```
  .----===-----------------------------------------===------------===------.
 |    /__/|          ____       __             /   /\          /   /\     |
 |   |   |:|        /__/\      /  /\          /   /::\        /   /:/     |
 |   |   |:|        \   \:\   /  /:/         /   /:/\:\      /   /:/      |
 |   |   |:|         \   \:\ /__/::\        /   /:/~/:/     /   /:/  ____ |
 |/__/\_|:|____  ____  \__\:\ \__\/\:\__   /__/:/ /:/___ /__/:/   /   /\|
 |\   \:\/:::::/ /__/\ |   |:|      \  \:\/\ \   \:\/:::::/ \   \:\ /   /:/|
 | \   \::/~~~~  \   \:\|   |:|       \_\::/  \   \::/~~~~   \   \:\  /:/ |
 |  \   \:\       \   \:\__|:|       /__/:/    \   \:\        \   \:\/:/  |
 |   \   \:\       \__\::::/        \__\/       \   \:\        \   \::/   |
 |    \__\/            ~~~~                      \__\/          \__\/     |
 |                                                                        |
 `------------------------------------------------------------------------'
 .-----------------,--------------------------------.-------------------.
 |                 ( The Art of Scripting Vol.4 )            by Grifisx |
 `-----------------`--------------------------------'-------------------'
```

Version: Sykes-0.1a/20070402

Some things just don't translate well. Any time I have completely rephrased a section or added my own comments to better explain what Grifisx is trying to say it will show in green. - Sykes

Graphical objects and layout

Start:

Now we are going to talk about graphics, we will learn to construct a graphical interface while being able to enjoy the full range of the comforts that a graphical system can offer to us.

It is right to leave some of the basic concepts like “widget” and “layout” that we will meet often in the next hour in order to speak about some of the many graphical classes that the KVIrc offers to us.

Up till now we have seen how to create classes using the class base object like in previous volumes;

```
class(test,object)
{
	constructor()
	{
		echo $k(4,8) Test
	}
}
%Test=$new(test)
```

Obviously this script doesn't do anything other than to output “Test” to the screen. In practice we have created a new class called test that “inherits” from KVIRC's default class object .

We will try to understand what it means to inherit. I will explain it in a simple and basic way because those who script might not be a programmer and if she is she does not need for me to explain what I mean by inherit =).

The child inherits from the parent means that the child can use all of the parents abilities. That a class inherits from other classes means that it possesses all of the parents abilities (the functions and characteristics of the parent) and obviously added to these it will be able to use its own characteristics and functions. This will make more sense when we go through a few examples.

We will make 2 classes where 1 class inherits the functions of the other so that we will understand better:

```
class(parent,object)
{
	testInherit()
	{
		echo $k(4,8) I am part of the class \"parent\"
	}
}
class(child,parent)
	{
	testChild()
	{
		echo $k(4,8) I am part of the class \"child\"
	}
}
%Proof=$new(child)
%Proof->$testInherit()
%Proof->$testChild()
echo %Proof->$classname()
```

We start to see what is happening with this bit of code: we have created a class called “parent”. As the code shows, to this class we have added the function “$testInherit()”. From this we can see that the class “parent” inherits from the class object - “class(parent,object)”. Therefore - after what we have said on the inheritance concept – it will be able to use all of the functions of the class object plus its own functions (in our case it only has the one function $testInherit()). The moment we created out class “parent” it can use its own functions plus those from the object class, but we will continue with the example in order to understand this better.

Next we created a class called “child” and we have made it so that it inherits from the class “parent”. In this way - “class (child, parent)” - according to the inheritance concept, the class “child” will enjoy its own functions and those of the class parent.

Next we have created an object %Proof using the class “child” (%Proof=$new(child)) and then used the function $testInherit() - which belongs to the parent – and the function $testChild() - which belongs to the child.

We have also used the function $classname() which belongs to the class “object” and returns the name of the class from where it was called. In fact since the class “parent” inherits from the class “object”, the class “child” will also enjoy all of object's functions (genetically it would be a grandparent =P).

Now you should go see KVIrc's manual for the class “object” to see what we inherit =P (from the manual Object Classes-> object class).

Now we move to the graphics.

While we must understand what is a “widget” we can simply say that a widget is a graphical object. We can make a test:

```
%graphical_object = $new(widget)
%graphical_object->$show()
```

Paste this into KVIrc's script tester and execute. As you will see a window appears – it is really just an empty graphical object. If we check the widget class in the manual we see many more functions (from the help manual Object Classes->widget class). This is the base for graphical objects and – like we will see – all graphical classes inherit from this. There are the graphical classes button, listview, label and lineedit, all which use their own functions as well as those belonging to the widget class.

