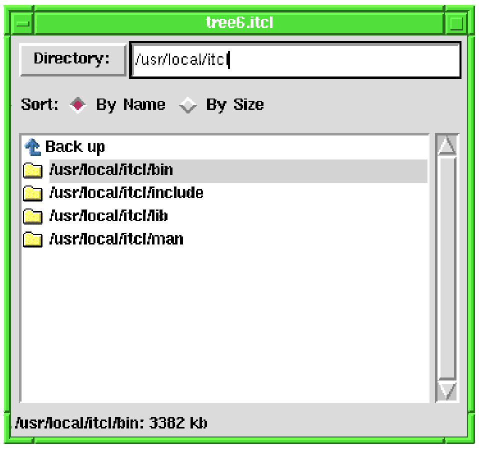


Figure 1-8 An improved "du" browser with a graphical display.

code example appears in the file *itcl/tree/tree6.itcl*, but the `VisualRep` class itself appears in Example 1-11.

Example 1-11 An improved VisualRep class with select/deselect methods.

```
image create photo defaultIcon -file default.gif

class VisualRep {
    public variable icon "defaultIcon"
    public variable title ""

    private variable canvas ""

    constructor {cwin args} {
        set canvas $cwin
        if {![info exists selectedObjs($canvas)]} {
            set selectedObjs($canvas) ""
        }
        eval configure $args
    }
    destructor {
        deselect
        $canvas delete $this
    }

    public method draw {ulVar midVar}
    public method select {}
    public method deselect {}

    public method canvas {args}

    private common selectedObjs
    public proc clear {canv}
    public proc selected {canv}
}
```

We have made a lot of improvements on the `VisualRep` class presented in Example 1-1. We still need to keep track of the canvas containing the `Visu-`

alRep, so we still have a private canvas variable. But we have added the public variables icon and title so that we can treat the icon image and the title string as configuration options. We also changed the constructor so that the canvas widget must be specified, but everything else is optional. If we create a VisualRep object like this:

```
canvas .display.canv
VisualRep vr1 .display.canv -title "/usr/local/lib"
```

we get the default icon with the title "/usr/local/lib". The constructor saves the canvas name in the canvas variable, does something with the selectedObjs array that we'll talk more about below, and then does the usual "eval configure $args" to handle the configuration options.

We also changed the way we use the draw method. We won't show the implementation here—you can check file *tree/tree6.itcl* for details—but this is how it works. Instead of a simple (x,y) coordinate, we pass in the names of two variables. These are used by the draw method, and then modified to return some drawing information. The first argument is an array representing the upper-left corner for the VisualRep object. If we have a VisualRep object called vr1 and we want its upper-left corner at the coordinate (25,37), we might call the draw method like this:

```
set ul(x) 25
set ul(y) 37
vr1 draw ul midpt
```

Before it returns, the draw method modifies the y coordinate in the ul array so that it points to the next position, immediately below the VisualRep object that we have just drawn. This makes it easy to draw a list of VisualRep objects on the canvas, even if their icons are different sizes. The draw method also stores the x and y coordinates for the midpoint of the icon in the midpt variable. This will come in handy for another example that we'll see later in this chapter.

As we said before, we have also added select and deselect methods to support file selection. When you click on a file in the browser, we call the select method for its VisualRep. Thus, if you click on a file that has a VisualRep named vr1, we call its select method like this:

```
vr1 select
```

the object would be highlighted with a gray rectangle. If we call the deselect method like this:

```
vr1 deselect
```

it would go back to normal. In theory, we could select as many objects as we want simply by calling their `select` methods. This might be useful in a file browser that allows many files to be moved, copied or deleted at once.

When multiple objects can be selected, we need to keep a list of all the `VisualRep` objects that are selected. But each `VisualRep` object keeps track of itself, and knows nothing about other objects in the class. Somewhere we have to keep a master list of selected objects. We want something like a global variable, but we want to keep it protected within the class, where it is actually used. In this case, we want a *common* variable.

We create a common variable called `selectedObjs`, as shown near the bottom of Example 1-11. We declare it to be private so that it can be accessed only within the class. Instead of keeping one master list with all the `VisualRep` objects that are selected, we keep a separate list for each canvas. That way, we can find out later what objects are selected on a particular canvas. To do this, we treat the `selectedObjs` variable as an array, with a different slot for each canvas. Whenever we create a `VisualRep` object, we make sure that a slot exists for its associated canvas, and if not, we create one. This is handled by some code in the constructor.

We handle the selection of a `VisualRep` object like this:

```
body VisualRep::select {} {
    $canvas itemconfigure $this-hilite -fill LightGray

    if {[lsearch $selectedObjs($canvas) $this] < 0} {
        lappend selectedObjs($canvas) $this
    }
}
```

The first statement turns on the gray rectangle on the canvas. In the `draw` method, we make an invisible rectangle tagged with the name `$this-hilite`, so when we want it to appear, we simply change its fill color. Next, we check to see if this object appears on the list of selected objects for its canvas. If not, we add it to the list.

Notice that we can access the `selectedObjs` variable without declaring it with anything like the Tcl `global` command. It has already been declared in the class definition, so it is known by all methods in the class.

We handle the de-selection like this:

```
body VisualRep::deselect {} {
    $canvas itemconfigure $this-hilite -fill ""

    set i [lsearch $selectedObjs($canvas) $this]
```

```
        if {$i >= 0} {
            set selectedObjs($canvas) [lreplace $selectedObjs($canvas) $i $i]
        }
    }
```

We turn off the gray rectangle by making its fill color invisible. Then we find the object on the list of selected objects, and we remove it from the list.

At this point, we know which `VisualRep` objects are selected, but we still haven't answered our question: What if someone using the class wants to get a list of all the `VisualRep` objects that are selected? Remember, the `selectedObjs` variable is private. It cannot be accessed outside of the class. We did this on purpose to prevent anyone else from tampering with it.

One way to solve this problem is to add a method called `selected` which returns a list of objects that are selected on a particular canvas. After all, a method has access to things inside the class. This would work, but then each time we wanted to use the method, we would need to find an object to talk to. For example, we might ask an object named `vr1` like this:

```
    set objlist [vr1 selected .display.canv]
```

This is awkward, and there is a better way to handle it. We need a function that belongs to the class as a whole. In C++, this is called a *static member function*. In [INCR TCL], it is called a *procedure* or *proc*. Class procedures are just like ordinary Tcl procedures, but they reside within the class, so their names won't conflict with other procedures in your application.

A procedure is declared with the `proc` command, as shown at the bottom of Example 1-11. In many respects, it looks like a method. But a procedure belongs to the class as a whole. It doesn't know about any specific object, so it doesn't have access to instance variables like `icon`, `title` and `canvas`. It has access only to common variables.

The advantage of using a procedure is that it can be called like this:

```
    set objlist [VisualRep::selected .display.canv]
```

Since we are calling this from outside of the class, we have to use the full name `VisualRep::selected`. But we do not have to talk to a specific object. In effect, we are talking to the class as a whole, asking for the objects that are selected on a particular canvas. The implementation of this procedure is fairly trivial:

```
body VisualRep::selected {canv} {
    if {[info exists selectedObjs($canv)]} {
        return $selectedObjs($canv)
    }
    return ""
}
```

We simply look for a value in the `selectedObjs` array, and return that list.

Procedures are also useful when you want to operate on several objects at once, or perhaps on the class as a whole. For example, we can add a `clear` procedure to deselect all of the `VisualRep` objects on a particular canvas. We might use the procedure like this:

```
VisualRep::clear .display.canv
```

and it is implemented like this:

```
body VisualRep::clear {canv} {
    if {[info exists selectedObjs($canv)]} {
        foreach obj $selectedObjs($canv) {
            $obj deselect
        }
    }
}
```

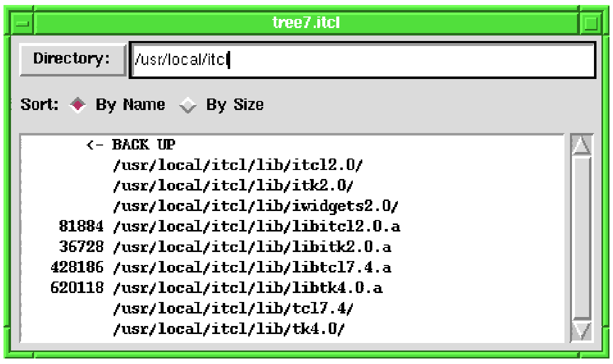
It simply finds the list of objects that are selected on the canvas, and tells each one to deselect itself.

Inheritance

Object-oriented systems provide a way for one class to borrow functionality from another. One class can *inherit* the characteristics of another, and add its own unique features. The more generic class is called a *base class*, and the more specialized class is called a *derived class*. This technique leads to a style of programming-by-differences, and helps to organize code into cohesive units. Without inheritance, object-oriented programming would be little more than a data-centric view of the world.

Single Inheritance

We can use our `Tree` class to build a regular file browser like the one shown in Figure 1-9. You enter a directory name at the top of the browser, and it lists the files and directories at that location. Directories are displayed with a trailing "`/`" character, and files are displayed along with their size in bytes. If you double-click on a directory name, the browser displays that directory. If you double-click on *BACK UP*, you go back to the parent directory.



```
       <- BACK UP
          /usr/local/itcl/lib/itcl2.0/
          /usr/local/itcl/lib/itk2.0/
          /usr/local/itcl/lib/iwidgets2.0/
  81884   /usr/local/itcl/lib/libitcl2.0.a
  36728   /usr/local/itcl/lib/libitk2.0.a
 428186   /usr/local/itcl/lib/libtcl7.4.a
 620118   /usr/local/itcl/lib/libtk4.0.a
          /usr/local/itcl/lib/tcl7.4/
          /usr/local/itcl/lib/tk4.0/
```

Figure 1-9 A simple file browser built with the FileTree class.

We could build a tree to represent all of the files on the file system and display it in this browser, just like we did for the `du` application. But instead of spending a lot of time to build a complete tree, we should start with a single node. When the user asks for the contents of a directory, we will look for files in that directory and add some nodes to the tree. With this scheme, we can bring up the file browser quickly and populate the tree as we go along.

We could add a little extra functionality to our `Tree` class to support the file system queries, but having a generic `Tree` class is useful for many different applications. Instead, it is better to create a separate `FileTree` class to represent the file system, and have it inherit the basic tree behavior from `Tree`. Inheritance relationships are often described as *is-a* relationships. If `FileTree` inherits from `Tree`, then a `FileTree` *is-a* `Tree`, but with a more specialized behavior. The relationship between these classes can be diagramed using the OMT notation[†] as shown in Figure 1-10.

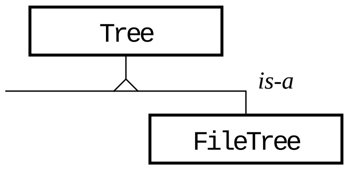


Figure 1-10 Diagram of the relationship between the Tree base class and its FileTree specialization.

† James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

The file *itcl/tree/tree7.itcl* contains a complete code example for the file browser, but the `FileTree` class is shown in Example 1-12. The `inherit` statement brings in all of the characteristics from the base class `Tree`. Because of this statement, the `FileTree` automatically acts like a tree. It keeps track of its parent and its children, and it has all of the usual `Tree` methods including `add`, `contents`, `back` and `clear`. It also has the configuration options `-name`, `-value` and `-sort`.

Example 1-12 The FileTree class inherits from Tree.

```
class FileTree {
    inherit Tree

    public variable procreate ""

    private variable file ""
    private variable mtime 0

    constructor {fname args} {
        if {![file exists $fname]} {
            error "file not found: $fname"
        }
        set file $fname
        eval configure $args
    }

    public method contents {}
    private method populate {}
}

body FileTree::populate {} {
    if {[file mtime $file] != $mtime} {
        clear
        foreach f [glob -nocomplain $file/*] {
            add [uplevel #0 $procreate $f]
        }
        set mtime [file mtime $file]
    }
}
body FileTree::contents {} {
    populate
    return [Tree::contents]
}
```

In the `FileTree` class, we redefine the `contents` method. When you ask for the contents of a `FileTree` node, we invoke another method called `populate` which automatically scans the file system and creates child nodes. After we have populated the node, we use the usual `Tree::contents` method to return the list of children.

Notice that we are careful to say `Tree::contents`. Whenever the base class and the derived class both have a method with the same name, you need to include a scope qualifier like this to avoid ambiguity. If you use a simple, unqualified name like `contents`, you will get the most-specific implementation for the object. For a `FileTree` object, the name `contents` means

`FileTree::contents`. If you want some other version of the method, you must use a qualified name like `Tree::contents`.

When an object gives you the most-specific implementation of a method, the method is said to be *virtual*. This is a fundamental feature of object-oriented programming. It lets you treat all the objects in a class the same way, but it lets specialized objects react in their own specialized manner. For example, all `Tree` objects have a `contents` method that returns a list of child nodes. So you can get the contents of either an ordinary `Tree` object or a `FileTree` object. When you get the contents of an ordinary `Tree` object, it simply returns a list of object names. But when you get the contents of a `FileTree` object, it will look for files and automatically create the child nodes before returning their names. You don't have to remember what kind of tree object you're talking to. You simply call the `contents` method, and each object does the right thing.

This is true even when you call a method from a base class context. Suppose for a moment that we had defined the `clear` method in the `Tree` base class like this:

```
body Tree::clear {} {
    set objs [contents]
    if {$objs != ""} {
        eval delete object $objs
    }
    set children ""
}
```

Instead of using the `children` variable directly, we have used the `contents` method to query the list of children. When you clear an ordinary `Tree` object, it would use `Tree::contents` to get the list of children. This simply returns `$children`, so it looks as though nothing has changed. But when you clear a `FileTree` object, it would use `FileTree::contents` to get the list of children. It would look for files and automatically create the child nodes, and then turn right around and delete them. In this case, using the `contents` method may be a dumb idea. But it does illustrate an important point: The methods that you call in a base class use the specialized behaviors that you provide later on for derived classes. Again, each object does the right thing depending on its type.

We set up the constructor so that you cannot create a `FileTree` object without saying what file or directory it represents. You might create a `FileTree` object like this:

```
FileTree barney /usr/local/lib -name "local libraries"
```

The first argument (`/usr/local/lib`) is assigned to the `fname` parameter. The constructor makes sure that the file exists, and then copies the name to the `file` variable. If the file is not found, the constructor returns an error, and the object creation is aborted.

The remaining arguments (`-name "local libraries"`) are treated as configuration options. They are absorbed by the `args` parameter, and they are applied by calling the `configure` method at the bottom of the constructor. Remember, a `FileTree` *is-a* `Tree`, so it has options like `-name` and `-value`.

When we query the contents of a `FileTree` node, it is automatically populated. The `populate` method treats the file name as a directory and uses the `glob` command to query its contents. We create a new `FileTree` object for each file in the directory and add it to the tree using the `add` method. Once a node has been populated, we save the modification time for its file in the `mtime` variable. We can call `populate` as often as we like, but the node will not be re-populated unless the modification time changes.

Each `FileTree` object populates itself by adding new `FileTree` objects as child nodes. We'll call this process *procreation*. We could create the offspring directly within the `populate` method, but this would make it hard to use the same `FileTree` in lots of different file browsers. For example, one file browser might set the `-value` option on each `FileTree` object to store the size of the file, so files could be sorted based on size. Another might set the `-value` option to store the modification time, so files could be sorted by date. We want to allow for both of these possibilities (and many more) when we create each `FileTree` object.

One solution is to add a procreation method to the `FileTree` class. The `populate` method would call this whenever it needs to create a `FileTree` object. We could have lots of different derived classes that overload the procreation method and create their offspring in different ways. This approach works fine, but we would probably find ourselves creating lots of new classes simply to override this one method.

Instead, let's think for a moment about the Tk widgets. You may have lots of buttons in your application, but they all do different things. Each button has a `-command` option that stores some code. When you push a button, its `-command` code gets executed.

In the same manner, we can add a `-procreate` option to the `FileTree` class. Whenever a `FileTree` object needs to procreate, it calls whatever procedure you specify with the `-procreate` option, passing it the file name for the child object. This is what we do in the `populate` method, as you can see in Example 1-12.

Whenever you have an option that contains code, you have to be careful how you execute the code. We could use the `eval` command to execute the procreation code, but it might be more than just a procedure name. For all we know, it could be a whole script of code. If it sets any variables, we don't want to affect variables inside the `populate` method by accident. Instead, we use "`uplevel #0`" to evaluate the command at the global scope, outside of the `FileTree` class. If it accidentally sets a variable like `file`, it will be a global variable called `file`, and not the private variable `file` that we can access inside the `populate` method. We will explore scoping issues like this in more detail later in this chapter. But for now, just remember to use "`uplevel #0`" to evaluate any code passed in through a configuration option.

We can tell a `FileTree` object like `barney` to procreate with a custom procedure like this:

```
barney configure -procreate create_node
```

When `barney` needs to procreate, it calls `create_node` with the child's file name as an argument. This in turn creates a `FileTree` object for the file, configures options like `-name`, `-value` and `-sort`, and returns the name of the new object. For example, we could use a procedure like this to set the file modification time as the value for each node:

```
proc create_node {fname} {
    set obj [FileTree #auto $fname -name "$fname"]
    $obj configure -value [file mtime $fname]
    return $obj
}
```

We can use all of this to build the file browser shown in Figure 1-9. Again, the file *itcl/tree/tree7.itcl* contains a complete code example, but the important parts are shown in Example 1-13.

When you enter a directory name at the top of the browser, we call the `load_dir` procedure to build a new file tree. If there is an existing tree, we destroy it by destroying its root node. Then, we create a new root object to represent the tree. At some point, we use another procedure called `show_dir` (not shown here) to display the contents of this node in a listbox. When you double-click on a directory, we call `show_dir` for that node. When you double-click on *BACK UP*, we call `show_dir` for the parent node. Whenever we call `show_dir`, it asks for the contents of a node, and the node populates itself as needed.

The root object uses the `create_node` procedure to procreate. When its child nodes are created, directory names are given a trailing "/", and regular files are given a value that represents their size. All child nodes are configured to procreate using the same `create_node` procedure, so each node expands the same way.

Example 1-13 A simple file browser built with the FileTree class.

```
set root ""
proc load_dir {dir} {
    global root

    if {$root != ""} {
        delete object $root
    }
    set root [FileTree #auto $dir -procreate create_node]
    return $root
}

proc create_node {fname} {
    if {[file isdirectory $fname]} {
        set obj [FileTree #auto $fname -name "$fname/"]
    } else {
        set obj [FileTree #auto $fname -name $fname]
        $obj configure -value [file size $fname]
    }
    $obj configure -procreate create_node

    return $obj
}
```

Multiple Inheritance

Suppose we want to create a file browser with a graphical display like the one shown in Figure 1-11.

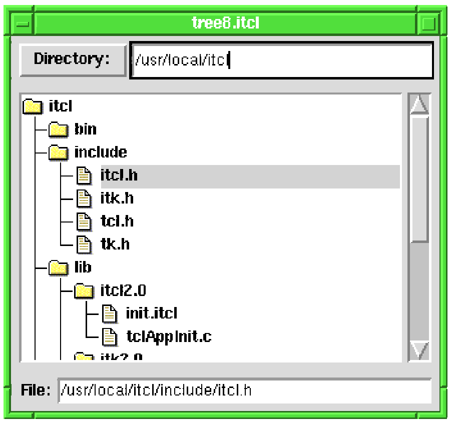


Figure 1-11 A file browser with a graphical display.

We have all of the pieces that we need. We can use the `FileTree` class to store the file hierarchy, and the `VisualRep` class to draw file elements on a canvas.

