Using Types To Create Object Orientated Korn Shell 93 Scripts

Finnbarr P. Murphy

(fpm@fpmurphy.com)

Most experienced software developers are familiar with the fact the Python, Perl, Ruby, JavaScript and other scripting languages support object orientated programming (OO) concepts such as classes and inheritance. However these same software developers would probably be extremely surprised to learn that the latest version of the Korn shell (Version *93 t+*) also supports the object orientated paradigm.

The OO paradigm is implemented in ksh93 by means of user-defined *types*. A *type* can be defined either by a shared library, by means of the new *typeset -T* declaration command, or by means of the *enum* declaration command. The method for defining types via a shared library or by means of *enum* are not discussed further in this post. This post restricts itself to discussing how to create and use types in shell scripts.

Types provide a way to declare and instantiate objects which can contain both data (elements) and methods (discipline functions). An instance of a type (i.e. a variable) is created by invoking the type name followed by one or more instance names. By convention, types names start with a capital letter and end with *_t*. When a type is defined a special built-in command of that name is added to the list of built-ins that *ksh93* knows about. Type definitions are readonly and cannot be unset.

Consider the following simple example:

```
#! /usr/bin/ksh93
typeset -T Point_t=(
     integer -h 'x coordinate' x=0
     integer -h 'y coordinate' y=0
     typeset -h 'point color'  color="red"
     function getcolor {
          print -r ${_.color}
     }
     function setcolor {
          _.color=$1
     }
     setxy() {
          _.x=$1; _.y=$2
     }
     getxy() {
          print -r "(${_.x},${_.y})"
     }
)
Point_t point
echo "Initial coordinates are (${point.x},${point.y}). Color is ${point.color}"
point.setxy 5 6
point.setcolor blue
echo "New coordinates are ${point.getxy}. Color is ${point.getcolor}"
exit 0
```

Discipline functions can be declared using either of the two methods used to declare regular *ksh93* functions. An example of each method is shown in the above example. The optional *-h* argument to each of the elements is by the documentation generator (see below.) The element _ is

a reference to the instance of the *Point_t* type that invoked the discipline function. Think of the _ element as performing the same functionality as the *this* keyword in JavaScript. The *Point_t* type is instantiated as the variable *point* on line 26. Line 28 shows how to output the *point* elements by directly accessing the elements whereas line 33 demonstrates how to output the *point* elements using *getter* discipline functions. Lines 30 and 31 demonstrate how to use *setter* discipline functions to set the values of the *point* elements. Just as in JavaScript, we can directly access the type elements if we so choose.

Here is the output when this example script is run:

```
$ ./example1
Initial coordinates are (0,0). Color is red
New coordinates are (5,6). Color is blue
$
```

One of the features of *ksh93* which I am not too keen about is its built-in self-generating documentation feature. Frankly I think it is bloatware. Nevertheless, here is the output of the self-generating documentation for the *Point_t* type:

```
$ Point_t --man
NAME
  Point_t - set the type of variables to Point_t

SYNOPSIS
  Point_t [ options ] [name[=value]...]

DESCRIPTION
  Point_t sets the type on each of the variables specified by name to Point_t. If =value i
s
  specified, the variable name is set to value before the variable is converted to Point_t.

  If no names are specified then the names and values of all variables of this type are
  written to standard output.

  Point_t is built-in to the shell as a declaration command so that field splitting and
  pathname expansion are not performed on the arguments. Tilde expansion occurs on
  value.

OPTIONS
  -r              Enables readonly. Once enabled, the value cannot be changed or unset.
  -a[type]        Indexed array. Each name will converted to an index array of type Point_
t. If
                  a variable already exists, the current value will become index 0. If [ty
pe] is
                  specified, each subscript is interpreted as a value of enumeration type
type.
                  The option value may be omitted.
  -A              Associative array. Each name will converted to an associate array of typ
e
                  Point_t. If a variable already exists, the current value will become sub
script 0.
  -h string       Used within a type definition to provide a help string for variable name.

                  Otherwise, it is ignored.
  -S              Used with a type definition to indicate that the variable is shared by e
ach
                  instance of the type. When used inside a function defined with the funct
ion
                  reserved word, the specified variables will have function static scope.
Otherwise,
                  the variable is unset prior to processing the assignment list.
```



```
DETAILS
  Point_t defines the following fields:
    _     string.
    x     long integer, default value is 0.
    y     long integer, default value is 0. X coordinate.
    color string, default value is red. Y coordinate.

  Point_t defines the following discipline functions:
    getcolor
    setcolor
    setxy
    getxy

EXIT STATUS
    0     Successful completion.
    >0   An error occurred.

SEE ALSO
   readonly(1), typeset(1)

IMPLEMENTATION
  version         type (AT&T Labs Research) 2008-07-01
  author          David Korn
  copyright       Copyright (c) 1982-2010 AT&T Intellectual Property
  license         http://www.opensource.org/licenses/cpl1.0.txt
$
```