We begin to become a little more familiar with the graphical object widget and its functions, enriching the code with some other functions that are found on the KVIrc help page.

```
%graphical_object = $new(widget)
%graphical_object->$setGeometry(250,150,500,155)
%graphical_object->$setCaption("here i am\!")
%graphical_object->$setIcon(217)

%graphical_object->$show()
```

As we can see we have used just 4 of the many functions that are available to us from the widget class in order "to model" our graphical object:
$setGeometry (<x>, <y>, <width>, <height>)
This allows us to set the position and dimensions of our object
$setCaption(<text>)
This allows us to set the title of the object.
$setIcon(<image_id>)
This allows us to set the icon whose number we get from the icon table in KVIrc (Tools_>Show Icon Table). This icon is displayed on your taskbar.
$show ()
This is the most important function as it shows the object and eventually all of its children.

The children of an object are the objects it is tied too and we do not need to get confused with the concept of inheritance that we saw before, because if an object is the child of the other it does not inherit a function from the last one. It is only a legacy ... an example will give you an idea.

```
%graphical_object = $new(widget)
%graphical_object->$setGeometry(250,150,200,155)
%graphical_object->$setCaption("Here I am\!")
%graphical_object->$setIcon(217)
%button=$new(button,%graphical_object)
%button->$setGeometry(50,50,90,40)
%button->$setText("Click Me\!")
%graphical_object->$show()
```

As usual the additions are in bold. Notice how we have reduced the main widget from 500 to 200.

%button=$new(button,%graphical_object)

Here we have created an object of the type "button" (look at the class button in the KVIrc manual to see what functions it can use) and as we can see we have created it as a child of "%graphical_object". That is, we have tied it to the main widget. In fact if we execute the code the button will be found inside the main widget.

Notice that the $show() function is only used with the main object. It isn't necessary to also use it with the child. Now we must make do something, that is , we must make it react to our click. In order to do this we will use the function privateimpl whose syntax is:

privateimpl (<object>, <function>) {<implementation>;};

In practice this function serves to re-implement a function that belongs to the object. That is, a way of executing a series of commands<implementations>. To explain it poorly, it is a re-implemented function in place of the one "that originates them". We will make an example in order to understand this better:

```
%graphical_object = $new(widget)
%graphical_object->$setGeometry(250,150,200,155)
```

```
%graphical_object->$setIcon(217)
privateimpl(%graphical_object, setCaption) {
echo Proof
}
%graphical_object->$setCaption("Here I am\!")
%graphical_object->$show()
```

As you can see we have re-implemented the function setCaption from the graphical_object object. In fact if we execute the code we will notice that when it makes use of the $setCaption() function it puts the output to the screen and not its usual function of setting the titlebar in the object (remember that the echo will output to the console window of KVIrc).

After this small digression carry on to re-implement the function

$mousePressEvent (<pressedbutton>, <x>, <y>)

Notice that by default this function does nothing. It is called like an automatic rifle when the button of the mouse is pressed. The widget <pressedbutton> will be 0 if the user has pressed the left mouse button, 1 for the right and 2 for the middle one. x and y are relative coordinates to the widget on which we have pressed the button and are expressed in pixels.

Now we modify our code:

```
%graphical_object = $new(widget)
%graphical_object->$setGeometry(250,150,200,155)
%graphical_object->$setIcon(217)
%graphical_object->$setCaption("Here I am\!")
%button=$new(button,%graphical_object)
%button->$setGeometry(50,50,90,40)
%button->$setText("Click Me\!")
%Label=$new(label,%graphical_object)
%Label->$setText("Here is a label")
%Label->$setGeometry(50,20,120,20)
privateimpl(%button,mousePressEvent)
{
if($0==0)
{
%Label->$setText("Left Mouse Button");
}
if($0==1)
{
%Label->$setText("Right Mouse Button");
}
if($0==2)
{
%Label->$setText("Middle Mouse Button");
}
}
%graphical_object->$show()
```

Here we have put practically all of it =D. As you can see we have added a label and called it %Label (w fantasy). Also notice that we have made it global (it begins with capital letters). Since we must change it through the press of a button it must persist and not die after creation (try to call it %label and see the effect).

After we have used the function privateimpl (and remembering that 0 is the left mouse button, 1 the right and 2 the middle), we have created our implementation of the function mousePressEvent. Then if the pressed key is the left (**if ($0==0)**), set the text of the label to "Left Mouse Button", if it is the right button "Right Mouse Button" and in the case of it being the middle then "Middle

Mouse Button".

Now we must introduce the concept of layout – the layout is a fantastic class – it allows us to not be driven crazy with geometric coordinates and it allows us to align and position in a clean way the elements of one widget.

As it says in the manual: "The layout is a geometry management tool for child widgets. You create a layout , give it some widgets to manage and it will layout them automatically.
The parent of the layout must be the widget for which child widget geometries have to be managed. A layout is a grid of NxM cells in which you insert child widgets with **$addwidget()**.
Widgets that must span multiple cells can be added to the layout with **$addmulticellwidget()**."