But how do we combine these elements together? One solution is to use inheritance. We might create a class VisualFileTree to represent each file on the display. We could say that VisualFileTree *is-a* FileTree, since it represents a node in the file hierarchy, and VisualFileTree *is-a* VisualRep, since it will be drawn on a canvas. In this case, VisualFileTree needs to inherit from two different base classes. This is called *multiple inheritance*. A diagram of these relationships is shown in Figure 1-12.

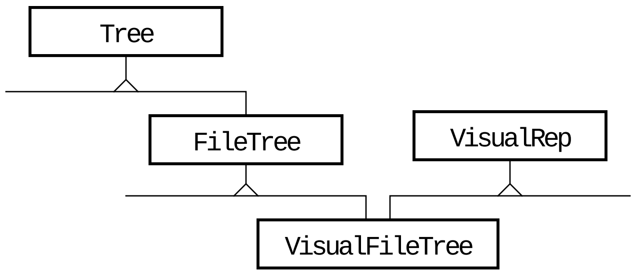


Figure 1-12 Diagram of class relationships with multiple inheritance.

The file *itcl/tree/tree8.itcl* contains a complete code example for the file browser, but the VisualFileTree class itself is shown in Example 1-14.

Example 1-14 VisualFileTree class used for the file browser shown in Figure 1-11.

```
class VisualFileTree {
    inherit FileTree VisualRep

    public variable state "closed"
    public variable selectcommand ""

    constructor {file cwin args} {
        FileTree::constructor $file
        VisualRep::constructor $cwin
    } {
        eval configure $args
    }

    public method select {}
    public method toggle {}

    public method draw {ulVar midVar}
    public method refresh {}
}

body VisualFileTree::select {} {
    VisualRep::clear $canvas
    VisualRep::select
    regsub -all {%o} $selectcommand $this cmd
    uplevel #0 $cmd
}

body VisualFileTree::toggle {} {
    if {$state == "open"} {
        set state "closed"
    } else {
```

Example 1-14 VisualFileTree class used for the file browser shown in Figure 1-11.

```
        set state "open"
    }
    refresh
}

configbody VisualFileTree::state {
    if {$state != "open" && $state != "closed"} {
        error "bad value \"$state\": should be open or closed"
    }
    refresh
}

body VisualFileTree::draw {ulVar midVar} {
    upvar $ulVar ul
    upvar $midVar mid

    VisualRep::draw ul mid
    $canvas bind $this <ButtonPress-1> "$this select"
    $canvas bind $this <Double-ButtonPress-1> "$this toggle"

    set lr(x) [expr $ul(x) + 2*($mid(x)-$ul(x))]
    set lr(y) $ul(y)

    if {$state == "open"} {
        foreach obj [contents] {
            $obj draw lr mid2
            set id [$canvas create line \
                $mid(x) $mid(y) $mid(x) $mid2(y) $mid2(x) $mid2(y) \
                -fill black]
            $canvas lower $id
        }
    }
    set ul(y) $lr(y)
}

body VisualFileTree::refresh {} {
    set root $this
    while {[$root back] != ""} {
        set root [$root back]
    }

    set oldcursor [$canvas cget -cursor]
    $canvas configure -cursor watch
    update
    $canvas delete all

    set ul(x) 5
    set ul(y) 5
    $root draw ul mid
    set bbox [$canvas bbox all]
    $canvas configure -cursor $oldcursor -scrollregion $bbox
}
```

Each class can have only one `inherit` statement, but it can declare several base classes, which should be listed in their order of importance. First and foremost, `VisualFileTree` is a `FileTree`, but it is also a `VisualRep`. This means that any methods or variables that are not defined in `VisualFileTree` are found first in `FileTree`, and then in `VisualRep`. When base classes have members with the same name, their order in the `inherit` statement can affect the behavior of the derived class.

Notice that we added a -state option to VisualFileTree, and we redefined the draw method to handle it. When we draw a node that has -state set to "open", we also draw the file hierarchy underneath it. First, we call VisualRep::draw to draw the file name and its icon on the canvas. Then, if this object is in the "open" state, we scan through the list of child nodes and tell each one to draw itself in the space below. If a child is also in the "open" state, it will tell its children to draw themselves, and so on.

It is easy to arrange things on the canvas. The draw method does all of the hard work. As you will recall from Example 1-11, we use the ul array to pass in the (x,y) coordinate for the upper-left corner of the icon. When we call VisualRep::draw, it draws only a file name and an icon, and it shifts ul(y) down below them. When we call VisualFileTree::draw, it draws a file name and an icon, and perhaps an entire file tree below it. But again, it shifts ul(y) down so we are ready to draw the next element.

The draw method also returns the midpoint of the icon via the midVar argument. This makes it easy to draw the connecting lines between a parent icon and each of the child icons. In the VisualFileTree::draw method, for example, we capture the parent coordinate in the mid array. When we call the draw method for the child, it returns the child coordinate in the mid2 array. We then draw the lines connecting these two points.

As we draw each file entry, we add some bindings to it. If you click on a file, we call the select method to select it. If you double-click on a file, we call the toggle method to toggle it between the "open" and "closed" states.

We redefined the select method for a VisualFileTree object to support a -selectcommand option. This is a lot like the -command option for a button widget. It lets you do something special each time a VisualFileTree object is selected. When we call the select method, it first calls VisualRep::clear to deselect any other files, and then calls the base class method VisualRep::select to highlight the file. Finally, it executes the code stored in the -selectcommand option. We use "uplevel #0" to execute this code at the global scope, so it doesn't change any variables within the select method by accident.

If the -selectcommand code contains the string "%o", we use regsub to replace it with the name of the VisualFileTree object before the code is executed. This is similar to the way the Tk bind command handles fields like "%x" and "%y". This feature lets us use the same -selectcommand for all of our VisualFileTree objects, but each time it is executed, we know which object was selected.

The `toggle` method toggles the `-state` option between `open` and `closed`, and refreshes the drawing on the canvas. In effect, this opens or closes a folder in the file hierarchy.

The `refresh` method should be called whenever anything changes that would affect the drawing on the canvas. Whenever the `-state` option changes, for instance, we need to refresh the drawing to expand or collapse the file tree at that point. The configbody for the `state` variable first checks to see if the new state is valid, and then calls `refresh` to update the drawing. The `refresh` method searches up through the hierarchy to find the root of the tree. It clears the canvas and then tells the root object to draw itself at the coordinate (`5,5`). If the root is “open,” then its children will be drawn, and if they are “open,” their children will be drawn, and so forth. The entire drawing is regenerated with just one call to `refresh`.

Protection Levels: Protected

So far, we have discussed two protection levels. Private class members can be accessed only in the class where they are defined. Public members can be accessed from any context. When one class inherits another, therefore, the inherited members that are public can be accessed from the derived class context. The private members are completely private to the base class.

Some members sit in the gray area between public and private. They need to be accessed in derived classes, but they should not be exposed to anyone using the class. For example, in the `VisualRep` base class shown in Example 1-11, we defined a `canvas` variable to store the name of the canvas used for drawing. Since this is a private variable, a derived class like `VisualFileTree` does not have access to it. The methods shown in Example 1-14 like `VisualFileTree::draw` and `VisualFileTree::select` will fail, claiming that `canvas` is an undefined variable.

Like C++, [INCR TCL] provides a third level of protection that falls between public and private. When members need to be shared with derived classes but shielded from anyone using the class, they should be declared *protected*. We can fix the `VisualRep` class to use a protected variable as shown in Example 1-15.

Example 1-15 “Protected” members can be accessed in derived classes.

```
class VisualRep {
    public variable icon "default"
    public variable title ""

    protected variable canvas ""

    ...
```

Example 1-15 “Protected” members can be accessed in derived classes.

```
}

class VisualFileTree {
    inherit FileTree VisualRep
    ...
    public method select {}
    ...
}

body VisualFileTree::select {} {
    VisualRep::clear $canvas
    VisualRep::select
    regsub -all {%o} $selectcommand $this cmd
    uplevel #0 $cmd
}
```

As a rule, it is better to use public and private declarations for most of your class members. Public members define the class interface, and private members keep the implementation details well hidden. Protected members are useful when you are creating a base class that is meant to be extended by derived classes. A few methods and variables may need to be shared with derived classes, but this should be kept to a minimum. Protected members expose implementation details in the base class. If derived classes rely on these details, they will need to be modified if the base class ever changes.

Constructors and Destructors

Each class can define one constructor and one destructor. However, a class can inherit many other constructors and destructors from base classes.

When an object is created, all of its constructors are invoked in the following manner. First, the arguments from the object creation command are passed to the most-specific constructor. For example, in the command:

```
VisualFileTree #auto /usr/local/lib .canv -icon dirIcon
```

the arguments “`/usr/local/lib .canv -icon dirIcon`” are passed to `VisualFileTree::constructor`. If any arguments need to be passed to a base class constructor, the derived constructor should invoke it using a special piece of code called an *initialization statement*. This statement is sandwiched between the constructor’s argument list and its body. For example, the `VisualFileTree` class shown in Example 1-14 has an initialization statement that looks like this:

```
FileTree::constructor $file
VisualRep::constructor $cwin
```

The `file` argument is passed to the `FileTree::constructor`, and the `cwin` argument is passed to the `VisualRep::constructor`. The remaining arguments are kept in the `args` variable, and are dealt with later.

After the initialization statement is executed, any base class constructors that were not explicitly called are invoked without arguments. If there is no initialization statement, all base class constructors are invoked without arguments. This guarantees that all base classes are fully constructed before we enter the body of the derived class constructor.

Each of the base class constructors invoke the constructors for their base classes in a similar manner, so the entire construction process is recursive. By default, an object is constructed from its least-specific to its most-specific class. If you’re not sure which is the least-specific and which is the most-specific class, ask an object to report its heritage. If we had a `VisualFileTree` object named `fred`, we could query its heritage like this:

```
% fred info heritage
VisualFileTree FileTree Tree VisualRep
```

This says that `VisualFileTree` is the most-specific class and `VisualRep` is the least-specific. By default, the constructors will be called in the order that you get by working backward through this list. Class `VisualRep` would be constructed first, followed by `Tree`, `FileTree`, and `VisualFileTree`. Our initialization statement changes the default order by calling out `FileTree::constructor` before `VisualRep::constructor`.

Objects are destroyed in the opposite manner. Since there are no arguments for the destructor, the scheme is a little simpler. The most-specific destructor is called first, followed by the next most-specific, and so on. This is the order that you get by working forward through the heritage list. `VisualFileTree` would be destructed first, followed by `FileTree`, `Tree` and `VisualRep`.

Inheritance versus Composition

Inheritance is a way of sharing functionality. It merges one class into another, so that when an object is created, it has characteristics from both classes. But in addition to combining classes, we can also combine objects. One object can contain another as a component part. This is referred to as a *compositional* or *has-a* relationship.

For example, suppose we rewrite our `VisualFileTree` class so that a `VisualFileTree` *is-a* `FileTree`, but *has-a* `VisualRep` as a component part. Figure 1-13 shows a diagram of this design.

The code for this `VisualFileTree` class is quite similar to Example 1-14, but we have highlighted several important differences in bold type. Whenever we create a `VisualFileTree` object, we create a separate `VisualRep` object to handle interactions with the canvas. We create this component in the

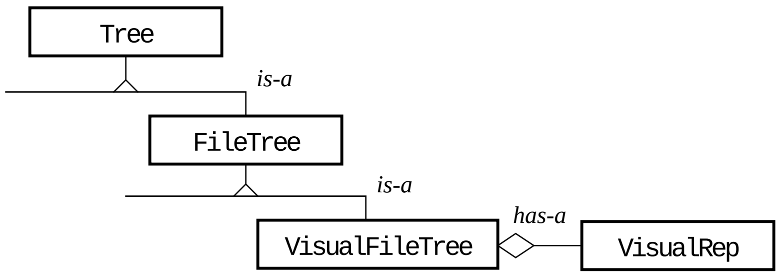


Figure 1-13 VisualFileTree class has-a VisualRep component.

Example 1-16 VisualFileTree class which brings in VisualRep using composition instead of inheritance.

```
class VisualFileTree {
    inherit FileTree

    public variable state "closed"
    public variable selectcommand ""

    public variable icon "" {
        $vis configure -icon $icon
    }
    public variable title "" {
        $vis configure -title $title
    }

    private variable vis ""

    constructor {file cwin args} {
        FileTree::constructor $file
    } {
        set vis [VisualRep #auto $cwin -icon $icon -title $title]
        eval configure $args
    }
    destructor {
        delete object $vis
    }

    public method select {}
    public method toggle {}

    public method draw {ulVar midVar}
    public method refresh {}
}

body VisualFileTree::select {} {
    VisualRep::clear [$vis canvas]
    $vis select
    regsub -all {%o} $selectcommand $this cmd
    uplevel #0 $cmd
}

...
```

constructor, and save its name in the variable `vis`. We delete this component in the destructor, so that when a `VisualFileTree` object is deleted, its `VisualRep` component is deleted as well. If we didn't do this, the `VisualRep` components would hang around indefinitely, and we would have a memory leak.

With inheritance, all of the public members from the base class are automatically integrated into the derived class, becoming part of its interface. With composition, nothing is automatic. If you need to access a method or a configuration option on the component, you must write a "wrapper" in the containing class. For example, the `VisualRep` component has `-icon` and `-title` options that control its appearance. If we want to be able to set `-icon` and `-title` for the `VisualFileTree` object, we must explicitly add these variables, and include configbody code to propagate any changes down to the `VisualRep` component.

With inheritance, we have access to protected data members defined in the base class. With composition, we have access only to the public interface for the component part. Since the `VisualRep` is now a separate object, we cannot access its `canvas` variable from `VisualFileTree`. But we can call its `canvas` method to query the name of its canvas. (We were smart enough to add this back in Example 1-11, although we hardly mentioned it at the time.) We use this in the `select` method to clear other `VisualRep` objects on the same canvas before selecting a new one.

Inheritance and composition are like two sides of the same coin. Sometimes inheritance leads to a better solution, sometimes composition. Many problems are solved equally well using either approach. Knowing whether to use inheritance or composition is a matter of experience and judgement, but I can give you a few simple guidelines here.

- Use inheritance to create layers of abstraction.

  For example, the code for a `VisualFileTree` is neatly abstracted into three classes: `VisualFileTree` *is-a* `FileTree`, which *is-a* `Tree`. Now suppose that we have a problem with the `VisualFileTree`. We won't have to search through all of the code to find the bug. If the problem has to do with the tree, we look in the `Tree` class. If it has to do with the file system, we look in the `FileTree` class. And so on.

- Use inheritance to build a framework for future enhancements.

  We can extend our tree library at any point by adding new classes into the hierarchy. For example, we might create a class `WidgetTree` that *is-a* `Tree`, but adds code to query the Tk widget hierarchy. We might create a class `SourceFileTree` that *is-a* `FileTree`, but adds methods to support source code control.

- Use composition when you catch yourself making exceptions to the *is-a* rule.

  With inheritance, all of the public variables and all of the methods in the base class apply to the derived class. For example, `FileTree` is-a `Tree`, so we can treat it exactly like any other `Tree` object. We can add nodes to it, reorder the nodes, clear the nodes, and set the `-name`, `-value` and `-sort` options. If you catch yourself making exceptions to this, then you are no longer talking about inheritance.†

  Suppose you're thinking that `FileTree` is like a `Tree`, except that you can't clear it, and it doesn't have the `-value` option. In that case, you should add the tree behavior using composition instead of inheritance. You could say that `FileTree` *has-a* `Tree` within it to maintain the actual data. The `Tree` would be completely hidden, but you could wrap the methods and the options that you want to expose.

- Use composition when the relationships between classes are dynamic.

  Again, with inheritance `FileTree` *is-a* `Tree`, once and for all time. Suppose you wanted to have `FileTree` switch dynamically between a tree representation and a flat list of files. In that case, you would be better off using composition to support interchangeable parts. You could say that `FileTree` *has-a* `Tree`, or that `FileTree` *has-a* `List`, depending on its mode of operation.

- Use composition when a single object must have more than one part of the same type.

  When we first presented class `VisualFileTree`, for example, we said that `VisualFileTree` *is-a* `VisualRep`, which appears on a canvas. But suppose that you wanted a single `VisualFileTree` object to appear on many different canvases. You could support this using composition. You could say that `VisualFileTree` *has-a* `VisualRep` component for each canvas that it appears on.

- Use composition to avoid deep inheritance hierarchies.

  With inheritance, each class builds on the one before it. At first, this seems like an exciting way to reuse code. But it can easily get out of hand. At some point, it becomes impossible to remember all the details that build up in a series of base classes. Most programmers reach their limit after some-

† C++ lets you suppress certain things coming from a base class through private inheritance. This evil feature is not supported by [INCR TCL].

thing like 5 levels of inheritance. If you trade off some of your inheritance relationships for composition, you can keep your hierarchies smaller and more manageable.

- If you can't decide between inheritance and composition, favor composition.

  Inheritance lets you reuse code, but it is white-box reuse. Each base class is exposed—at least in part—to all of its derived classes. You can see this in Example 1-15. The `VisualFileTree` class relies on the `canvas` variable coming from the `VisualRep` base class. This introduces coupling between the two classes and breaks encapsulation. If we ever change the implementation of `VisualRep`, we may have to revisit `VisualFileTree`.

  On the other hand, composition supports black-box reuse. The internal workings of each object are completely hidden behind a well-defined interface. In Example 1-16, we modified the `VisualFileTree` class to use a `VisualRep` component. Instead of relying on its internal `canvas` variable, we used a well-defined method to interact with its canvas. Therefore, `VisualFileTree` is completely shielded from any changes we might make inside `VisualRep`.

Neither inheritance nor composition should be used exclusively. Using only one or the other is like using only half of the tools in a tool box. The choice of tool should be based on the problem at hand. Realistic designs have many different classes with a mixture of both relationships.

Namespaces

A *namespace* is a collection of commands, variables and classes that is kept apart from the usual global scope. It provides the extra packaging needed to create reusable libraries that plug-and-play with one another.

For example, suppose we want to reuse our file browser code in other applications. We need to include our classes, along with procedures like `load_dir` and `create_node` shown in Example 1-13. But if an application happens to have procedures named `load_dir` or `create_node`, adding the file browser code will break it. If an application already uses a global variable named `root`, calling the `load_dir` procedure will corrupt its value.

Name collisions like this make it difficult to construct large Tcl/Tk applications. They cause strange errors that are difficult to debug, and they are a barrier to code reuse. But when commands, variables and classes are packaged in their own namespace, they are shielded from the rest of an application. Libraries can be used freely, without fear of unwanted interactions.

Creating Namespaces

We can turn our file browser code into a file browser library by packaging it in a namespace. A complete code example appears in the file *itcl/tree/tree10.itcl*, but the important parts are shown in Example 1-17. Variables and procedures are added to a namespace in much the same way that they are added to a class. Procedures are defined using the usual `proc` command. Variables are defined using the `variable` command, which may include an initialization value. These are not instance variables like you would have in a class. These variables act like ordinary “global” variables, but they reside within the namespace, and not at the usual global scope. Defining a variable causes it to be created, but unlike a class, the variable is not automatically available in the procedures in the namespace. You must declare each variable with the Tcl `global` command to gain access to it.