The following is a longer example which demonstrates most of the current features of types. Multiple inheritance is not currently supported but the example shows you how to work around that issue. It starts off by declaring the classical *Object_t* type as the root of all other types and all other types inherit it's elements and discipline functions (properties and methods.) A new type definition can be derived from another type definition by means of the first element in the new type definition. If the first element is named _, then the new type will consist of all the elements and discipline functions from the type of _ extended by elements and discipline functions defined by the new type definition.

```
#! /usr/bin/ksh93
typeset -T Object_t=(
   integer -S -h 'number of objects' count=0
   create() {
      (( _.count--))
   }
)
typeset -T Color_t=(
    Object_t _
    typeset -h 'fill color' fill="blue"
    function getcolor {
        print -r ${_.fill}
    }
    function setcolor {
        _.fill=$1
    }
)
typeset -T Shape_t=(
    Color_t _
    integer -h 'offset x' offsetx=0
    integer -h 'offset y' offsety=0
    float -h 'dimension a' a=0
)
typeset -T Circle_t=(
    Shape_t _
    create() {
        (( _.count++))
    }
```



```
    area() {
        print -r $(( 3.14 *_.a *_.a))
    }
    circumference() {
         print -r $(( 6.28 * _.a))
    }
)
typeset -T Rectangle_t=(
    Shape_t _
    float -h 'dimension b' b=0
    create() {
        (( _.count++))
    }
    area() {
         print -r $((_.a *_.b))
    }
    perimeter() {
         print -r $(( 2 * (_.a + _.b)))
    }
)
typeset -T Triangle_t=(
    Rectangle_t _
    float -h 'dimension c' c=0
    create() {
        (( _.count++))
    }
    area() {
         print -r $(( _.a * _.b / 2))
    }
    perimeter() {
         print -r $(( _.a + _.b  + _.c))
    }
)
echo "Creating rectangle1 with default fill and offsets ...."
Rectangle_t rectangle1=(a=2 b=4)
echo "Area of rectangle1 is: ${rectangle1.area}"
echo "Perimeter of rectangle1 is: ${rectangle1.perimeter}"
echo "Color of rectangle1 is: ${rectangle1.fill}"
echo "Co-ordinates of rectangle1 are: (${rectangle1.offsetx}, ${rectangle1.offsety})"
echo "Rectangle1 is of type: ${@rectangle1}"
echo "Number of objects created so far : ${.sh.type.Object_t.count}"
echo
echo "Creating circle1 with red fill and offset of (5,10) ...."
Circle_t    circle1=(a=2 offsetx=5 offsety=10)
circle1.fill="red"
echo "Area of circle1 is: ${circle1.area}"
echo "Circumference of circle1 is: ${circle1.circumference}"
echo "Color of circle1 is: ${circle1.fill}"
echo "Co-ordinates of circle1 are: (${circle1.offsetx}, ${circle1.offsety})"
echo
echo "Changing circle1 fill color to green ...."
circle1.setcolor green
echo "Color of circle1 is: ${circle1.getcolor}"
echo "Circle1 is of type: ${@circle1}"
echo "Number of objects created so far: ${.sh.type.Object_t.count}"
echo
echo "Creating triangle1 with offset of (12,12) ...."
.sh.type.Triangle_t=(offsetx=12 offsety=12)
Triangle_t  triangle1=(a=3 b=4 c=5)
echo "Area of triangle1 is: ${triangle1.area}"
echo "Perimeter of triangle1 is: ${triangle1.perimeter}"
echo "Color of triangle1 is: ${triangle1.fill}"
echo "Co-ordinates of triangle1 are: (${triangle1.offsetx}, ${triangle1.offsety})"
echo "Triangle1 is of type: ${@triangle1}"
echo "Number of objects created so far: ${.sh.type.Object_t.count}"
echo
echo "Creating triangle2 and triangle3 and comparing them ...."
Triangle_t triangle2=(a=6 b=3 c=5)
```



```
Triangle_t triangle3=(a=6 b=3 c=5)
if [[ $triangle2 == $triangle3 ]]
then
    echo "CORRECT: triangle2 and triangle3 match"
else
    echo "ERROR: triangle2 and triangle3 do not match"
fi
echo "Creating triangle4 by assignment from triangle3 and comparing them ...."
Triangle_t triangle4=triangle3
if [[ $triangle4 == $triangle3 ]]
then
    echo "CORRECT: triangle4 and triangle3 match"
else
    echo "ERROR: triangle4 and triangle3 do not match"
fi
echo "Changing color of triangle4 and comparing them again ...."
triangle4.fill="green"
if [[ $triangle4 != $triangle3 ]]
then
    echo "CORRECT: triangle4 and triangle3 differ"
else
    echo "ERROR: triangle4 and triangle3 do not differ"
fi
exit 0
```