We need to see an example:

```
# We create 3 objects
%graphical_object = $new(widget)
%layout=$new(layout,%graphical_object)
%button=$new(button,%graphical_object)
# Main object
%graphical_object->$setIcon(217)
%graphical_object->$setCaption("Here I am")
# Label
%Label=$new(label,%graphical_object)
%Label->$setText("Here is a label")
# Button
%button->$setText("Click Me\!")
privateimpl(%button,mousePressEvent)
{
        if($0==0) {%Label->$setText("Left Mouse Button");}
        if($0==1) {%Label->$setText("Right Mouse Button");}
        if($0==2) {%Label->$setText("Middle Mouse Button");}
}
# We add the child objects to the main layout
%layout->$addWidget(%Label,0,0)
%layout->$addWidget(%button,1,0)
# Monster All
%graphical_object->$show()
```

As you can see we have eliminated all the $setGeometry, leaving to layout the management of the geometries. The function used is $addwidget() and it has the syntax:

$addWidget (<graphical_object>, <row>, <column>)

Think of the layout as a virtual grill. In our example we have put the label to row 0 and column 0 of our virtual grill. Notice also that if we increase or shrink the window the graphical objects will adapt, maintaining the correct proportions and position. When the label changes the main widget changes size to adapt to the new size.

If we wanted an example that the button size was half that of the label – so that it took 1 virtual cell, while the label took 2 – we could use the function:

$addMultiCellWidget (<object>, <from_row>, <to_row>, <from_con the >, <to_con the >)

And the code would turn out to be thus:

```
%graphical_object = $new(widget)
%layout=$new(layout,%graphical_object)
%button=$new(button,%graphical_object)

%graphical_object->$setIcon(217)
%graphical_object->$setCaption("Here I am")

%Label=$new(label,%graphical_object)
%Label->$setText("I take 2 cells!!!!!!!!!!!!!!!!!!")

%button->$setText("Click Me\!")
privateimpl(%button,mousePressEvent)
{
        if($0==0) {%Label->$setText("Left Mouse Button");}
        if($0==1) {%Label->$setText("Right Mouse Button");}
        if($0==2) {%Label->$setText("Middle Mouse Button");}
}

%layout->$addMultiCellWidget(%Label,0,0,0,1)
%layout->$addMultiCellWidget(%button,1,1,0,0)

%graphical_object->$show()
```

As you can see now the button occupies 1 cell (%button,1,1,0,0 at the row[1,1] and the column[0,0]), while the label occupies 2 (%button,1,1,0,0 at the row[0,0] but 2 columns [0,0 and 1,1 in fact (%Label,0,0,0,1)).

We try to make our window more complex:

```
# We create 3 objects
%graphical_object = $new(widget)
%layout=$new(layout,%graphical_object)
# Main object
%graphical_object->$setIcon(217)
%graphical_object->$setCaption("Qt Mirror")
# Create the label
%Labelphrasebase=$new(label,%graphical_object)
%Labelphrasebase->$setText("Phrase Base")
%Labelphrasereverse=$new(label,%graphical_object)
%Labelphrasereverse->$setText("Phrase Reverse")
# Create the Quadrants
%Quadrantphrasebase=$new(multilineedit,%graphical_object)
%Quadrantphrasereverse=$new(multilineedit,%graphical_object)
%Quadrantphrasereverse->$setReadOnly(1)
# Create the buttons
%button=$new(button,%graphical_object)
%button->$setText("Reverse\!")
privateimpl(%button,mousePressEvent)
{
        if($0==0)
        {
                %texttouse = %Quadrantphrasebase->$text()
                %idx=$($str.len(%texttouse)+1)
                while(%idx!=0)
                {
                        %sztmp=$str.section(%texttouse,"",%idx,%idx)
                        %szTextreverse=%szTextreverse%sztmp;
                        %idx--;
                }
                %Quadrantphrasereverse->$setText(%szTextreverse)
        }
}
# Add the child objects to the layout
%layout->$addMultiCellWidget(%Labelphrasebase,0,0,0,6)
%layout->$addMultiCellWidget(%Labelphrasereverse,0,0,7,13)
%layout->$addMultiCellWidget(%Quadrantphrasebase,1,1,0,6)
%layout->$addMultiCellWidget(%Quadrantphrasereverse,1,1,7,13)
%layout->$addMultiCellWidget(%button,2,2,0,0)
%graphical_object->$show()
```

There is not much to add to what has already been said. The code speaks for itself and the routine to reverse the phrase is similar to what we have seen in other tutorials and all the other functions used we've already seen.

The part $setReadOnly() as you can imagine sets the right side to read only. The function uses a boolean value (its syntax is $setReadOnly(<bool_value>)). So 1(TRUE) or 0(FALSE) to set the property to read only or not. The default is false.

We have finished for now.

Satisfaction is ours...
 /ECHO STOP

 - - - - - - - -- - - - - - - - -- - - - - - - - - - -- - - - - - -- - - - - -

"You see things; and you say `Why? ' But I dream things that never were; and I say `Why not? "
(George Bernad Shaw)

- - - - - - - - -- - - - - - - - -- -- - - - - - - - - - - - - - - - - - - -

Grifisx