Example 1-17 Namespace for the file browser library.

```
namespace filebrowser {
    variable roots

    proc load_dir {cwin dir {selcmd ""}} {
        global roots

        if {[info exists roots($cwin)]} {
            delete object $roots($cwin)
        }
        set roots($cwin) [create_node $cwin $selcmd $dir]
        $roots($cwin) configure -state open
        $roots($cwin) refresh

        return $roots($cwin)
    }

    proc create_node {cwin selcmd fname} {
        ...
    }

    proc cmp_tree {option obj1 obj2} {
        ...
    }
}
```

Within the context of the namespace, commands and variables can be accessed using simple names like `load_dir` and `roots`. All of the procedures defined in a namespace execute in that context, so within the body of `load_dir`, we can access things like `create_node` and `roots` without any extra syntax. In another context, names must have an explicit namespace qualifier. For example, an application could use the `load_dir` procedure like this:

```
filebrowser::load_dir .display.canv /usr/local/lib
```

This is just how we would call a class procedure, and the similarity is no accident. A class is a namespace, but with a little extra functionality to create and manage objects. Classes are also more rigid. Once the class interface is defined, it cannot be modified unless the class is deleted. But a namespace can be updated on-the-fly to create, redefine or delete commands and variables.

We can add another procedure to the `filebrowser` namespace with another `namespace` command, like this:

```
namespace filebrowser {
    proc all {} {
        global roots
        return [array names roots]
    }
}
```

This activates the `filebrowser` context, and then executes the `proc` command within it, defining the new procedure. Another way of creating the procedure is to define it with an ordinary `proc` command, but include the namespace context in its name:

```
proc filebrowser::all {} {
    global roots
    return [array names roots]
}
```

The procedure can be deleted like this:

```
namespace filebrowser {
    rename all ""
}
```

or like this:

```
rename filebrowser::all ""
```

An entire namespace can be deleted using the `delete` command, like this:

```
delete namespace filebrowser
```

This deletes all commands and variables in the namespace, and removes all trace of the namespace itself.

The namespace containing a command or variable is part of the identity for that command or variable. Elements with the same name in another namespace are totally separate. Suppose we wrap our `du` browser in a namespace, as shown in Example 1-18.

Example 1-18 Namespace for the “du” browser library.

```
namespace diskusage {
    variable roots

    proc load_dir {twin dir} {
        global roots

        set parentDir [file dirname $dir]
        set roots($twin) [Tree ::#auto -name $parentDir]
        set hiers($parentDir) $roots($twin)
```

Example 1-18 Namespace for the “du” browser library.

```
        set info [split [exec du -b $dir] \n]
        set last [expr [llength $info]-1]

        for {set i $last} {$i >= 0} {incr i -1} {
            ...
        }
        show_dir $twin $roots($twin)
        ...
    }

    proc show_dir {twin obj} {
        ...
    }

    proc add_entry {twin line obj} {
        ...
    }

    proc cmp_tree {obj1 obj2} {
        ...
    }
}
```

The `diskusage` namespace also contains a `load_dir` command and a `roots` variable, but they are completely separate from those in the `filebrowser` namespace. This is obvious when we try to use them. An application could load a directory into the file browser like this:

```
filebrowser::load_dir .display.canv /usr/local/lib
```

and display the usage information for a directory like this:

```
diskusage::load_dir .textwin /usr/local/lib
```

The explicit namespace qualifiers remove the ambiguity between these two commands.

One namespace can contain another namespace inside it, so one library can have its own private copy of another library. For example, we could include the `diskusage` library within the `filebrowser` library like this:

```
namespace filebrowser {
    namespace diskusage {
        variable roots
        proc load_dir {twin dir} {
            ...
        }
        ...
    }
}
```

Within the `filebrowser` namespace, the usage information for a directory could be displayed as shown earlier:

```
namespace filebrowser {
    diskusage::load_dir .textwin /usr/local/lib
}
```

Outside of `filebrower`, the complete namespace path must be specified:

```
filebrowser::diskusage::load_dir .textwin /usr/local/lib
```

Every interpreter has a global namespace called "`::`" which contains all of the other namespaces. It also contains the usual Tcl/Tk commands and global variables. Each Tcl/Tk application starts off in this namespace, which I call the *global context*. When you define other namespaces and call their procedures, the context changes.

Name Resolution

Qualified names are like file names in the Unix file system, except that a "`::`" separator is used instead of "`/`". Any name that starts with "`::`" is treated as an absolute reference from the global namespace. For example, the command

```
::filebrowser::diskusage::load_dir .textwin /usr/local/lib
```

refers to the `load_dir` command in the `diskusage` namespace, in the `filebrowser` namespace, in the global namespace.

If a name does not have a leading "`::`", it is treated relative to the current namespace context. Lookup starts in the current namespace, then continues along a search path. Each namespace has an *import* list that defines its search path. When a namespace is added to the import list, all of the commands and variables in that namespace can be accessed with simple names.

For example, we could import the `filebrowser` namespace into the global namespace like this:

```
import add filebrowser
```

We could then use the `load_dir` command in the global namespace without an explicit qualifier, like this:

```
load_dir .display.canv /usr/local/lib
```

The `load_dir` command is not found directly in the global namespace, but resolution continues along the import path to the `filebrowser` namespace, where the `filebrowser::load_dir` command is found.

It is okay to import other namespaces that have the same command or variable names. We could import the `diskusage` namespace, even though it also has a `load_dir` procedure. The first command or variable found along the import path is the one that gets used.

If you have any questions regarding name resolution, they can be answered by using the "info which" command. This command returns the fully qualified name for any command, variable or namespace in the current context. In this example, the command:

```
info which -command load_dir
```

would return the fully qualified name ::filebrowser::load_dir.

By default, each namespace imports its parent, so commands and variables in the global namespace are automatically accessible. Other import relationships should be used sparingly. After all, if the global namespace imported all of the others, we would be back to one big pot of commands and variables, and there wouldn't be much point to having namespaces.

Using Objects Outside of Their Namespace

If you create an object within a namespace, you'll have trouble referring to it outside of the namespace. Suppose you create a VisualFileTree object within the filebrowser namespace like this:

```
namespace filebrowser {
    VisualFileTree fred /usr/local/lib .display.canv
}
```

and then you try to add a node to it in another namespace like this:

```
namespace diskusage {
    VisualFileTree wilma /usr/local/bin .display.canv
    fred add wilma
}
```

This will fail. Since the fred object was created in the filebrowser namespace, the fred command is local to that namespace. We will not be able to find a fred command in diskusage unless the filebrowser namespace is somewhere on its import path.

Usually, this is a good thing. Namespaces are doing their job of keeping the two packages separate, and protecting the elements inside them. But from time to time, you will want to share objects between packages. This problem all has to do with naming, and it can be solved through proper naming too.

One solution is to use the full name of an object when you are referring to it in another namespace. For example, we could say:

```
namespace diskusage {
    VisualFileTree wilma /usr/local/bin .display.canv
    ::filebrowser::fred add wilma
}
```

You may have noticed that an object's `this` variable reports the full name of the object, including its namespace path. This is the reason. If you use `$this` is a command, you will be able to find the object from any context. When you use the full name, you leave nothing to chance in command resolution.

Another solution is to create the object in some namespace that all of your packages naturally import. For example, all namespaces import the global "`::`" namespace. You can create an object in the global namespace like this:

```
namespace filebrowser {
    uplevel #0 VisualFileTree fred /usr/local/lib .display.canv
}
```

or like this:

```
namespace filebrowser {
    namespace :: { VisualFileTree fred /usr/local/lib .display.canv }
}
```

or like this:

```
namespace filebrowser {
    VisualFileTree ::fred /usr/local/lib .display.canv
}
```

In the first case, we use the "`uplevel #0`" command to transition to the 0th call frame, which is the global context, and we create the object there. In the second case, we use the `namespace` command to get the same effect. In the third case, we execute the `VisualFileTree` command in the `filebrowser` namespace, but we give the object a name that belongs to the global namespace. The effect is the same. We create an object named `fred` that we can access from the global namespace, and therefore, we can access it from any namespace in the application.

Instead of putting an object all the way out in the global namespace, you may want to put it in a more restricted namespace that only certain packages have access to. Remember, namespaces can be nested, and each namespace automatically imports things from its parent. We could wrap the `filebrowser` and the `diskusage` namespace in another namespace called `filestuff`, for example, and put all of the shared objects in `filestuff`:

```
namespace filestuff {
    namespace filebrowser {
        ...
        VisualFileTree ::filestuff::fred /usr/local/lib .display.canv
    }
    namespace diskusage {
        ...
        VisualFileTree ::filestuff::wilma /usr/local/bin .display.canv
        fred add wilma
    }
}
```

That way, these objects can still be shared across `filebrowser` and `diskusage`, but they won't interfere with any other packages.

Sometimes it is easy to forget that other classes need access to an object. When the `Tree` class adds an object to a tree, for example, it needs to talk to that object to set its parent. If all of our `Tree` objects are sitting in the `filestuff` namespace, but the `Tree` class itself is sitting one level up in the global namespace, we will again have problems. As much as possible, keep all of the code related to a package together in the same namespace. If the `Tree` class is needed only for the `filebrowser` package, put it in the `filebrowser` namespace. If it needs to be shared across both the `filebrowser` and the `diskusage` packages, put it above them in the `filestuff` namespace.

Classes can be defined within a namespace like this:

```
namespace filestuff {
    class Tree {
        ...
    }
    class FileTree {
        ...
    }
    ...
}
```

or like this:

```
class filestuff::Tree {
    ...
}
class filestuff::FileTree {
    ...
}
...
```

In either case, the classes are completely contained within the `filestuff` namespace, so if an application has another `Tree` class, it will not interfere with the one in the `filestuff` namespace. More importantly, since the `Tree` class now resides within `filestuff`, it automatically has access to the objects in `filestuff`.

Protection Levels

Just as you can have public, private and protected elements in a class, you can have public, private and protected elements in a namespace. This helps to document your interface, so that someone using your library knows which variables and procedures they can access, and which ones they should leave alone. For example, look at the filebrowser library shown in Example 1-19. It is obvious that `load_dir` procedure is the only thing that you need to use to access a file browser. Everything else is private to the `filebrowser` namespace.

Example 1-19 File browser library with public/private declarations.

```
namespace filebrowser {
    private variable roots

    public proc load_dir {cwin dir {selcmd ""}} {
        global roots

        if {[info exists roots($cwin)]} {
            delete object $roots($cwin)
        }
        set roots($cwin) [create_node $cwin $selcmd $dir]
        $roots($cwin) configure -state open
        $roots($cwin) refresh

        return $roots($cwin)
    }

    private proc create_node {cwin selcmd fname} {
        ...
    }

    private proc cmp_tree {option obj1 obj2} {
        ...
    }
}
```

If you don’t specify a protection level, everything is public by default, including your variables. This makes namespaces backward-compatible with the rest of Tcl/Tk, but it also makes them different from classes. In classes, methods are public by default, but variables are protected.

Namespaces are also a little different when it comes to protected elements. In a class, protected elements can be accessed in any derived class. But there is no “derived” namespace. The closest equivalent is a nested namespace. If you create a protected element in one namespace, you can access the element in any of the other namespaces nested within it. You might create a protected variable in a namespace like `filestuff` and share it among the namespaces like `filebrowser` and `diskusage` nested within it.

On the other hand, a private element is completely private to the namespace that contains it. If you create a private variable in `filestuff`, it will not show up in any other context, including nested namespaces like `filebrowser` and `diskusage`.

Using Classes and Namespaces

There are some strong similarities between classes and namespaces, but they play different roles in your application. Classes are data structures. They let you create objects to represent the data in your application. For example, we used `VisualFileTree` objects to represent each of the files in our file browser. On the other hand, namespaces are a way of organizing things. We used the `filebrowser` namespace to wrap up the variables and procedures for our file browser library. There is one variable `roots` and one procedure `load_dir` for the file browser, but instead of floating around at the global scope, they are grouped together in the `filebrowser` namespace.

You can use namespaces to organize classes. For example, we grouped `Tree`, `FileTree` and `VisualFileTree` into the `filestuff` namespace. Again, instead of floating around at the global scope, these classes reside with the rest of the file browser library, where they are needed.

You can also use namespaces to organize other namespaces. For example, we grouped the `filebrowser` namespace and the `diskusage` namespace into the same `filestuff` namespace. We can add the `filestuff` library to any of our applications, and access the separate `filebrowser` and `diskusage` utilities within it.

Scoped Commands and Variables

Classes and namespaces are really good at protecting the elements within them. But suppose you want something to be private or protected, but there is one other class—or perhaps one other object—that needs to have access to it. This may be a completely separate class with no inheritance relationship, so we can't rely on "protected" access to solve the problem. And we don't want to open things up for "public" access. In C++, you can declare certain classes and functions as *friends*, thereby granting them special access privileges. In [INCR TCL], we handle this in a different manner, but the effect is the same.

You can see the problem more clearly in the following example. Suppose we have a `folder::create` procedure that creates a checkbutton with an associated file folder icon. We might use this procedure like this:

```
set counter 0
foreach dir {/usr/man /usr/local/man /usr/X11/man} {
    set name ".dir[incr counter]"
    folder::create $name $dir
    pack $name -fill x
}
```

to create the checkbuttons shown in Figure 1-14. When you toggle one of these checkbuttons, it changes the indicator box, and it also opens or closes the folder icon.

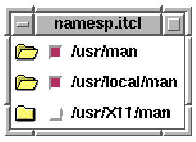


Figure 1-14 Some checkbuttons created by folder::create.

The `folder::create` procedure is shown in Example 1-20. Each time we call it, we create a frame with a label and a checkbutton. Each checkbutton needs a variable to keep track of its state. If we use an ordinary global variable, it might conflict with other variables in the application. Instead, we create a `modes` variable inside the `folder` namespace, and we make it private so that no one else can tamper with it. We treat this variable as an array, and we give each folder assembly a different slot within it. Whenever the checkbutton is invoked, it toggles this variable and calls the `redisplay` procedure to update the icon.

Example 1-20 Using the code and scope commands to share command and variable references.

```
namespace folder {
    private variable images
    set images(open)   [image create photo -file dir1.gif]
    set images(closed) [image create photo -file dir2.gif]

    private variable modes

    public proc create {win name} {
        frame $win
        label $win.icon
        pack $win.icon -side left

        checkbutton $win.toggle -text $name \
            -onvalue "open" -offvalue "closed" \
            -variable [scope modes($win)] \
            -command [code redisplay $win]

        pack $win.toggle -side left -fill x
        $win.toggle invoke
    }

    public proc get {win} {
        global modes
        return $modes($win)
    }

    private proc redisplay {win} {
        global modes images
        set state $modes($win)
        $win.icon configure -image $images($state)
    }
}
```

The checkbutton is clearly a key player in the `folder` library. We want it to have access to the `modes` variable and to the `redisplay` procedure, but we also want to keep these things private. No one else should really be using them. Unless we do something special, the checkbutton will be treated as an outsider and it will be denied access to these elements.

The problem is that options like `-command` and `-variable` are being set inside the `folder` namespace, but they are not evaluated until much later in the program. It is not until you click on a checkbutton that it toggles the variable and invokes the command. This happens in another context, long after we have left the `folder::create` procedure.

There are two commands that let you export part of a namespace to a friend. The `scope` command lets you export a variable reference, and the `code` command lets you export a code fragment. Both of these commands are used on a case-by-case basis. When we create the checkbutton and set the `-variable` option, for example, we enclosed the `modes` variable in the `scope` command. This gives the checkbutton access to just this variable.† If we set the `-variable` option to a different variable name, it will lose access to the `modes` variable. Similarly, when we set the `-command` option, we enclosed the code fragment in the `code` command. This lets the checkbutton execute the `redisplay` command. But if we set the `-command` option to something else, again, it will lose access to `redisplay`.

The `code` and `scope` commands work by capturing the namespace context. They preserve it in such a way that it can be revived again later. So when the checkbutton needs to access its variable, it actually jumps back into the `folder` namespace and looks for the `modes` variable. When the checkbutton needs to invoke its command, again, it jumps back into the `folder` namespace and looks for the `redisplay` command. Since it accesses things from within the `folder` namespace, it by-passes the usual protection levels. In effect, we have given the checkbutton a “back door” into the namespace.

You can see how this works if you query back the actual `-command` or `-variable` string that the checkbutton is using. For example, we created the checkbutton with a command like this:

```
checkbutton $win.toggle ... -command [code redisplay $win]
```

But if we query back the `-command` string, it will look like this:

```
@scope ::folder {redisplay .dir1}
```

† Actually, to just one slot in the array.

This string is the result of the `code` command, and is called a *scoped value*. It is really just a list with three elements: the `@scope` keyword, a namespace context, and a value string. If this string is executed as a command, it automatically revives the `::folder` namespace, and then executes the code fragment "`redisplay .dir1`" in that context.

Note that the `code` command does not execute the code itself. It merely formats the command so that it can be executed later. We can think of `[code ... ]` as a new way of quoting Tcl command strings.

When the `code` command has multiple arguments, they are formatted as a Tcl list and the resulting string becomes the "value" part of the scoped value. For example, if you execute the following command in the `folder` namespace:

```
set cmd [code $win.toggle configure -text "empty folder"]
```

it produces a scoped value like this:

```
@scope ::folder {.dir1.toggle configure -text {empty folder}}
```

Notice how the string "empty folder" is preserved as a single list element. If it were not, the command would fail when it is later executed.

The `code` command can also be used to wrap up an entire command script like this:

```
bind $win.icon <ButtonPress-1> [code "
    $win.toggle flash
    $win.toggle invoke
"]
```

In this case, we combined two commands into one argument. There are no extra arguments, so the code paragraph simply becomes the "value" part of the scoped value that is produced.

The `scope` command works the same way as the `code` command, except that it takes only one argument, the variable name. For example, we created the checkbutton like this:

```
checkbutton $win.toggle ... -variable [scope modes($win)]
```

But if we query back the `-value` string, it will look like this:

```
@scope ::folder modes(.dir1)
```

This entire string represents a single variable name. If we try to get or set this variable, the `@scope` directive shifts us into the `folder` namespace, and looks for a variable named `modes` in that context.

If you forget to use the code and scope commands, you'll get the normal Tk behavior—your commands and variables will be handled in the global context. For example, if we created the checkbutton like this:

```
checkbutton $win.toggle -text $name \
    -onvalue "open" -offvalue "closed" \
    -variable modes($win) \
    -command "redisplay $win"
```

then it would look for a variable named modes in the global namespace, and it would try to execute a command called redisplay in the global context. In some cases this is okay, but more often than not you will need to use the code and scope commands to get things working properly.

You should use the code and scope commands whenever you are handing off a reference to something inside of a namespace. Use the code command with configuration options like -command, -xscrollcommand, -yscrollcommand, *etc*., and with Tk commands like bind, after and fileevent. Use the scope command with options like -variable and -textvariable, and with Tk commands like "tkwait variable".

But although you should use these commands, you should not abuse them. They undermine a key feature of object-oriented programming: encapsulation. If you use these commands to break into a class or a namespace where you don't belong, you will pay for it later. At some point, details inside the class or the namespace may change, and your code will break miserably.

Interactive Development

[INCR TCL] has many features that support debugging and interactive development. Each class has a built-in info method that returns information about an object. So you can query things like an object's class or its list of methods on the fly. This is not possible in C++, but it is quite natural in a dynamic language like Tcl.

Suppose we have defined classes like Tree and FileTree, and we create a FileTree object by typing the following command at the "%" prompt:

```
% FileTree henry /usr/local -procreate "FileTree #auto"
henry
```

We get the result henry which tells us that an object was created successfully.

If someone hands us this object and we want to determine its class, we can use the "info class" query:

```
% henry info class
FileTree
```

This says that `henry` was created as a `FileTree` object, so its most-specific class is `FileTree`. You can get a list of all the classes that `henry` belongs to using the "`info heritage`" query:

```
% henry info heritage
FileTree Tree
```

This says that first and foremost, `henry` is a `FileTree`, but it is also a `Tree`. The classes are visited in this order whenever a method or a variable reference needs to be resolved.