Here is the output of this script:

```


Creating rectangle1 ....
Area of rectangle1 is: 8
Perimeter of rectangle1 is: 12
Color of rectangle1 is: blue
Co-ordinates of rectangle1 are: (0, 0)
Rectangle1 is of type: Object_t.Color_t.Shape_t.Rectangle_t
Number of objects created so far : 1

Creating circle1 ....
Area of circle1 is: 12.56
Circumference of circle1 is: 12.56
Color of circle1 is: red
Co-ordinates of circle1 are: (5, 10)
Changing circle1 fill color to green
Color of circle1 is: green
Circle1 is of type: Object_t.Color_t.Shape_t.Circle_t
Number of objects created so far: 2

Creating triangle1 ....
Area of triangle1 is: 6
Perimeter of triangle1 is: 12
Color of triangle1 is: blue
Co-ordinates of triangle1 are: (12, 12)
Triangle1 is of type: Object_t.Color_t.Shape_t.Rectangle_t.Triangle_t
Number of objects created so far: 3

Create triangle2 and triangle3 and compare them
CORRECT: triangle2 and triangle3 match
Create triangle4 by assignment from triangle3 and compare them
CORRECT: triangle4 and triangle3 match
Change color of triangle4 and compare thema gain
CORRECT: triangle4 and triangle3 differ
```

Types support the notion of singletons for elements. The element *count* in *Object_t* was defined using a *-S* argument. Elements defined with the -S option are shared by all instances of the type. Unsetting a non-shared element of a type restores it to its default value. Unsetting a shared

element has no effect. Instances of types behave like a compound variable except that only the variables defined by the type can be referenced or set. The discipline function *create* is special in that it is invoked, in a similar manner to a C++ constructor, when an instance is created. The type of any variable can be obtained by using the @ operator. Instances of types can be assigned and compared.

Types in *ksh93* are still evolving and hopefully David Korn is open to input from end users. My wish list? I would like to see types support multiple inheritance (even support for two base types would be great!) and change to using more conventional nomenclature and notation. Why not call *discipline functions* methods, and *elements* properties? Why not call *types* classes? Also I would like to be able to more fully encapsulate elements (data hiding) so that elements are out of scope unless specifically accessed via a *getter* or *setter* method. Yes, I have read that *getters* and *setters* are evil but I still think they serve a useful purpose!