When you want to know if an object belongs to a certain class, you can check its heritage. You can also use the built-in `isa` method to check for base classes. You give `isa` a class name, and it returns non-zero if the class can be found in the object's heritage. For example:

```
% henry isa Tree
1
% henry isa VisualRep
0
```

This says that `henry` belongs to class `Tree`, but not to class `VisualRep`.

The "`info function`" query returns the list of class methods and procs. This includes the built-in methods like `configure`, `cget` and `isa` as well:

```
% henry info function
FileTree::populate FileTree::contents FileTree::constructor Tree::configure
Tree::reorder Tree::cget Tree::isa Tree::constructor Tree::destructor
Tree::add Tree::back Tree::parent Tree::contents Tree::clear
```

Each function is reported with its full name, like `Tree::add`. This helps clarify things if you inherit methods from a base class. You can retrieve more detailed information if you ask for a particular function:

```
% henry info function contents
public method FileTree::contents {} {
    populate
    return [Tree::contents]
}
```

The "`info variable`" query returns the list of variables, which includes all instance variables and common variables defined in the class, as well as the built-in `this` variable:

```
% henry info variable
FileTree::mtime FileTree::file FileTree::this FileTree::procreate
Tree::lastSort Tree::sort Tree::children Tree::value Tree::name Tree::parent
```

Again, you can retrieve more detailed information if you ask for a particular variable:

```
% henry info variable mtime
private variable FileTree::mtime 0 0
```

The last two elements represent the initial value and the current value of the variable. In this case, they are both `0`. But suppose we query the contents of the file tree like this:

```
% henry contents
fileTree0 fileTree1 fileTree2 fileTree3 fileTree4 fileTree5 fileTree6
fileTree7 fileTree8 fileTree9 fileTree10 fileTree11 fileTree12 fileTree13
fileTree14 fileTree15
```

The populate method creates a series of child nodes, and saves the modification time for this directory in the `mtime` variable, as a reminder that the file system has been checked. If we query `mtime` again, we can see that it has changed:

```
% henry info variable mtime
private variable FileTree::mtime 0 845584013
```

You can obtain other high-level information via the usual Tcl `info` command. You can ask for the list of classes in the current namespace like this:

```
% info classes
VisualFileTree FileTree Tree VisualRep
```

and for the list of objects in the current namespace like this:

```
% info objects
fileTree11 fileTree2 fileTree7 fileTree9 fileTree12 fileTree1 fileTree6
fileTree15 henry fileTree13 fileTree3 fileTree14 fileTree0 fileTree5
fileTree8 fileTree10 fileTree4
```

This introspection facility is extremely useful for debugging, and it could support the construction of a class browser or an interactive development environment.

As you are testing your code and finding bugs, you may want to fix things in a class. You can use the `body` command to redefine the body of any method or proc. You can also use the `configbody` command to change the configuration code for a public variable.

This is particularly easy to do in the "`tcl-mode`" of the Emacs editor. You simply load an [INCR TCL] script into Emacs, and tell Emacs to run it. As you are testing it and finding bugs, you can make changes to your script and test them out immediately. You don't have to shut down and start over. Bodies can be changed on the fly. You simply highlight a new `body` or `configbody` definition and tell Emacs to send it off to the test program.

If you don't use Emacs, you can keep your body definitions in a separate file, and you can use the Tcl `source` command to load them into a test program again and again, as bugs are found and corrected.

Although the bodies may change, the class interface cannot be defined more than once. This prevents collisions that would otherwise occur if two developers chose the same class name by accident. But you can delete a class like this:

```
delete class Tree
```

This deletes all objects that belong to the class, all derived classes which depend on this class, and then deletes the class itself. At that point, you can source in your script to redefine the class, and continue debugging.

Autoloading

Tcl provides a way to create libraries of procedures that can be loaded as needed in an application. This facility is called *autoloading*, and it is supported by [INCR TCL] as well.

To use a class library that has been set up for autoloading, you simply add the name of the directory containing the library to the `auto_path` variable:

```
lappend auto_path /usr/local/oreilly/itcl/lib
```

The first time that a class is referenced in a command like this:

```
Tree henry -name "Henry Fonda"
```

the class definition is loaded automatically. The autoloading mechanism searches each directory in the `auto_path` list for a special *tclIndex* file. This file contains a list of commands defined in the directory, along with the script file that should be loaded to define each command. When a command like `Tree` is found in one of the *tclIndex* files, it is automatically loaded, and the command is executed. The next time that this command is needed, it is ready to use.

To create an autoloadable class library, you simply create a directory containing all of the code for the library. Put each class definition in a separate file. These files typically have the extension "*.itcl*" or "*.itk*", but any naming convention can be used. Finally, generate a *tclIndex* file for the directory using the `auto_mkindex` command like this:

```
auto_mkindex /usr/local/oreilly/itcl/lib *.itcl
```

This scans all of the files matching the pattern "**.itcl*" in the directory */usr/local/oreilly/itcl/lib* and creates a *tclIndex* file in that directory. Once the index file is in place, the library is ready to use. Of course, the index file should be regenerated whenever the source code for the library changes.

Adding C code to [INCR TCL] Classes

With a little extra C code, we can extend the Tcl/Tk system to have new commands and capabilities.† This is easy to do, and it is one area where Tcl/Tk outshines other packages. C code can also be integrated into [INCR TCL] classes, to implement the bodies of class methods and procs.

For example, suppose we write a C implementation for the `add` method in our `Tree` class, shown in Example 1-21. Instead of specifying the body as a Tcl script, we use the name `@tree-add`. The leading "`@`" sign indicates that this is the symbolic name for a C procedure.

Example 1-21 Tree class with a C implementation for the "add" method.

```
class Tree {
    variable parent ""
    variable children ""

    method add {obj} @tree-add

    method clear {} {
        if {$children != ""} {
            eval delete object $children
        }
        set children ""
    }
    method parent {pobj} {
        set parent $pobj
    }

    method contents {} {
        return $children
    }
}
```

Somewhere down in the C code for our `wish` executable, we have a Tcl-style command handler for the `add` method. We must give the command handler a symbolic name by registering it with the `Itcl_RegisterC` procedure. We do this in the `Tcl_AppInit` procedure, which is called automatically each time the `wish` executable starts up. You can find the `Tcl_AppInit` procedure in the standard Tcl/Tk distribution, in a file called *tclAppInit.c* (for building `tclsh`) or *tkAppInit.c* (for building `wish`). Near the bottom of this procedure, we add a few lines of code like this:

```
    if (Itcl_RegisterC(interp, "tree-add", Tree_AddCmd) != TCL_OK) {
        return TCL_ERROR;
    }
```

This gives the symbolic name "`tree-add`" to the C procedure `Tree_AddCmd`. This procedure will be called to handle any class method or class proc that has the body "`@tree-add`".

† For details, see John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

Example 1-22 shows the implementation for the Tree_AddCmd procedure. It takes the usual arguments for a Tcl-style command handler: The first argument is required but not used; interp is the interpreter handling a Tcl command; argc is the number of arguments on the Tcl command line; and argv is the list of Tcl argument strings.

Example 1-22 Implementation for the Tree_AddCmd handler.

```
#include <tcl.h>
int
Tree_AddCmd(dummy, interp, argc, argv)
    ClientData dummy;         /* unused */
    Tcl_Interp *interp;       /* current interpreter */
    int argc;                 /* number of arguments */
    char **argv;              /* argument strings */
{
    char *val;
    Tcl_DString buffer;

    if (argc != 2) {
        Tcl_AppendResult(interp, "wrong # args: should be \"",
            argv[0], " treeObj\"", (char*)NULL);
        return TCL_ERROR;
    }

    /*
     *  Build a command string like "treeObj parent $this" and
     *  execute it.
     */
    Tcl_DStringInit(&buffer);
    val = Tcl_GetVar(interp, "this", TCL_LEAVE_ERR_MSG);
    if (val == NULL) {
        Tcl_DStringFree(&buffer);
        return TCL_ERROR;
    }
    Tcl_DStringAppendElement(&buffer, argv[1]);
    Tcl_DStringAppendElement(&buffer, "parent");
    Tcl_DStringAppendElement(&buffer, val);
    val = Tcl_DStringValue(&buffer);

    if (Tcl_Eval(interp,val) != TCL_OK) {
        Tcl_DStringFree(&buffer);
        return TCL_ERROR;
    }
    Tcl_ResetResult(interp);

    /*
     *  Add the specified object to the "children" list.
     */
    val = Tcl_SetVar(interp, "children", argv[1],
        TCL_LEAVE_ERR_MSG | TCL_LIST_ELEMENT | TCL_APPEND_VALUE);

    if (val == NULL) {
        Tcl_DStringFree(&buffer);
        return TCL_ERROR;
    }

    Tcl_DStringFree(&buffer);
    return TCL_OK;
}
```

This procedure has to mimic our add method. It takes the name of another Tree object, and adds it to the list of children for the current node. Whenever

Tree_AddCmd is called, therefore, we should have two argument strings: the command name “add” (stored in argv[0]), and the name of the child object (stored in argv[1]). We first check to make sure that this is true, and if not, we immediately return an error.

Next, we build the command string “$obj parent $this” in a dynamic string buffer. This command notifies the child that it has a new parent. We query the value of the this variable using Tcl_GetVar. We build the command string in a Tcl_DString buffer, and then use Tcl_Eval to execute the command.

The name of the child object is then appended to the children list using Tcl_SetVar.

This implementation is identical to the Tcl version shown in Example 1-3, although it requires many more C language statements to perform the same task. In this case, the result is no better. The C version is not much faster, and the Tcl version was considerably easier to write.

But the interesting part of this example is the interface between the C code and the [INCR TCL] class. When the command handler is executed, class variables can be accessed as ordinary variables. Class methods can be invoked as ordinary commands. [INCR TCL] handles this automatically by setting up the object context before the handler is invoked. Because of this, we were able to access the children variable and the built-in this variable with ordinary Tcl_GetVar and Tcl_SetVar calls.

Therefore, a single class can have some parts written in C code, and others written in Tcl. The Tcl parts can be migrated to C for better performance as the need arises.

Tcl is an excellent “glue” language. It stitches C code blocks together with Tcl statements to form applications. [INCR TCL] takes the glue to a higher-level. Bits of Tcl and C code can be mixed together to create classes. These high-level building blocks provide better support for building larger applications.

Summary

Extension:	[incr Tcl] - Object-Oriented Programming for Tcl
Author:	Michael J. McLennan Bell Labs Innovations for Lucent Technologies mmclennan@lucent.com
Other Contributors:	Jim Ingham Lee Bernhard ...and many others listed on the web site
Platforms Supported:	All major Unix platforms Linux Windows 95 (release itcl2.2 and beyond) Macintosh (release itcl2.2 and beyond)
Web Site:	`http://www.tcltk.com/itcl`

Mailing List: (bug reports)	`mail -s "subscribe" itcl-request@tcltk.com`
	to subscribe to the mailing list
	`mail itcl@tcltk.com`
	to send mail

Quick Reference

Classes

```
class className {
    inherit baseClass ?baseClass...?

    constructor args ?init? body
    destructor body

    method name ?args? ?body?
    proc name ?args? ?body?

    variable varName ?init? ?configBody?
    common varName ?init?

    set varName ?value?
    array option ?arg arg ...?

    public command ?arg arg ...?
    protected command ?arg arg ...?
    private command ?arg arg ...?
}
```

Defines a new class of objects.

`body`	`className::function args body` — Redefines the body for a class method or proc.
`configbody`	`className::varName body` — Redefines the body of configuration code for a public variable or a mega-widget option.
`delete`	`class name ?name...?` — Deletes a class definition and all objects in the class
`info`	`classes ?pattern?` — Returns a list of all classes, or a list of classes whose names match *pattern*.

Objects

`className`	`objName ?arg arg ...?` — Creates an object that belongs to class *className*.

`objName` `method ?arg arg ...?`

Invokes a method to manipulate an object.

`delete` `object objName ?objName...?`

Deletes one or more objects.

`info` `objects ?-class className? ?-isa className? ?pattern?`

Returns a list of all objects, or a list of objects in a certain class *className*, whose names match *pattern*.

Namespaces

```
namespace namespaceName {
    variable varName ?value?
    proc cmdName args body

    private command ?arg arg ...?
    protected command ?arg arg ...?
    public command ?arg arg ...?

    command ?arg arg ...?
}
```

Finds an existing namespace or creates a new namespace and executes a body of commands in that context. Commands like `proc` and `variable` create Tcl commands and variables that are local to that namespace context.

```
namespaceName::cmdName ?arg arg ...?
namespaceName::namespaceName::...::cmdName ?arg arg ...?
```

Invokes a procedure that belongs to another namespace.

`code` `command ?arg arg ...?`

Formats a code fragment so it can be used as a callback in another namespace context.

`delete` `namespace namespaceName ?namespaceName...?`

Deletes a namespace and everything in it.

import	add *name* ?*name*...? ?-*where pos*...?
	all ?*name*?
	list ?*importList*?
	remove *name* ?*name*...?
	Changes the import list for a namespace.
info	context
	Returns the current namespace context.
info	namespace all ?*pattern*?
	namespace children ?*name*?
	namespace parent ?*name*?
	Returns information about the namespace hierarchy.
info	namespace qualifiers *string*
	namespace tail *string*
	Parses strings with :: namespace qualifiers.
info	protection ?-command? ?-variable? *name*
	Returns the protection level (public/protected/private) for a command or variable.
info	which ?-command? ?-variable? ?-namespace? *name*
	Searches for a command, variable or namespace and returns its fully-qualified name.
scope	*string*
	Formats a variable name so it can be accessed in another namespace context.

2 Building Mega-Widgets with [incr Tk]

Tk lets you create objects like buttons, labels, entries, and so forth, but it is not truly object-oriented. You can't create a new widget class like `HotButton` and have it inherit its basic behavior from class `Button`. So you really can't extend the Tk widget set unless you tear apart its C code and add some of your own.

[INCR TK] lets you create brand new widgets, using the normal Tk widgets as component parts. These *mega-widgets* look and act like ordinary Tk widgets, but you can create them without writing any C code. Instead, you write an [INCR TCL] class to handle each new type of mega-widget.

If you read Chapter XXX on the [INCR WIDGETS] library, you can see what great results you'll get using [INCR TK]. [INCR WIDGETS] has more than 30 new widget classes including `Fileselectionbox`, `Panedwindow`, `Canvasprintbox`, `Optionmenu` and `Combobox`, and they were all built with the [INCR TK] framework.

You can understand the essence of a mega-widget simply by looking at one of these widgets. For example, the `Spinint` widget shown in Figure 2-1 is created like this:

```
spinint .s -labeltext "Repeat:" -width 5 -range {1 10}
pack .s
```

It has an entry component that holds a numeric value, and a pair of buttons for adjusting that value. Whenever you create a `Spinint` widget, all of these internal components are created and packed automatically. When you set the `-labeltext` option, a label appears. You can set the `-range` option to control the range of integer values. If you use the arrow buttons and bump the number beyond this range, it will wrap around to the other end of the scale.

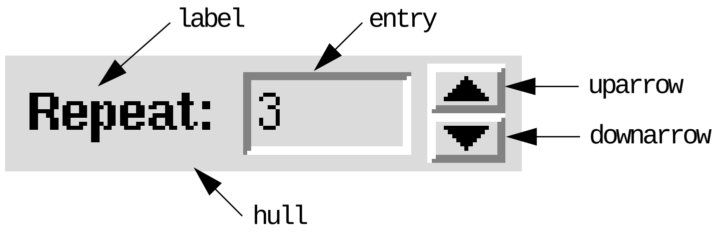


Figure 2-1 A Spinint mega-widget has many component parts.

A `Spinint` can be configured like a normal Tk widget. It has many internal components, but they all work together as one widget. All of their configuration options are merged together into a single list called the *master option list*. When you set master configuration options like this:

```
.s configure -background tan -textbackground white
```

the effects propagate down to all of the internal components. Setting the `-background` option changes the background of the `hull`, `label`, `uparrow` and `downarrow` components. Setting the `-textbackground` option changes the background of the entry component.

A `Spinint` also has options to control the layout of its components. You can rearrange the buttons like this:

```
.s configure -arroworient horizontal
```

and reposition the label like this:

```
.s configure -labelpos nw
```

You can even query the current option settings like this:

```
set bg [.s cget -background]
```

Of course, you can add all of these settings to the options database, so that `Spinint` widgets will have these values by default:

```
option add *Spinint.background tan
option add *Spinint.textBackground white
option add *Spinint.arrowOrient horizontal
option add *Spinint.labelPos nw
```

A `Spinint` widget has a well-defined set of operations or *methods* to manipulate it. You can load a new integer into the text area like this:

```
.s clear
.s insert 0 "10"
```

and you can programmatically bump up the value like this:

```
.s up
```

When you destroy the widget:

```
destroy .s
```

all of its internal components are destroyed automatically.

Mega-widgets have all of the characteristics that we would expect from a Tk widget. But since they do not require any C code or X library programming, they are considerably easier to implement.

Overview

To understand [INCR TK], you have to understand how a mega-widget handles its component parts and their configuration options. In this section, we'll explore [INCR TK] from a conceptual standpoint. Later on, we'll look at real code examples.

Class Hierarchy

To create a new type of mega-widget, you simply derive a new [INCR TCL] class from one of the existing [INCR TK] base classes. The [INCR TK] class hierarchy is shown in Figure 2-2. All of these classes reside in the `itk` namespace, so they will not interfere with the rest of your application.

There are basically two different kinds of mega-widgets, so there are two [INCR TK] base classes that you use to build them. If you want a mega-widget to pop up in its own toplevel window, then have it inherit from `itk::Toplevel`. This lets you build dialog widgets like the `Fileselectiondialog`, `Messagedialog`, and `Canvasprintdialog` in the [INCR WIDGETS] library. Otherwise, if you want a mega-widget to sit inside of some other toplevel window, then have it inherit from the `itk::Widget` class. This lets you build things like the `Optionmenu`, `Combobox` and `Panedwindow` in the [INCR WIDGETS] library.

Suppose we were starting from scratch to create the `Spinint` class. `Spinint` widgets are the kind that sit inside of other toplevel windows, so we should use the `itk::Widget` class as a starting point.

Both `itk::Widget` and `itk::Toplevel` inherit the basic mega-widget behavior from `itk::Archetype`. This class keeps track of the mega-widget components and their configuration options.

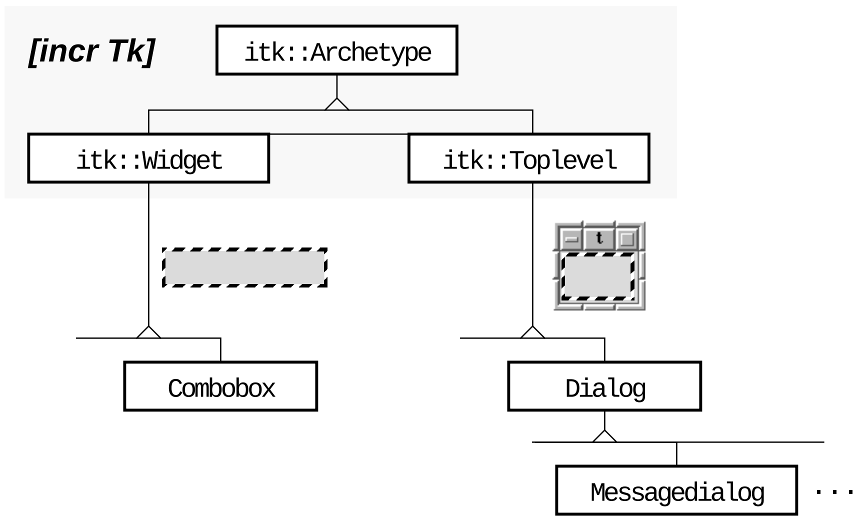


Figure 2-2 Mega-widgets are created by extending one of the base classes in [incr Tk].

Class Definition

If we wanted to implement the `Spinint` widget, we would write a class definition that looks something like the one shown in Figure 2-3.†

Notice that we use a class name like `Spinint` that starts with a capital letter. This is a rule in Tk. For the time being, you can assume that we also have to create mega-widgets with a capitalized command like this:

```
Spinint .s -labeltext "Repeat:" -width 5 -range {1 10}
```

Later on, I will show you how to get around this.

Inside the class definition, we start off with an `inherit` statement that brings in the `itk::Widget` base class. As we will see below, this automatically gives us a container for the mega-widget called the *hull*. We write a constructor to create all of the component widgets and pack them into the hull. Instead of including the actual code, we simply illustrated this process in the constructor shown in Figure 2-3.

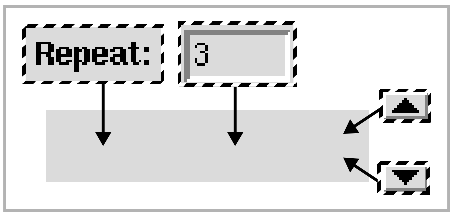
Notice that the constructor uses the `args` argument to handle any configuration options that might be specified when a widget is created, like this for example:

```
Spinint .s -labeltext "Number of Copies:" -background red
```

† The `Spinint` class in the [INCR WIDGETS] library is a little more complicated, but this example will give you the basic idea.

```
class Spinint {
    inherit itk::Widget

    constructor {args} {
```



```
        eval itk_initialize $args
    }

    public method clear {}
    public method insert {index value}

    public method up {}
    public method down {}
}
```

Figure 2-3 Conceptual view of Spinint mega-widget class.

But instead of handling these arguments with:

```
eval configure $args
```

as we would for an ordinary [INCR TCL] class, we use:

```
eval itk_initialize $args
```

You must call `itk_initialize` instead of `configure` for all of your [INCR TK] mega-widgets. This is a protected method that belongs to the `itk::Archetype` base class. It not only applies the configuration changes, but it also makes sure that all mega-widget options are properly initialized. If you forget to call it for a particular class, some of the configuration options may be missing whenever you create a mega-widget of that class.

Near the bottom of the class definition, we include some methods to handle the operations for this mega-widget. As I said before, you can load a new value into a `Spinint` widget like this:

```
.s clear
.s insert 0 "10"
```

So we have a method `clear` to clear the entry, and a method `insert` to insert some new text. We also have a method called `up` to increment the value, and a method called `down` to decrement it. We can add more operations to the `Spinint` class simply by defining more methods.

Notice that we didn't define a destructor. The `itk::Archetype` base class keeps track of the component widgets and destroys them for you when the mega-widget is destroyed. You won't need a destructor unless you have to close a file or delete some other object when the mega-widget is destroyed.

Mega-Widget Construction

Let's take a moment to see what happens when a mega-widget is constructed. For example, suppose we create a `Spinint` widget like this:

```
Spinint .s -labeltext "Starting Page:" -range {1 67}
```

When [INCR TCL] sees this command, it creates an object named `.s` in class `Spinint`, and calls its constructor with the remaining arguments. But before it can actually run the `Spinint::constructor`, all of the base classes must be fully constructed. This process is illustrated in Figure 2-4.

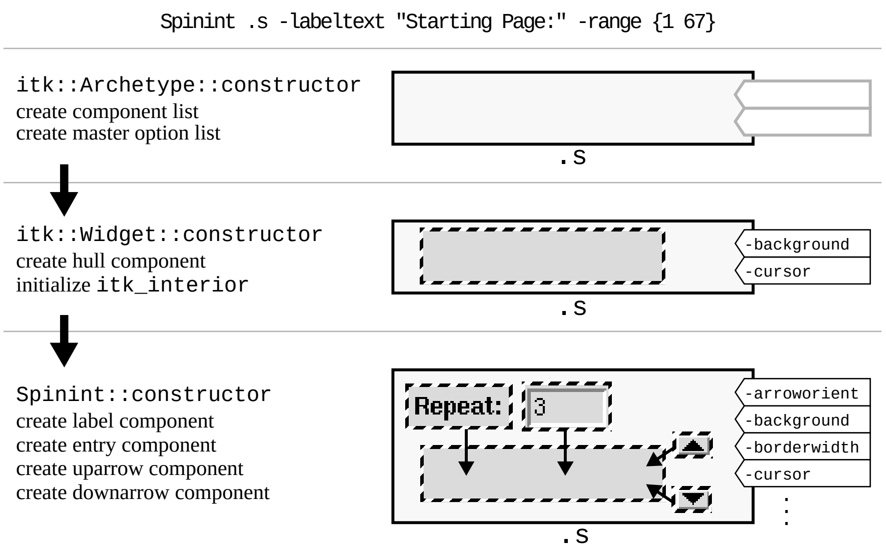


Figure 2-4 Construction of a Spinint mega-widget.

The constructor for the least-specific class `itk::Archetype` is called first. It initializes some internal variables that keep track of the component widgets and their configuration options. Next, the `itk::Widget` constructor is called. It creates the hull frame that acts as a container for the component widgets. The name of this frame widget is stored in a protected variable called `itk_interior`. We will use this name later on as the root for component widget names. Finally, the `Spinint` constructor is called. It creates the `label`, `entry` and `uparrow` and `downarrow` components, and packs them into the hull.

As each component is created, its configuration options are merged into a master list of options for the mega-widget. We will see precisely how this is done in the next section. But we end up with a mega-widget that has an overall list of configuration options. Near the end of the `Spinint` constructor, we call `itk_initialize` to finalize the list and apply any configuration changes.

Creating Component Widgets

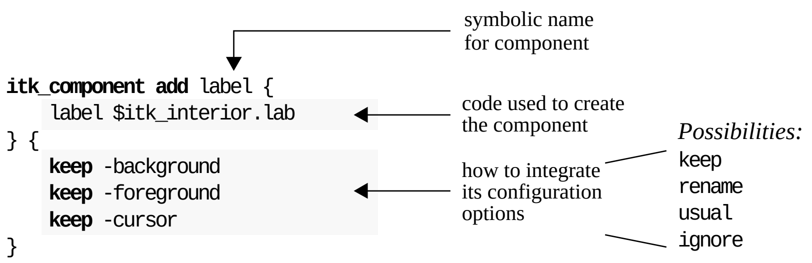
Let's look inside the constructor now and see how we create each of the mega-widget components. Normally, when we create a Tk widget, we use a simple command like this:

```
label .s.lab
```

This says that we have a frame called `.s` and we want to put a label named `lab` inside it. For a mega-widget, we can't hard-code the name of the containing frame. It will be different for each widget that gets created. If we create a `Spinint` named `.s`, it will have a hull named `.s`, and the label should be called `.s.lab`. But if we create a `Spinint` named `.foo.bar`, it will have a hull named `.foo.bar`, and the label should be called `.foo.bar.lab`. Instead of hard-coding the name of a frame, we use the name in the `itk_interior` variable, like this:

```
label $itk_interior.lab
```

We also have to do something special to let the mega-widget know that this is a component. We wrap the widget creation command inside an `itk_component` command like the one shown in Figure 2-5.



```
itk_component add label {
    label $itk_interior.lab
} {
    keep -background
    keep -foreground
    keep -cursor
}
```

symbolic name for component
code used to create the component
how to integrate its configuration options
Possibilities:
keep
rename
usual
ignore

Figure 2-5 Syntax of the itk_component command.

This command executes the code that you give it to create the component, and saves the name of the resulting widget. It stores this name in a protected array called `itk_component`, using the symbolic name as an index. When you want to refer to the component later on, you can look it up in this array using its symbolic name. For example, in Figure 2-5 we created a label with the symbolic name `label`. We can pack this component using its symbolic name, like this:

```
pack $itk_component(label) -side left
```

The expression `$itk_component(label)` expands to a real widget path name like `.s.lab` or `.foo.bar.lab`. We can use this in any of the methods in the `Spinint` class to refer to the label component.

You can also use symbolic component names outside of the mega-widget class, but you do it a little differently. The `itk::Archetype` class provides a method called `component` that you can use to access components. If you call this method without any arguments:

```
% Spinint .s
.s
% .s component
hull label entry uparrow downarrow
```

it returns a list of symbolic component names. You can also use this method to reach inside the mega-widget and talk directly to a particular component. For example, we might configure the label to have a sunken border like this:

```
.s component label configure -borderwidth 2 -relief sunken
```

Using symbolic names insulates you from the details inside of a mega-widget class. Suppose we decide next week to rearrange the components, and we change the name of the actual label widget from `$itk_interior.lab` to `$itk_interior.box.l1`. Code inside the class like:

```
pack $itk_component(label) -side left
```

and code outside the class like:

```
.s component label configure -borderwidth 2 -relief sunken
```

will not have to change, since we used the symbolic name in both places.

The `itk_component` command does one other important thing. As you add each component, its configuration options are merged into the master list of options for the mega-widget. When you set a master option on the mega-widget, it affects all of the internal components. When you set the master `-back-ground` option, for example, the change is propagated to the `-background` option of the internal components, so the entire background changes all at once.

You can control precisely how the options are merged into the master list by using a series of commands at the end of the `itk_component` command. We will explain all of the possibilities in greater detail below, but in Figure 2-5 we used the `keep` command to merge the `-background`, `-foreground` and `-cursor` options for the label into the master list.

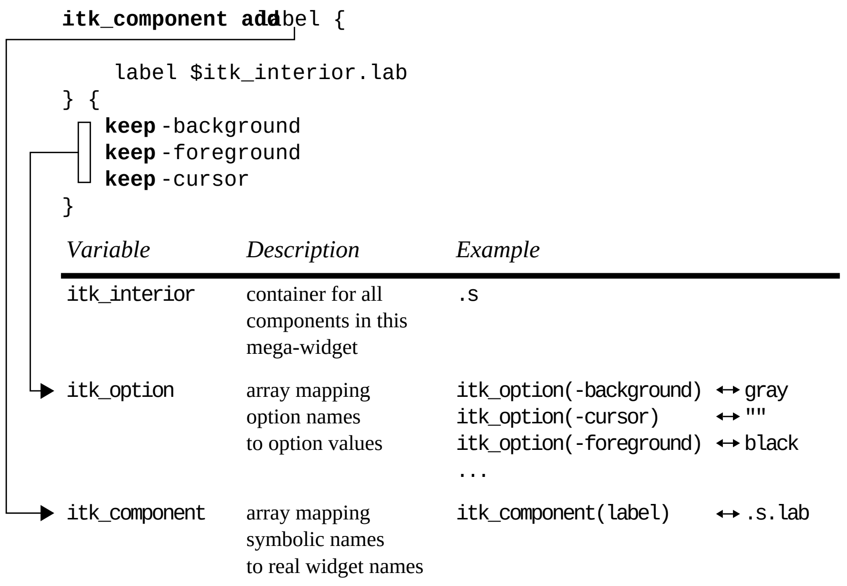
All of the master configuration options are kept in a protected array called `itk_option`. You can use this in any of the methods to get the current value for a configuration option. It will save you a call to the usual `cget` method. For example, if we were in some method like `Spinint::insert`, we could find out the current background color using either of these commands:

```
set bg [cget -background]         ;# a little slow
set bg $itk_option(-background)   ;# better
```

But if you want to change an option, you can't just set the value in this array. You must always call the `configure` method, as shown below:

```
set itk_option(-background) red    ;# error!  color does not change
configure -background red          ;# ok
```

As you can see, there is a close relationship between the `itk_component` command, and the `itk_component` and `itk_option` arrays. Whenever you add a new component, its symbolic name is added to the `itk_component` array, and its configuration options are merged into the `itk_option` array. This relationship is summarized in Figure 2-6.



```
itk_component add label {
    label $itk_interior.lab
} {
    keep -background
    keep -foreground
    keep -cursor
}
```

Variable	*Description*	*Example*
itk_interior	container for all components in this mega-widget	.s
itk_option	array mapping option names to option values	itk_option(-background) ↔ gray itk_option(-cursor) ↔ "" itk_option(-foreground) ↔ black ...
itk_component	array mapping symbolic names to real widget names	itk_component(label) ↔ .s.lab

Figure 2-6 How the itk_component command ties in with class variables.

Keeping Configuration Options

Each mega-widget has a master list of configuration options. When you set a master option, it affects all of the internal components that are tied to that option. For example, if we have a `Spinint` mega-widget named `.s` and we configure its master `-background` option:

```
.s configure -background green
```

the change is automatically propagated down to the `hull`, `label`, `uparrow`, and `downarrow` components. In effect, the overall background turns green with one simple command. This is what you would naively expect, since a mega-widget is supposed to work like any other Tk widget. But [INCR TK] has special machinery under the hood that allows this to take place.

When you create a component widget, you can specify how its configuration options should be merged into the master list. One possibility is to add component options to the master list using the `keep` command. When you keep an option, it appears on the master list with the same name. For example, in Figure 2-7 we show two different `Spinint` components being created. The `label` component keeps its `-background`, `-foreground` and `-cursor` options, so these options are added directly to the master list. The `entry` component keeps these same options, and keeps the `-borderwidth` option as well.

```
itk_component add label {
    label $itk_interior.lab
} {
    keep -background
    keep -foreground
    keep -cursor
}
```

```
itk_component add entry {
    entry $itk_interior.ent
} {
    keep -background
    keep -foreground
    keep -cursor
    keep -borderwidth
}
```

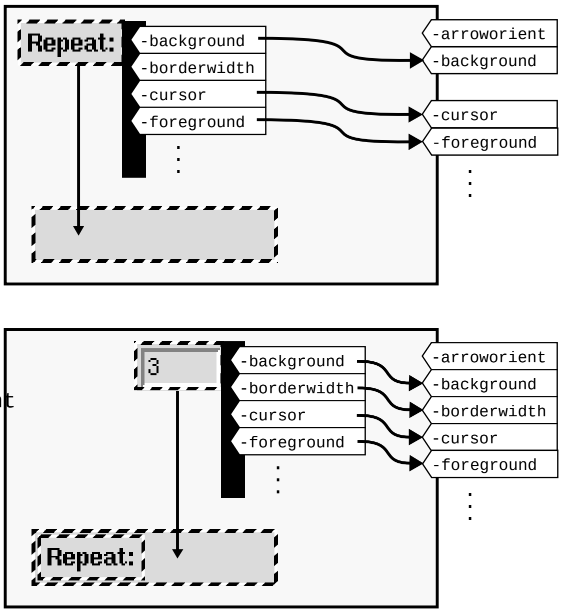


Figure 2-7 Keeping component options on the master option list.

When we configure a master option for the mega-widget, the change is propagated down to all of the components that kept the option. This process is shown in Figure 2-8.

```
.s configure -background tan -borderwidth 2
```

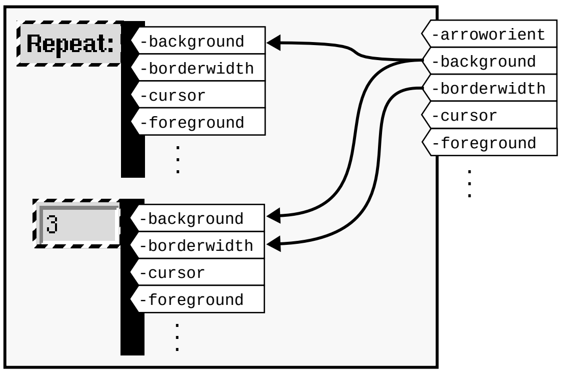


Figure 2-8 Configuration changes are propagated down to component widgets.

When we configure the `-background` option, both the label and the entry are updated, but when we configure `-borderwidth`, only the entry is updated. Since we did not keep `-borderwidth` for the label, it is not affected by a change in border width.

You must include a `keep` statement for each of the component options that you want to access on the master list. The rest of the component options will be ignored by default. It is usually a good idea to keep options like `-background`, `-foreground`, `-font` and `-cursor`, which should be synchronized across all components in the mega-widget. Options like `-text` or `-command`, which are different for different components, should be ignored.

Renaming Configuration Options

Suppose you want to keep an option on the master list, but you want to give it a different name. For example, suppose you want to have an option named `-textbackground` for the `Spinint` mega-widget that changes the background color of the entry component. Having a separate option like this would let you highlight the entry field with a contrasting color, so that it stands out from the rest of the mega-widget. We want to keep the `-background` option for the entry component, but we want to tie it to a master option with the name `-textbackground`. We can handle this with a `rename` command like the one shown in Figure 2-9.

```
itk_component add entry {
    entry $itk_interior.ent
} {
    rename -background -textbackground textBackground Background
    keep -foreground
    keep -cursor
    keep -borderwidth
}
```

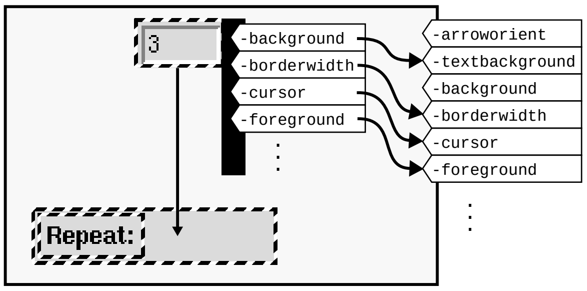


Figure 2-9 Renaming component options on the master option list.

We could create another component and rename its `-background` option to `-textbackground` as well. If we did, then both of these components would be controlled by the master `-textbackground` option. We could even create a component and rename its `-foreground` option to `-textbackground`. Again, any change to a master option like `-textbackground` is propagated down to all of the component options that are tied to it, regardless of their original names.

When you rename an option, you need to specify three different names for the option: an option name for the `configure` command, along with a resource name and a resource class for the options database. In Figure 2-9, we renamed the entry's `-background` option, giving it the name `-textbackground`, the resource name `textBackground`, and the resource class `Background`. Each of these names can be used as follows.

We can use the option name to configure the entry part of a `Spinint` mega-widget like this:

```
.s configure -textbackground white
```

We can use the resource name in the options database to give all of our `Spinint` mega-widgets this value by default:

```
option add *Spinint.textBackground white
```

Instead of setting a specific resource like `textBackground`, we could set a more general resource class like `Background`:

```
    option add *Spinint.Background blue
```

This affects all of the options in class `Background`, including both the regular `-background` option and the `-textbackground` option. In this case, we set both background colors to blue.

"Usual" Configuration Options

If you have to write `keep` and `rename` statements for each component that you create, it becomes a chore. You will find yourself typing the same statements again and again. For a label component, you always keep the `-background`, `-foreground`, `-font` and `-cursor` options. For a button component, you keep these same options, along with `-activebackground`, `-activeforeground` and `-disabledforeground`.

Fortunately, the `keep` and `rename` statements are optional. If you don't include them, each widget class has a default set of `keep` and `rename` statements to fall back on. These defaults are included in the [INCR TK] library directory, and they are called the *usual option-handling code*. You can change the "usual" code or even add "usual" code for new widget classes, as we'll see later on.

You can ask for the "usual" options one of two ways, as shown in Figure 2-10. You can either include the `usual` command in the option-handling commands passed to `itk_component`, or you can leave off the option-handling commands entirely. As you can see, the second way makes your code look much simpler.

```
itk_component add label {
    label $itk_interior.lab
} {
    usual
}
```

or

```
itk_component add label {
    label $itk_interior.lab
}
```

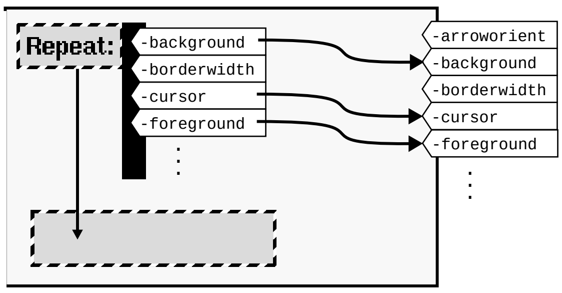


Figure 2-10 Adding a component with the "usual" option-handling code.

Having the `usual` command is useful if you want to have most of the "usual" options, but with a few changes. For example, suppose we are adding the entry component to a `Spinint`. We can get all of the "usual" options, but then override how the `-background` option is handled. We can rename the `-background` option to `-textbackground` like this:

```
itk_component add entry {
    entry $itk_interior.ent
} {
    usual
    rename -background -textbackground textBackground Background
}
```

This is much better than the code shown in Figure 2-9. There are many entry widget options like `-insertbackground` and `-selectbackground` that we had ignored earlier. The "usual" code for an entry handles these properly, without any extra work.

Ignoring Configuration Options

In addition to the `keep`, `rename` and `usual` commands, you can also ask for certain options to be ignored using the `ignore` command. In most cases, this is not really needed. If you include any option-handling code at all, it will start by assuming that all options are ignored unless you explicitly `keep` or `rename` them. But the `ignore` command is useful when you want to override something in the "usual" code.

Suppose the "usual" option-handling code keeps an option like `-foreground`, and you really want that option to be ignored for a particular component. You can use the `usual` command to bring in the "usual" code, and then ignore a particular option like this:

```
itk_component add entry {
    entry $itk_interior.ent
} {
    usual
    ignore -foreground
}
```

Setting Widget Defaults

As we saw earlier, you can establish a default value for any mega-widget option using the options database. For example, suppose we are creating an application, and we set the following resources:

```
option add *Spinint.background blue
option add *Spinint.textBackground white
```

The "`*Spinint`" part says that these values apply to all `Spinint` widgets in the application, regardless of their name or where they appear in the window hierarchy. The "`.background`" and "`.textBackground`" parts access specific resources on each `Spinint` widget.

Remember, a master option like `-background` may be tied to many component widgets that kept or renamed that option. In this case, the `-background` option of a `Spinint` is tied to the `-background` option of the `hull`, `label`, `up` and `down` components. The default value for the `Spinint` background is automatically propagated down to each of these components.

As a mega-widget designer, it is your responsibility to make sure that all of the options in your mega-widget have good default values. It's a good idea to include settings like these just above each mega-widget class:

```
option add *Spinint.textBackground white widgetDefault
option add *Spinint.range "0 100" widgetDefault
option add *Spinint.arrowOrient horizontal widgetDefault
option add *Spinint.labelPos nw widgetDefault
```

All of these settings are given the lowest priority `widgetDefault`, so that you can override them later on. You might add other `option` statements to customize a particular application. On Unix platforms, the user might add similar resource settings to a *.Xdefaults* or *.Xresources* file.

If you don't provide a default value for an option, then its initial value is taken from the component that first created the option. For example, we did not include a default value for the `background` resource in the statements above. If there is no other setting for `background` in the application, then the default value will be taken from the `hull` component, which was the first to keep the `-background` option. The `hull` is a frame, and its default background is probably gray, so the default background for the `Spinint` will also be gray. Many times, the default values that come from components work quite well. But when they do not, you should set the default explicitly with an `option` statement.

Simple Example

Now that we understand how the components fit into a mega-widget, we can see how everything works in a real example. In the previous chapter, we saw how [INCR TCL] classes could be used to build a file browser. We wrote classes to handle the file tree and its visual representation. We even wrote a few procedures so that we could install a file tree on any canvas widget.

Now we can take this idea one step further. Instead of grafting our file tree onto an external canvas, we can wrap the canvas and the file tree code into a `Fileviewer` mega-widget. When we are done, we will be able to create a `Fileviewer` like this:

```
Fileviewer .viewer -background LightSlateBlue -troughcolor NavyBlue
pack .viewer -expand yes -fill both -pady 4 -pady 4
```

and have it display a file tree like this:

```
.viewer display /usr/local/lib
```

This will create a widget that looks like the one shown in Figure 2-11. It has a canvas to display the file tree, and a built-in scrollbar to handle scrolling. If you click on a file or a folder, it becomes selected with a gray rectangle. If you double-click on a folder, it expands or collapses the file hierarchy beneath it.

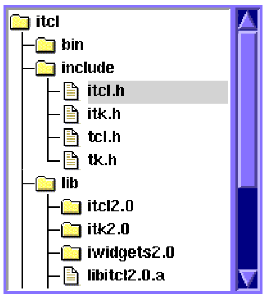


Figure 2-11 Fileviewer mega-widget.

Now, we'll write the `Fileviewer` class.

Fileviewer Construction

A complete code example appears in the file *itcl/itk/fileviewer1.itk*, but the `Fileviewer` class itself is shown below in Example 2-1.

Example 2-1 Class definition for the Fileviewer mega-widget.

```
option add *Fileviewer.width 2i widgetDefault
option add *Fileviewer.height 3i widgetDefault

class Fileviewer {
    inherit itk::Widget

    constructor {args} {
        itk_component add scrollbar {
            scrollbar $itk_interior.sbar -orient vertical \
                -command [code $itk_interior.canv yview]
        }
        pack $itk_component(scrollbar) -side right -fill y

        itk_component add display {
            canvas $itk_interior.canv -borderwidth 2 \
                -relief sunken -background white \
                -yscrollcommand [code $itk_interior.sbar set]
        } {
            keep -cursor -height -width
            keep -highlightcolor -highlightthickness
            rename -highlightbackground -background background Background
```

Example 2-1 Class definition for the Fileviewer mega-widget.

```
        }
        pack $itk_component(display) -side left -expand yes -fill both

        eval itk_initialize $args
    }

    private variable root ""

    public method display {dir}

    private method createNode {dir}
    private proc cmpTree {option obj1 obj2}
}
```

We start off by inheriting the basic mega-widget behavior from `itk::Widget`. This means that the `Fileviewer` will be the kind of widget that sits inside of another toplevel window, so we can use a `Fileviewer` component in many different styles of file selection dialogs.

In the constructor, we create the components within each `Fileviewer`, and pack them into the hull. We create a scrollbar component named `scrollbar` like this:

```
itk_component add scrollbar {
    scrollbar $itk_interior.sbar -orient vertical \
        -command [code $itk_interior.canv yview]
}
```

As we saw in Figure 2-6, we use `$itk_interior` as the root of the component widget name. If we create a `Fileviewer` mega-widget named `.fv`, then `$itk_interior` will also be `.fv`, and the scrollbar will be named `.fv.sbar`.

Since we didn't include any `keep` or `rename` statements, we will get the "usual" option-handling code for scrollbars. This automatically adds options like `-background` and `-troughcolor` to the master options for a `Fileviewer`. The "usual" code ignores options like `-orient` and `-command` that are probably unique to each scrollbar component. We really don't want anyone using a `Fileviewer` to change these options. We just set them once and for all when the scrollbar is first created.

Notice that we used the `code` command to wrap up the code for the `-command` option. This isn't absolutely necessary, but it is a good idea for the reasons that we discussed in the previous chapter. If you do something like this:

```
itk_component add scrollbar {
    scrollbar $itk_interior.sbar -orient vertical \
        -command "$itk_interior.canv yview"
}
```

it will still work, but the scrollbar command will take longer to execute. Each time it tries to talk to the canvas widget, it will start looking for it in the global namespace. Since the canvas is created in the `Fileviewer` constructor, its access command is buried inside of the Fileviewer namespace, and it will take a little longer to find.† The code command wraps up the scrollbar command so that when it is needed later on, it will be executed right in the Fileviewer namespace, so the canvas will be found immediately. Whenever you are configuring a component widget, you should always use a code command to wrap up code fragments for options like -command or -yscrollcommand. Likewise, you should also use a scope command to wrap up variable names for options like -variable.

Once the scrollbar has been created, we can use its symbolic name in the itk_component array to refer to it later on. For example, we pack the scrollbar like this:

```
pack $itk_component(scrollbar) -side right -fill y
```

We create a canvas component called display in a similar manner. But instead of getting the "usual" configuration options, we include explicit keep and rename statements to merge its options into the master list:

```
itk_component add display {
    canvas $itk_interior.canv -borderwidth 2 \
        -relief sunken -background white \
        -yscrollcommand [code $itk_interior.sbar set]
} {
    keep -cursor -height -width
    keep -highlightcolor -highlightthickness
    rename -highlightbackground -background background Background
}
```

You can list all of the options in a single keep statement, or you can include lots of different keep statements. In this case, we used two different keep statements to make the code more readable. We did not keep the -background, -borderwidth or -relief options. We simply fix their values when the canvas is created. If you configure the -background option of a Fileviewer, the rest of the widget will change, but the canvas background is not tied in, so it will always remain white.

Notice that we renamed the -highlightbackground option to -background. Whenever we configure the master -background option, the -highlightbackground option on the canvas component will be updated as well. If you don't do this, you will see a problem as soon as you change the master -background option. Most of the background will change, but the focus highlight rings inside the mega-widget will remain a different color. This rename trick fixes

† If this were an ordinary object, it wouldn't be found at all. But there is some special code in the Tcl unknown proc that finds widgets no matter where they are in the namespace hierarchy.

the problem. It is such a good trick that it is part of the "usual" option-handling code that you normally get by default.

Fileviewer Methods

The Fileviewer class in Example 2-1 has one public method. If we have created a Fileviewer named .viewer, we can tell it to display a certain directory by calling the display method:

```
.viewer display /home/mmc
```

The createNode method and the cmpTree proc are there only to help the display method, so we make them private. We'll see how they are used in a moment.

A Fileviewer mega-widget works just like the file browser that we created in Example 1-14. If you have forgotten all about VisualFileTree objects and how we built the file browser, you should take a moment to remind yourself.

The implementation for the Fileviewer::display method is shown in Example 2-2.

Example 2-2 Implementation for the Fileviewer::display method.

```
body Fileviewer::display {dir} {
    if {$root != ""} {
        delete object $root
    }
    set root [createNode $dir]
    $root configure -state open
    $root refresh
}
```

Each Fileviewer maintains a tree of VisualFileTree objects that represent the files on its display. We use the private root variable to store the name of the root object for the tree. Whenever we call the display method, we destroy the existing file tree by destroying the root node, and then we start a new file tree by creating a new root node. We configure the root node to the "open" state, so that when it draws itself, it will display other files and folders below it. Finally, we tell the root node to refresh itself, and it draws the entire file tree onto the canvas.

Whenever we need to create a VisualFileTree node for the Fileviewer, we call the createNode method, giving it the name of the file that we want to represent. The implementation of this method is shown in Example 2-3.

We start by creating a VisualFileTree object. Remember, its constructor demands two arguments: the file that it represents, and the canvas that will display it. We use the display component that we created for this Fileviewer

Example 2-3 Implementation for the Fileviewer::createNode method.

```
body Fileviewer::createNode {fname} {
    set obj [VisualFileTree ::#auto $fname $itk_component(display)]

    $obj configure -name $fname \
        -sort [code cmpTree -name] \
        -procreate [code $this createNode]

    if {[file isdirectory $fname]} {
        $obj configure -icon dirIcon
    } elseif {[file executable $fname]} {
        $obj configure -icon programIcon
    } else {
        $obj configure -icon fileIcon
    }
    $obj configure -title [file tail $fname]

    return $obj
}
```

as the display canvas. We get the real window path name for this component from the `itk_component` array, and we pass it into the `VisualFileTree` constructor. We create the `VisualFileTree` object with the name "`::#auto`" so we will get an automatically generated name like "`::visualFileTree12`". As I discussed earlier in the section "Using Objects Outside of Their Namespace" in Chapter 1, this puts the object in the global namespace, so we can share it with other classes like `Tree` that will need to access it.

We configure the `-name` and `-sort` options so that all files will be sorted alphabetically by name. We use the `Fileviewer::cmpTree` procedure as the comparison function for `lsort`. If we were calling this procedure right now in the context of `Fileviewer`, we could use a simple command like `cmpTree`. But we are giving this command to a completely separate `VisualFileTree` object, and it will be used later in the `Tree::reorder` method. In that context, there is no command called `cmpTree`. Therefore, we cannot use a simple command like "`cmpTree -name`". We must wrap it up with the `code` command like "`[code cmpTree -name]`". Roughly translated, this means that the `Fileviewer` is telling the `VisualFileTree` object: "When you need to compare two `VisualFileTree` objects later on, come back to the current (`Fileviewer`) context and call the `cmpTree` procedure. Since we're friends, I'm giving you access to my namespace and letting you use my private procedure."

We also configure the `-procreate` option so that all child `VisualFileTree` nodes are created by the `Fileviewer::createNode` method. Remember, we start with a single root node and build the file tree gradually, as needed. When you double-click on a folder in the display, you open it and ask it to display its contents. If it hasn't already done so, the `VisualFileTree` object will scan the file system at that point, and automatically create child nodes for the files within it. Whatever command we give for the `-procreate` option will be executed by the `VisualFileTree` object in a completely different context. Again, we must be careful to use the `code` command. But in this case, `createNode` is not just a procedure, it is a method, so we must do something extra. We use the command "`[code $this createNode]`". Roughly translated, the `Fileviewer` is telling the `VisualFileTree` object: "When you need to create a node later on, talk to me. My name is `$this`, and you can use my `createNode` method. This is usually a private method, but since we're friends, I'm letting you back in to the current (`Fileviewer`) namespace, and you can access `createNode` from there."

Near the end of the `createNode` method, we configure the `VisualFileTree` object to display the file name and an icon that indicates whether the file is a directory, a program or an ordinary file. When we are done configuring the object, we return its name as the result of the `createNode` method.

Each node uses the `Fileviewer::cmpTree` procedure when sorting its child nodes. This is a standard `lsort`-style procedure. It takes the names of two `VisualFileTree` objects, compares them, and returns "`+1`" if the first goes after the second, "`-1`" if the first goes before the second, and "`0`" if the order does not matter. The implementation of the `cmpTree` procedure is shown in Example 2-4.

Example 2-4 Implementation for the Fileviewer::cmpTree procedure.

```
body Fileviewer::cmpTree {option obj1 obj2} {
    set val1 [$obj1 cget $option]
    set val2 [$obj2 cget $option]
    if {$val1 < $val2} {
        return -1
    } elseif {$val1 > $val2} {
        return 1
    }
    return 0
}
```

We have made this procedure general enough that we can use it to sort based on any option of the `VisualFileTree` object. If we want an alphabetical listing, we use `-name` for the `option` argument.† If we want to sort based on file size, we use `-value` for the `option` argument, and we set the `-value` option to the file size when each `VisualFileTree` object is created.

Fileviewer Creation Command

You create a `Fileviewer` widget like any other [INCR TCL] object—by using the class name as a command:

† This is what we did in the `createNode` procedure shown above.

```
Fileviewer .viewer -background tan
```

Unfortunately, all of the other Tk widget commands have lower case letters. If we want to follow the Tk convention, we should really have a command called `fileviewer` to create a `Fileviewer` widget.

You might wonder: Why not just change the class name to `fileviewer`? We could do this, but Tk has a convention that all widget class names start with a capital letter. You should follow this same convention in [INCR TK]. If you don't, you'll have trouble accessing defaults in the options database, and you'll have trouble with class bindings.

We simply need to add a `fileviewer` procedure that acts as an alias to the real `Fileviewer` command, like this:

```
proc fileviewer {pathName args} {
    uplevel Fileviewer $pathName $args
}
```

This procedure takes a window path name and any option settings, and passes them along to the `Fileviewer` command. Notice that `pathName` is a required argument, so if you forget to specify a window path name, you'll get an error. We use the `uplevel` command so that the widget is created in the context of the caller. After all, the caller wants ownership of whatever widget we create. If we didn't do this, the widget would be created in the namespace that contains the `fileviewer` proc, and in some cases,† this can cause problems.

Defining New Configuration Options

So far, all of the configuration options for a mega-widget like `Fileviewer` have been added by keeping or renaming options from its component widgets. But what if you want to add a brand-new option that doesn't belong to any of the components?

For example, suppose we want to add a `-selectcommand` option to the `Fileviewer`. This is something like the `-command` option for a Tk button. It lets you configure each `Fileviewer` to do something special whenever you select a node in its file tree.

As a trivial example, we could create a `Fileviewer` that prints out a message when each file is selected, like this:

† Suppose we put the `fileviewer` proc in a namespace called `utilities`. Without the `uplevel` command, the `Fileviewer` widgets that it creates would have their access commands added to the `utilities` namespace. This would make it harder to access these widgets, and therefore slow down the application.

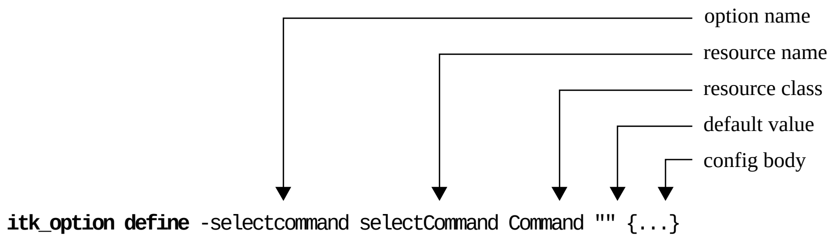
```
fileviewer .fv -selectcommand {puts "selected file: %n"}
pack .fv
```

We will set things up so that any `%n` fields in the command string will be replaced with the name of the selected file. This mimics the Tk `bind` command, and it makes it easy to know which file was selected whenever the command is executed.

Having this feature opens the door for more interesting applications. We might use it to create an image browser for a drawing program. Whenever you click on a file in a recognized image format like GIF, TIFF or JPEG, the selection command could load a thumbnail image that you could preview before clicking *OK*.

The `-selectcommand` option is not kept or renamed from a component widget. It is a brand-new option that we are adding to the `Fileviewer` class itself. If this were an ordinary [INCR TCL] class, we would add a configuration option by defining a public variable. You can do this for a mega-widget too, but if you do, the option won't be tied into the options database properly. Remember, public variables have one name, but each widget option has three names: an option name, a resource name, and a resource class.

Instead, when you define an option in a mega-widget class, you should use the "`itk_option define`" command with the syntax shown in Figure 2-12.



```
itk_option define -selectcommand selectCommand Command "" {...}
```

Figure 2-12 Syntax of the "itk_option define" command.

Believe it or not, this looks a lot like a public variable declaration. It includes the three names for the option, an initial value, and some code that should be executed whenever the option is configured. Like a public variable, the configuration code is optional, and you can specify it outside of the class definition using a `configbody` command.

We can add the `-selectcommand` option to the `Fileviewer` class as shown in Example 2-5. You can find the complete code example in the file *itcl/itk/fileviewer2.itk*. We have also added a `select` method to the `Fileviewer` class.

We'll see in a moment how the `-selectcommand` option and the `select` method work together.

Example 2-5 Adding the "-selectcommand" option to the Fileviewer mega-widget.

```
class Fileviewer {
    inherit itk::Widget

    constructor {args} {
        ...
    }

    itk_option define -selectcommand selectCommand Command ""

    private variable root ""

    public method display {dir}
    public method select {node}

    private method createNode {dir}
    private proc cmpTree {option obj1 obj2}
}

...

body Fileviewer::select {node} {
    set name [$node cget -name]
    regsub -all {%n} $itk_option(-selectcommand) $name cmd
    uplevel #0 $cmd
}
```

Notice that the "`itk_option define`" statement appears outside of the constructor, at the level of the class definition. Again, think of it as a public variable declaration. It defines something about the class.

The `-selectcommand` option has the resource name `selectCommand` and the resource class `Command` in the options database. Whenever a `Fileviewer` widget is created, the options database is used to determine the initial value for this option. If a value cannot be found for either of these names, the default value (in this case, the null string) is used as a last resort.

Whenever a file is selected on the canvas, we'll call the `select` method shown in Example 2-5, giving it the name of the `VisualFileTree` object that was selected. This method replaces all "`%n`" fields in the `-selectcommand` code with the name of the selected file, and executes the resulting command. We are careful to use "`uplevel #0`" instead of `eval` to evaluate the code. That way, the code is executed in the global context, and if it uses any variables, they will be global variables.

You might wonder how we know when a file has been selected. As you will recall from Example 1-14, each `VisualFileTree` object has its own `-select-command` option that is executed whenever a file is selected. We simply tell each `VisualFileTree` node to call the `Fileviewer::select` method when a node is selected. We do this when each `VisualFileTree` node is created, as shown in Example 2-6.

Example 2-6 VisualFileTree nodes notify the Fileviewer of any selections.

```
body Fileviewer::createNode {fname} {
    set obj [VisualFileTree ::#auto $fname $itk_component(display)]

    $obj configure -name $fname \
        -sort [code cmpTree -name] \
        -procreate [code $this createNode] \
        -selectcommand [code $this select %o]
    ...
}
```

When you click on a file, the entire chain of events unfolds like this. Your click triggers a binding associated with the file, which causes the `VisualFileTree` object to execute its `-selectcommand` option. This, in turn, calls the `select` method of the Fileviewer, which executes its own `-selectcommand` option. In effect, we have used the primitive `-selectcommand` on each `VisualFileTree` object to support a high-level `-selectcommand` for the entire `Fileviewer`.

As another example of a brand-new option, suppose we add a `-scrollbar` option to the `Fileviewer`, to control the scrollbar. This option might have three values. If it is `on`, the scrollbar is visible. If it is `off`, the scrollbar is hidden. If it is `auto`, the scrollbar appears automatically whenever the file tree is too long to fit on the canvas.

Example 2-7 shows the `Fileviewer` class with a `-scrollbar` option. You can find a complete code example in the file *itcl/itk/fileviewer3.itk*.

Example 2-7 Adding the "-scrollbar" option to the Fileviewer mega-widget.

```
class Fileviewer {
    inherit itk::Widget

    constructor {args} {
        ...
    }

    itk_option define -selectcommand selectCommand Command ""

    itk_option define -scrollbar scrollbar Scrollbar "on" {
        switch -- $itk_option(-scrollbar) {
            on - off - auto {
                fixScrollbar
            }
            default {
                error "bad value \"$itk_option(-scollbar)\""
            }
        }
    }

    private variable root ""

    public method display {dir}
    public method select {node}
```

Example 2-7 Adding the "-scrollbar" option to the Fileviewer mega-widget.

```
    private method createNode {dir}
    private proc cmpTree {option obj1 obj2}

    private method fixScrollbar {args}
    private variable sbvisible 1
}
```

In this case, we have added some configuration code after the default "on" value. Whenever the configure method modifies this option, it will execute this bit of code to check the new value and bring the widget up to date. In this case, we check the value of the -scrollbar option to make sure that it is on or off or auto. You can always find the current value for a configuration option in the itk_option array. If the value looks good, we use the fixScrollbar method to update the scrollbar accordingly. If it does not have one of the allowed values, we signal an error, and the configure method sets the option back to its previous value, and then aborts with an error.

We must also call fixScrollbar whenever any conditions change that might affect the scrollbar. Suppose the scrollbar is in auto mode. If we shorten the widget, we might need to put up the scrollbar. If we lengthen the widget, we might need to take it down. If we double-click on a file and expand or collapse the file tree, again, we might need to fix the scrollbar. All of these conditions trigger a change in the view associated with the canvas. To handle them, we must make sure that fixScrollbar gets called whenever the view changes. We do this by hijacking the normal communication between the canvas and the scrollbar, as shown in Example 2-8.

Example 2-8 Using fixScrollbar to handle changes in the canvas view.

```
class Fileviewer {
    inherit itk::Widget

    constructor {args} {
        ...
        itk_component add display {
            canvas $itk_interior.canv -borderwidth 2 \
                -relief sunken -background white \
                -yscrollcommand [code $this fixScrollbar]
        } {
            ...
        }
        pack $itk_component(display) -side left -expand yes -fill both
        eval itk_initialize $args
    }
    ...
}
```

Each time the view changes, the canvas calls its -yscrollcommand to notify the scrollbar. In this case, it calls our fixScrollbar method instead, which checks to see if the scrollbar should be visible, and updates it accordingly. The fixScrollbar method then passes any arguments through to the scrollbar, so the normal canvas/scrollbar communication is not interrupted.

The `fixScrollbar` method is implemented as shown in Example 2-9.

Example 2-9 Implementation for the Fileviewer::fixScrollbar method.

```
body Fileviewer::fixScrollbar {args} {
    switch $itk_option(-scrollbar) {
        on  { set sbstate 1 }
        off { set sbstate 0 }

        auto {
            if {[$itk_component(display) yview] == "0 1"} {
                set sbstate 0
            } else {
                set sbstate 1
            }
        }
    }
    if {$sbstate != $sbvisible} {
        if {$sbstate} {
            pack $itk_component(scrollbar) -side right -fill y
        } else {
            pack forget $itk_component(scrollbar)
        }
        set sbvisible $sbstate
    }

    if {$args != ""} {
        eval $itk_component(scrollbar) set $args
    }
}
```

First, we check the `-scrollbar` option and determine whether or not the scrollbar should be visible, saving the result in the variable `sbstate`. If the scrollbar is `on` or `off`, the answer is obvious. But if it is `auto`, we must check the current view on the `display` canvas. If the entire canvas is visible, then the view is “`0 1`”, and the scrollbar is not needed.

We then consult the `sbvisible` variable defined in Example 2-7 to see if the scrollbar is currently visible. If the scrollbar needs to be put up, it is packed into the hull. If it needs to be taken down, then the “`pack forget`” command is used to unpack it.

Finally, we pass any extra arguments on to the `set` method of the scrollbar component. Normally, there are no arguments, and this does nothing. But having this feature lets the `fixScrollbar` method be used as the `-yscrollcommand` for the canvas, without disrupting the normal communication between the canvas and the scrollbar.

Defining “Usual” Options

When you add a component to a mega-widget, you must keep, rename or ignore its configuration options. As we saw earlier, each of the Tk widget classes has a default set of `keep` and `rename` statements to handle its configuration options in the "usual" manner. There is even a usual statement to request the "usual" option-handling code.

But what happens if you use a mega-widget as a component of a larger mega-widget? What if you use a Fileviewer as a component within a larger Fileconfirm mega-widget? Again, you must keep, rename or ignore the configuration options for the Fileviewer component. And what if someone asks for the "usual" option-handling code for a Fileviewer component? It is your job as the mega-widget designer to provide this.

The option-handling commands for a new widget class are defined with a usual declaration, like the one shown in Example 2-10.

Example 2-10 Defining the "usual" options for a Fileviewer component.

```
option add *Fileviewer.width 2i widgetDefault
option add *Fileviewer.height 3i widgetDefault
option add *Fileviewer.scrollbar auto widgetDefault

class Fileviewer {
    ...
}

usual Fileviewer {
    keep -activebackground -activerelief
    keep -background -cursor
    keep -highlightcolor -highlightthickness
    keep -troughcolor
}

proc fileviewer {pathName args} {
    uplevel Fileviewer $pathName $args
}
```

Here, the keep commands refer to the overall options for a Fileviewer mega-widget. Suppose you use a Fileviewer as a component in a Fileconfirm mega-widget, and you ask for the "usual" options. Each of the options shown above would be kept in the Fileconfirm option list. For example, if you set the master -background option on a Fileconfirm, it would propagate the change to the -background option of its Fileviewer component, which in turn would propagate the change to the -background option on its scrollbar and the -highlightbackground option on its canvas.

It is best to write the "usual" declaration at the last moment, after you have put the finishing touches on a mega-widget class. You simply examine the master configuration options one-by-one and decide if they should be kept, renamed or ignored.

Only the most generic options should be kept or renamed in the "usual" declaration for a widget class. If we had two Fileviewer components within a Fileconfirm mega-widget, both of them might be tied to the Fileconfirm option list in the "usual" way. Which options should they have in common?

Options like `-background`, `-foreground`, `-cursor` and `-font` are all good candidates for the `keep` command. On the other hand, options like `-text`, `-bitmap` and `-command` are usually unique to each component, so options like these should be ignored.

Inheritance and Composition

Mega-widgets can be used to build even larger mega-widgets. Like the Tk widgets, mega-widgets support composition. One mega-widget can be used as a component within another. But mega-widgets also support inheritance. One mega-widget class can inherit all of the characteristics of another, and add its own specializations. You are no longer limited to what a class like `Fileviewer` provides. You can derive another class from it and add your own enhancements. So a mega-widget toolkit can be extended in a way that transcends the standard Tk widgets.

In this section, we explore how inheritance and composition can be used to build mega-widgets. These relationships become even more powerful when combined.

Designing a Base Class

Suppose we plan to build many different kinds of confirmation windows. We may build a `Messageconfirm` mega-widget, which prompts the user with a question and requests a *Yes*/*No* or *OK*/*Cancel* response. We may build a `Fileconfirm` mega-widget, which gives the user a file browser to select a file, and requests a *Load*/*Cancel* or *Save*/*Cancel* response.

Both of these mega-widgets have a common abstraction. They pop up in their own toplevel window, they have *OK*/*Cancel* buttons at the bottom, and they prevent the application from continuing until the user has responded. When mega-widgets share a common abstraction like this, we can design a mega-widget base class to handle it. In this case, we will create a base class called `Confirm` which provides the basic functionality for a confirmation dialog.

A `Confirm` mega-widget looks like the one shown in Figure 2-13. It has an empty area called the “contents” frame at the top, which can be filled in with messages, file browsers, or whatever information is being confirmed. A separator line sits between this frame and the *OK* and *Cancel* buttons at the bottom of the dialog. This dialog always pops up on the center of the desktop, and it locks out the rest of the application until the user has pressed either *OK* or *Cancel*.

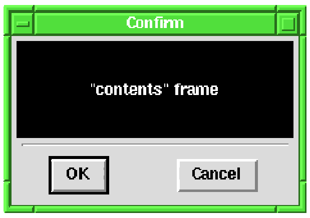


Figure 2-13 Generic Confirm mega-widget.

The class definition for a `Confirm` mega-widget is shown in Example 2-11. A complete code example appears in the file *itcl/itk/confirm.itk*.

Example 2-11 The class definition for a Confirm mega-widget.

```
class Confirm {
    inherit itk::Toplevel

    constructor {args} {
        itk_component add contents {
            frame $itk_interior.contents
        }
        pack $itk_component(contents) -expand yes -fill both -padx 4 -pady 4

        itk_component add separator {
            frame $itk_interior.sep -height 2 \
                -borderwidth 1 -relief sunken
        }
        pack $itk_component(separator) -fill x -padx 8

        private itk_component add controls {
            frame $itk_interior.cntl
        }
        pack $itk_component(controls) -fill x -padx 4 -pady 8

        itk_component add ok {
            button $itk_component(controls).ok -text "OK" \
                -command [code $this dismiss 1]
        }
        pack $itk_component(ok) -side left -expand yes

        itk_component add cancel {
            button $itk_component(controls).cancel -text "Cancel" \
                -command [code $this dismiss 0]
        }
        pack $itk_component(cancel) -side left -expand yes

        wm withdraw $itk_component(hull)
        wm group $itk_component(hull) .
        wm protocol $itk_component(hull) \
            WM_DELETE_WINDOW [code $this dismiss]

        after idle [code $this centerOnScreen]
        set itk_interior $itk_component(contents)

        eval itk_initialize $args
    }

    private common responses
```

Example 2-11 The class definition for a Confirm mega-widget.

```
    public method confirm {}
    public method dismiss {{choice 0}}

    protected method centerOnScreen {}
}
```

The `Confirm` class inherits from the `itk::Toplevel` base class, so each `Confirm` widget pops up with its own toplevel window. We create a frame component called `contents` to represent the “contents” area at the top of the window. We use another frame component called `separator` to act as a separator line, and we add two button components called `ok` and `cancel` at the bottom of the window. Note that the `ok` and `cancel` components sit inside of a frame component called `controls`. This frame was added simply to help with packing.

When you have a component like `controls` that is not an important part of the mega-widget, you can keep it hidden. You simply include a `protected` or `private` declaration in front of the `itk_component` command. This is the same `protected` or `private` command that you would normally use in a namespace to restrict access to a variable or procedure. It simply executes whatever command you give it, and it sets the protection level of any commands or variables created along the way. When a mega-widget component is marked as protected or private, it can be used freely within the mega-widget class, but it cannot be accessed through the built-in `component` method by anyone outside of the class.

Once we have created all of the components, we do a few other things to initialize the `Confirm` widget. Since this is a toplevel widget, we use the `wm` command to tell the window manager how it should handle this window. We ask the window manager to withdraw the window, so that it will be invisible until it is needed. We group it with the main window of the application. Some window managers use the group to iconify related windows when the main application window is iconified. We also set the “delete” protocol, so that if the window manager tries to delete the window, it will simply invoke the `dismiss` method, as if the user had pressed the *Cancel* button.

In all of these commands, we are talking to the window manager about a specific toplevel window—the one that contains our `Confirm` mega-widget. Remember, the container for any mega-widget is a component called the `hull`, which in this case is created automatically by the `itk::Toplevel` base class. The window manager won’t understand a symbolic component name like `hull`, so we give it the real window path name stored in `itk_component(hull)`.

When the `Confirm` mega-widget appears, we want it to be centered on the desktop. We have a method called `centerOnScreen` that determines the overall size of the dialog, and uses the "`wm geometry`" command to position it on the desktop. You can find the implementation of this method in the file *itcl/itk/confirm.itk*. The details are not particularly important. We should call this method once, when the widget is first created. But we can't call it directly in the constructor. At this point, we haven't finished building the `Confirm` dialog. As we'll see shortly, more widgets need to be created and packed into the "contents" frame. If we call `centerOnScreen` too early, the dialog will be centered based on its current size, and when more widgets are added, it will appear to be off-center.

This situation arises from time to time—you want something to happen *after* construction is complete. You can handle this quite easily with the Tk `after` command. Normally, you give `after` a command and a certain time interval, and the command is executed after that much time has elapsed. In this case, we don't care exactly when `centerOnScreen` is called, so instead of using a specific time interval, we use the key word `idle`. This tells `after` to execute the command at the first opportunity when the application is idle and has nothing better to do. Again, since the `centerOnScreen` method will be called in another context, long after we have returned from the constructor, we are careful to include the object name `$this`, and to wrap the code fragment with the `code` command.

As always, we finish the construction by calling `itk_initialize` to initialize the master option list and apply any option settings.

A `Confirm` widget can be created and packed with a label like this:

```
confirm .ask
set win [.ask component contents]
label $win.message -text "Do you really want to do this?"
pack $win.message
```

Although we did not explicitly create options for the labels on the *OK*/*Cancel* buttons, we can still change them like this:

```
.ask component ok configure -text "Yes"
.ask component cancel configure -text "No"
```

Sometimes it is better to access individual components like this, instead of adding more options to the master option list. If a mega-widget has too many options, it is difficult to learn and its performance suffers.

Whenever a confirmation is needed, the `confirm` method can be used like this:

```
if {[.ask confirm]} {
    puts "go ahead"
} else {
    puts "abort!"
}
```

The `confirm` method pops up the `Confirm` window, waits for the user's response, and returns `1` for *OK* and `0` for *Cancel*. The `if` statement checks the result and prints an appropriate message.

The `confirm` method is implemented as shown in Example 2-12.

Example 2-12 Implementation for the Confirm::confirm method.

```
body Confirm::confirm {} {
    wm deiconify $itk_component(hull)
    grab set $itk_component(hull)
    focus $itk_component(ok)

    tkwait variable [scope responses($this)]

    grab release $itk_component(hull)
    wm withdraw $itk_component(hull)

    return $responses($this)
}
```

First, we ask the window manager to pop up the window using the "`wm deiconify`" command, and we set a grab on the window. At this point, all other windows in the application will be unresponsive, and the user is forced to respond by pressing either *OK* or *Cancel*. The default focus is assigned to the *OK* button, so the user can simply press the space bar to select *OK*.

The `tkwait` command stops the normal flow of execution until the user has responded. In this case, we watch a particular variable that will change as soon as the user presses either *OK* or *Cancel*. Each `Confirm` widget should have its own variable for `tkwait`. Normally, we would use an object variable for something like this, but there is no way to pass an object variable to a command like `tkwait`. The `scope` operator will capture the namespace context for a variable, but not the object context. So the `scope` command works fine for common class variables, but not for object variables. We can use the following trick to get around this problem: We define a common array called `responses`, and we assign each widget a slot with its name `$this`. As long as we wrap each slot `responses($this)` in the `scope` command, we have no trouble passing it along to `tkwait`.

Thanks to the `-command` option of the `ok` and `cancel` components, pressing *OK* invokes the `dismiss` method with the value `1`, and pressing *Cancel* invokes the `dismiss` method with the value `0`. The `dismiss` method itself is quite trivial. Its body is shown in Example 2-13.

Example 2-13 Implementation for the Confirm::dismiss method.

```
body Confirm::dismiss {{choice 0}} {
    set responses($this) $choice
}
```

It simply stores whatever value you give it in the `responses` array. But if we're sitting at the `tkwait` instruction in the `confirm` method, this is just what we're looking for. Setting this variable causes `tkwait` to return control, and execution resumes within the `confirm` method. We release the grab, hide the dialog, and return the user's response.

The `Confirm` mega-widget is useful in its own right, but it can be even more useful as the basis of other mega-widget classes. Derived classes like `Messageconfirm` and `Fileconfirm` can inherit most of the basic functionality, and simply add a few components into the `contents` frame.

But how do derived classes know that they are supposed to use the `contents` frame? We use the variable `itk_interior` to track this. In the `itk::Widget` or `itk::Toplevel` base class, `itk_interior` is set to the window path name of the `hull` component. In the `Confirm` base class, we create components in this interior, and then change `itk_interior` to the window path name of the `contents` frame. Derived classes create components in this interior, and perhaps change `itk_interior` to their own innermost window. If all classes use `itk_interior` like this, making classes work together becomes a simple matter of changing their `inherit` statements.

Using Inheritance

We can continue with the example described above, using inheritance to create a `Messageconfirm` mega-widget like the one shown in Figure 2-14. A `Messageconfirm` *is-a* `Confirm`, but it has an icon and a text message in the `contents` frame.

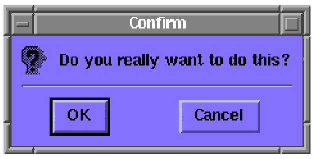


Figure 2-14 A Messageconfirm mega-widget.

The class definition for Messageconfirm is shown in Example 2-14. A complete code example appears in the file *itcl/itk/messageconfirm.itk*.

Example 2-14 Class definition for a Messageconfirm mega-widget.

```
class Messageconfirm {
    inherit Confirm

    constructor {args} {
        itk_component add icon {
            label $itk_interior.icon -bitmap questhead
        } {
            usual
            rename -bitmap -icon icon Bitmap
        }
        pack $itk_component(icon) -side left

        itk_component add message {
            label $itk_interior.mesg -wraplength 3i
        } {
            usual
            rename -text -message message Text
        }
        pack $itk_component(message) -side left -fill x

        eval itk_initialize $args
    }
}
```

By inheriting from the `Confirm` class, `Messageconfirm` automatically has its own toplevel window with a `contents` frame, a separator line, and *OK* and *Cancel* buttons. It has `confirm` and `dismiss` methods, and it automatically comes up centered on the desktop.

It has the same basic configuration options too, but it does not inherit any default settings from the base class. If you have defined some resource settings for the `Confirm` class, like this:

```
option add *Confirm.background blue widgetDefault
option add *Confirm.foreground white widgetDefault
```

you will have to repeat those settings for the derived class:

```
option add *Messageconfirm.background blue widgetDefault
option add *Messageconfirm.foreground white widgetDefault
```

In its constructor, the `Messageconfirm` adds an `icon` component, which represents the bitmap icon to the left of the message. We use the `usual` command in the option-handling commands for this component to integrate most of its options in the "usual" manner, but we rename the `-bitmap` option, calling it `-icon` in the master list. This is a better name for the option, since it indicates which bitmap we are controlling.

The `Messageconfirm` also adds a `message` component, which represents the message label. Again, we use the `usual` command to integrate most of its options, but we rename the `-text` option, calling it `-message` in the master list.

As always, we create these two component widgets with the root name `$itk_interior`. But in this case, `$itk_interior` contains the name of the contents frame that we created in the constructor for base class Confirm. So these new components automatically sit inside of the contents frame, as I explained earlier.

We might create a Messageconfirm widget like this:

```
messageconfirm .check -background tomato -icon warning \
    -message "Do you really want to do this?"
```

and use it like this:

```
if {[.check confirm]} {
    puts "go ahead"
} else {
    puts "abort!"
}
```

With a simple inherit statement and just a few lines of code, we have created a very useful widget.

Mixing Inheritance and Composition

Inheritance is a powerful technique, but so is composition. Many good designs use both relationships. For example, suppose we create a Fileconfirm mega-widget like the one shown in Figure 2-15. A Fileconfirm *is-a* Confirm, and *has-a* Fileviewer component packed into the contents frame. It also *has-a* entry component packed into the contents frame. When the user selects a file, its name is automatically loaded into the entry component. Of course, the user can also edit this name, or type an entirely different name into the entry component.

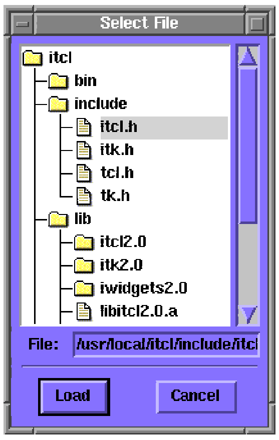


Figure 2-15 A Fileconfirm mega-widget.

The class definition for Fileconfirm is shown in Example 2-15. A complete code example appears in the file *itcl/itk/fileconfirm.itk*.

Example 2-15 Class definition for a Fileconfirm mega-widget.

```
 class Fileconfirm {
    inherit Confirm

    constructor {args} {
        itk_component add fileTree {
            fileviewer $itk_interior.files \
                -selectcommand [code $this select %n]
        }
        pack $itk_component(fileTree) -expand yes -fill both

        itk_component add fileLabel {
            label $itk_interior.flabel -text "File:"
        }
        pack $itk_component(fileLabel) -side left -padx 4

        itk_component add fileEntry {
            entry $itk_interior.fentry
        }
        pack $itk_component(fileEntry) -side left -expand yes -fill x

        eval itk_initialize $args
    }

    itk_option define -directory directory Directory "" {
        $itk_component(fileTree) display $itk_option(-directory)
    }

    public method get {} {
        return [$itk_component(fileEntry) get]
    }
    protected method select {name} {
        $itk_component(fileEntry) delete 0 end
        $itk_component(fileEntry) insert 0 $name
    }
}
```

Again, by inheriting from the `Confirm` class, `Fileconfirm` automatically has its own toplevel window with a `contents` frame, a separator line, and *OK* and *Cancel* buttons. It has `confirm` and `dismiss` methods, and it automatically comes up centered on the desktop.

In its constructor, `Fileconfirm` adds a `Fileviewer` component. It also adds a *File:* label and an entry component at the bottom of the `contents` frame. These are three separate components, but they interact within the `Fileconfirm` in the following manner. When the user selects a file, the `Fileviewer` executes its `-selectcommand` code, which calls the `Fileconfirm::select` method with the selected file name substituted in place of `%n`. The `select` method then loads the file name into the entry component. Whatever name is sitting in the entry component is treated as the official file selection. At any point, you can use the `Fileconfirm::get` method to get the file name sitting in the entry component.

The -directory option controls the top-level directory in the Fileconfirm. Whenever it is configured, it automatically invokes the display method of the Fileviewer to update the display.

We might create a Fileconfirm widget like this:

```
fileconfirm .files -directory $env(HOME)
```

and use it like this:

```
if {[.files confirm]} {
    puts "selected file: [.files get]"
} else {
    puts "abort!"
}
```

We use the confirm method to pop up the dialog and wait for the user to select a file and press *OK* or *Cancel*. If he pressed *OK*, we use the get method to get the name of the selected file, and we print it out.

We leveraged the Confirm class with inheritance, and the Fileviewer class with composition. Together, these two techniques produce a complex widget with just a little extra code.

Reviving Options

Sometimes a derived class needs to override the way a base class handles its configuration options. For example, suppose we want to define the -width and -height options of a Fileviewer widget so that they represent the overall width and height, including the scrollbar. Previously, we kept the -width and -height options from the canvas component, so the overall width was a little bigger when the scrollbar was visible. Instead, we need to keep the -width and -height options from the hull component. But the hull component is created in the itk::Widget base class, and we can't modify that code.

Options that belong to a base class component can be revived in a derived class using the "itk_option add" command. You simply tell the mega-widget to add an option that was previously ignored back into the master list. A complete code example appears in the file *itcl/itk/fileviewer4.itk*, but the important parts are shown in Example 2-16.

Example 2-16 Options can be revived using "itk_option add".

```
option add *Fileviewer.width 2i widgetDefault
option add *Fileviewer.height 3i widgetDefault
option add *Fileviewer.scrollbar auto widgetDefault

class Fileviewer {
    inherit itk::Widget

    constructor {args} {
```

Example 2-16 Options can be revived using "itk_option add".

```
        itk_option add hull.width hull.height
        ...
        itk_component add display {
            canvas $itk_interior.canv -borderwidth 2 \
                -relief sunken -background white \
                -yscrollcommand [code $this fixScrollbar] \
                -width 1 -height 1
        } {
            keep -cursor
            keep -highlightcolor -highlightthickness
            rename -highlightbackground -background background Background
        }
        pack $itk_component(display) -side left -expand yes -fill both

        eval itk_initialize $args

        pack propagate $itk_component(hull) 0
        bind $itk_component(display) <Configure> [code $this fixScrollbar]
    }
    ...
}
```

The "`itk_option add`" command is different from the "`itk_option define`" command that we saw earlier. You use "`itk_option define`" as part of a class definition to define a new configuration option. On the other hand, you use "`itk_option add`" in the constructor (or in any other method) to reinstate a configuration option that already exists but was ignored by a base class. The "`itk_option add`" command can appear anywhere in the constructor, but it is normally included near the top. It should be called before `itk_initialize`, since options like `-width` and `-height` might appear on the `args` list.

Each option is referenced with a name like "*component*.*option*" if it comes from a component, or with a name like "*class*::*option*" if it comes from an "`itk_option define`" command. In either case, *option* is the option name without the leading "`-`" sign. In this example, we are reviving the `-width` and `-height` options of the `hull` component, so we use the names `hull.width` and `hull.height`. `Fileviewer` widgets will behave as if these options had been kept when the component was first created.

Now that we have reinstated the `-width` and `-height` options, we must make sure that they work. Frames normally shrink-wrap themselves around their contents, but we can use the "`pack propagate`" command to disable this, so the hull will retain whatever size is assigned to it. We also set the width and height of the canvas to be artificially small, but we pack it to expand into any available space.

Suppressing Options

Options coming from a base class can be suppressed using the "`itk_option remove`" command. But this command should be used carefully.

A derived class like `Fileviewer` should have all of the options defined in its base class `itk::Widget`. After all, a `Fileviewer` *is-a* `Widget`. An option should be suppressed in the base class only if it is being redefined in the derived class.

For example, suppose we want to change the meaning of the `-cursor` option in the `Fileviewer` widget. We set things up previously so that when you configure the master `-cursor` option, it propagates the change down to all of the components in the `Fileviewer`. Suppose instead that we want the `-cursor` option to affect only the `display` component. That way, we could assign a special pointer for selecting files, but leave the scrollbar and the hull with their appropriate default cursors.

To do this, we must keep the `-cursor` option on the `display` component, but avoid keeping it on the `scrollbar` and `hull` components. A complete code example appears in the file *itcl/itk/fileviewer5.itk*, but the important parts are shown below in Example 2-17.

Example 2-17 Options can be suppressed using "itk_option remove".

```
option add *Fileviewer.width 2i widgetDefault
option add *Fileviewer.height 3i widgetDefault
option add *Fileviewer.scrollbar auto widgetDefault
option add *Fileviewer.cursor center_ptr widgetDefault

class Fileviewer {
    inherit itk::Widget

    constructor {args} {
        itk_option add hull.width hull.height
        itk_option remove hull.cursor

        itk_component add scrollbar {
            scrollbar $itk_interior.sbar -orient vertical \
                -command [code $itk_interior.canv yview]
        } {
            usual
            ignore -cursor
        }
        ...

        eval itk_initialize $args
        component hull configure -cursor ""

        pack propagate $itk_component(hull) 0
        bind $itk_component(display) <Configure> [code $this fixScrollbar]
    }
    ...
}
```

Since we create the `scrollbar` component in class `Fileviewer`, we can simply fix its option-handling code to suppress the `-cursor` option. We integrate its options in the "usual" manner, but we specifically ignore its `-cursor` option. The `hull` component, on the other hand, is created in the `itk::Widget` base class, and we can't modify that code. Instead, we use the "itk_option remove" command to disconnect its -cursor option from the master list. We create the display component just as we did before, keeping its -cursor option. Having done all this, we can configure the master -cursor option, and it will affect only the display component.

We might even add a default cursor like this:

```
option add *Fileviewer.cursor center_ptr widgetDefault
```

Whenever we create a new Fileviewer widget, its -cursor option will be center_ptr by default, so the file area will have a cursor that is more suitable for selecting files.

At this point, the example should be finished. But there is one glitch that keeps this example from working properly. Unfortunately, when you set a resource on a class like Fileviewer, it affects not only the master Fileviewer options, but also the options on the hull component that happen to have the same name. We were careful to disconnect the hull from the master -cursor option, but unless we do something, the hull will think its default cursor should be center_ptr. Even though it is not connected to the master option, it will accidentally get the wrong default value.

We can counteract this problem by explicitly configuring the hull component in the Fileviewer constructor like this:

```
component hull configure -cursor ""
```

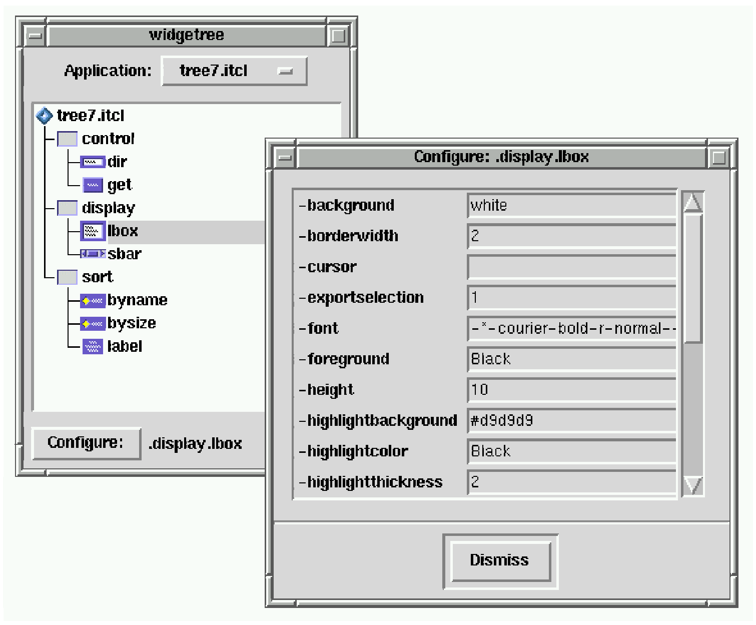
So the hull will indeed get the wrong default value, but we have explicitly set it back to its default value, which is the null string.† This problem is rare. It occurs only when you try to suppress one of the hull options like -cursor, -borderwidth or -relief, and yet you set a class resource in the options database. It is easily fixed with an explicit configuration like the one shown above.

Building Applications with Mega-Widgets

Using mega-widgets as building blocks, applications come together with astonishing speed. Consider the widgetree application shown in Figure 2-16, which is modeled after the hierquery program.‡ It provides a menu of Tcl/Tk applications that are currently running on the desktop. When you select a target application, its widget hierarchy is loaded into the main viewer. You can double-click on any widget in the tree to expand or collapse the tree at that point. If you select a widget and press the *Configure* button, you will get a

† In Tk, widgets with a null cursor inherit the cursor from their parent widget.
‡ David Richardson, "Interactively Configuring Tk-based Applications," *Proceedings of the Tcl/Tk Workshop*, New Orleans, LA, June 23-25, 1994.

panel showing its configuration options. You can change the settings in this panel and immediately apply them to the target application. This tool is a great debugging aid. It lets you explore an unfamiliar Tk application and quickly make changes to its appearance.



-background white
-borderwidth 2
-cursor
-exportselection 1
-font -*-courier-bold-r-normal--
-foreground Black
-height 10
-highlightbackground #d9d9d9
-highlightcolor Black
-highlightthickness 2

Figure 2-16 The "widgetree" application lets you explore any Tk application.

The `widgetree` application was built with a handful of mega-widgets and about 100 lines of Tcl/Tk code. Most of the mega-widgets came off-the-shelf from the [INCR WIDGETS] library, described in Chapter **XXX**. The application menu is an `Optionmenu` widget. The panel of configuration options is a `Dialog` with an internal `Scrolledframe` containing `Entryfield` widgets, which represent the various configuration options.

We developed one customized mega-widget for this application: a `Widgetviewer` class to manage the widget tree. You can find the code for the `Widgetviewer` class in the file *itcl/widgetree/widgetviewer.itk*. The details are not all that important. As you might have noticed, the `Widgetviewer` looks suspiciously like a `Fileviewer`. It has a `display` component and a `scrollbar` component, and it stores its data using `VisualWidgetTree` objects. Like the `VisualFileTree` class, the `VisualWidgetTree` class inherits from the `Tree` and `VisualRep` classes developed in the previous chapter. But instead of populating itself with nodes that represent files, each `VisualWidgetTree` object populates itself with nodes that represent child widgets. When you expand a

VisualWidgetTree node on the display, you trigger a call to its contents method and the node populates itself. It sends the "winfo children" command to the target application, gets a list of child widgets, and creates other VisualWidgetTree objects to represent the children.

The widgetree application has many different classes that all contribute to its operation. You can find the code for this application in the file *itcl/widgetree/widgetree*. Rather than present the code here, we will simply comment on the way that these classes were used to structure the code.

The relationships between these classes are a mixture of inheritance and composition. They can be diagrammed using the OMT notation† as shown in Figure 2-17. A Widgetviewer *is-a* itk::Widget, and it *has-a* VisualWidgetTree root object. A VisualWidgetTree *is* both a WidgetTree and a VisualRep, and a WidgetTree *is-a* Tree.

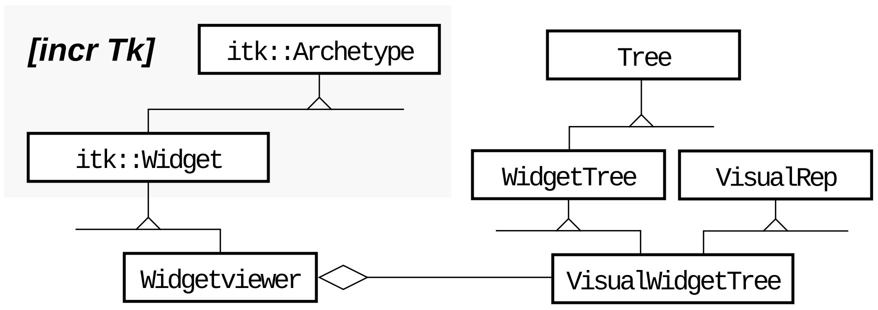


Figure 2-17 The "widgetree" application has many different classes working together.

The same application can be built without objects and mega-widgets, but it requires more code, and the final result might not have as many finishing touches. For example, the configuration options for our widgetree application are presented on a scrollable form, in case the list is long. Nodes in the widget tree can be expanded or collapsed, and a scrollbar comes and goes as needed. Many developers avoid writing extra code for features like these. With mega-widgets, the code can be written once and reused again and again on future projects. This makes Tcl/Tk even more effective for building large applications.

† James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

Summary

Extension:	[incr Tk] - Mega-Widget Framework
Author:	Michael J. McLennan Bell Labs Innovations for Lucent Technologies mmclennan@lucent.com
Other Contributors:	Mark L. Ulferts Jim Ingham ...and many others listed on the web site
Platforms Supported:	All major Unix platforms Linux Windows 95 (release itcl2.2 and beyond) Macintosh (release itcl2.2 and beyond)
Web Site:	`http://www.tcltk.com/itk`

Mailing List: (bug reports)	`mail -s "subscribe" itcl-request@tcltk.com`
	to subscribe to the mailing list
	`mail itcl@tcltk.com`
	to send mail

Quick Reference

Public Methods

The following methods are built into all mega-widgets. If you have created a mega-widget with the Tk name *pathName*, you can access these methods as follows:

`pathName` `cget -option`

Returns the current value for any mega-widget option. Works just like the usual `cget` method in Tk.

`pathName` `component ?symbolicName? ?command arg arg ...?`

Provides access to well-known components within a mega-widget.

`pathName` `configure ?-option? ?value -option value ...?`

Used to query or set mega-widget options. Works just like the usual `configure` method in Tk.

Protected Methods

The following methods are used within a mega-widget class as part of its implementation:

```
itk_component add symbolicName {
    widget pathName ?arg arg...?
}
```

or

```
itk_component add symbolicName {
    widget pathName ?arg arg...?
} {
    ignore -option ?-option -option ...?
    keep -option ?-option -option ...?
    rename -option -newName resourceName resourceClass
    usual ?tag?
}
```

Creates a widget and registers it as a mega-widget component. The extra `ignore`, `keep`, `rename` and `usual` commands control how the configuration options for this component are merged into the master option list for the mega-widget.

`itk_option` `add` *`optName`* `?`*`optName optName`*`...?`

where *optName* is *component*`.`*option* or *className*`::`*option*. Adds an option that was previously ignored back into the master option list.

`itk_option` `remove` *`optName`* `?`*`optName optName`*`...?`

where *optName* is *component*`.`*option* or *className*`::`*option*. Removes an option that was previously merged into the master option list.

`itk_option` `define` *`-option resourceName resourceClass init`* `?`*`configBody`*`?`

Defines a new configuration option for a mega-widget class.

`itk_initialize` `?`*`-option value -option value`*`...?`

Called when a mega-widget is constructed to initialize the master option list.

Protected Variables

The following variables can be accessed within a mega-widget class:

`itk_component(`*`symbolicName`*`)`

Contains the Tk window path name for each component named *symbolicName*.

`itk_interior`

Contains the name of the toplevel or frame within a mega-widget which acts as a container for new components.

`itk_option(`*`-option`*`)`

Contains the current value for any configuration option.

Auxiliary Commands

The following commands are available outside of a mega-widget class. They provide useful information about all Tk widgets:

`usual`	`*tag* ?*commands*?` Used to query or set "usual" option-handling commands for a widget in class *tag*.
`winfo`	`command *window*` Returns the access command for any widget, including its namespace qualifiers.
`winfo`	`megawidget *window*` Returns the name of the mega-widget containing the widget named *window*.