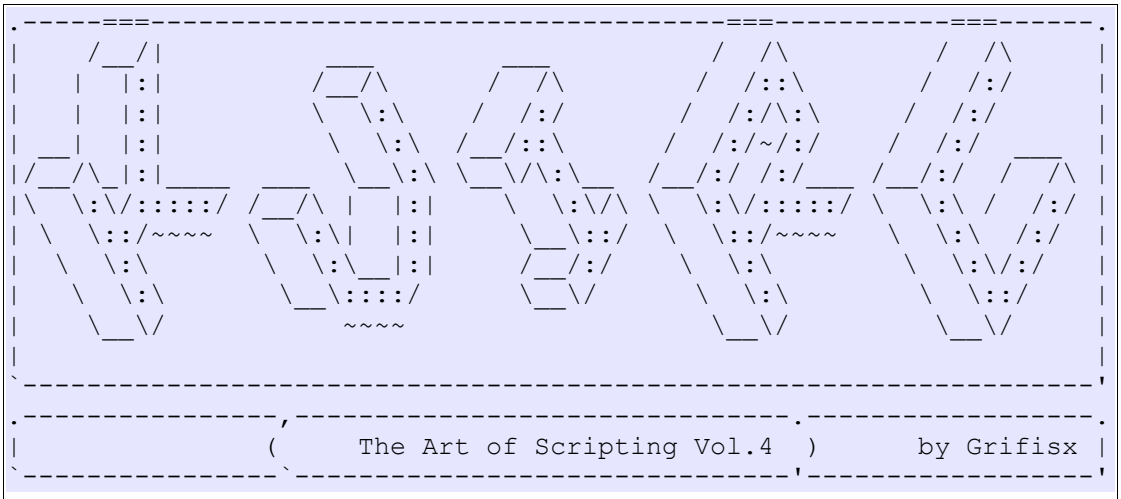

Oggetti grafici, eredità e layout

Start:
Aggiornamento 20.07.2010

Questa volta parleremo della grafica, impareremo a costruirci delle interfacce gradevoli per poter godere a pieno delle comodità che una GUI può offrirci.
Credo sia doveroso partire dalla spiegazione di alcuni concetti di base, come "widget" e "layout" che incontreremo spesso d'ora in poi, per poi parlare dei alcune tra le tante classi grafiche che il KVIrc ci offre.
Abbiamo visto fino ad adesso la creazione di classi, utilizzando la classe base object come negli esempi dei volume precedenti, eccone un altro:

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

Questo script non fa altro che dare l'output a schermo della scritta *Test* in pratica abbiamo creato una *classe test*, che "eredita" dalla *classe object* del KVIrc.
Cerchiamo di capire in parole povere cosa significa ereditare, lo spiegherò in modo semplice ed elementare perché chi scripta non necessariamente deve essere un programmatore, e qualora lo sia non ha certamente bisogno che io gli spieghi cosa significhi =).
Che Tizio eredita da Caio significa che Tizio entra in possesso di tutto ciò che ha Caio, ed ad esso aggiungerà tutto ciò che ha lui.
Che una classe eredita da un'altra significa che entra in possesso di tutte le sue capacità (quindi le funzioni e le caratteristiche di quest'ultima) ed in più, ovviamente, a queste potrà aggiungere le proprie caratteristiche e funzioni.

Proviamo a fare 2 classi che ereditano una le funzioni dell'altra così capiremo meglio:

```
class(padre,object)
{
      testEredita()
      {
            echo $k(4,8) Faccio parte della classe \"padre\"
      }
}
class(figlio,padre)
{
      testFiglio()
      {
            echo $k(4,8) Faccio parte della classe \"figlio\"
      }
}

%prova=$new(figlio)
%prova->$testEredita()
%prova->$testFiglio()
echo %prova->$classname()
```

Vediamo un po' di capire questo semplice codice:
intanto abbiamo creato una classe chiamata "*padre*", a questa classe, come possiamo vedere dal codice, abbiamo dato solo una funzione cioè a dire "*$testEredita()*", notiamo che questa classe "*padre*" eredita dalla classe object, in questo modo "**class(padre,object)**", quindi, secondo quanto abbiamo detto sul concetto di ereditarietà, essa avrà a disposizione tutte le funzioni della classe object più le proprie funzioni (che nel nostro caso sono una soltanto cioè $testEredita()), cioè nel momento in cui creassimo un oggetto con la nostra classe "*padre*" esso godrebbe delle sue funzioni più quelle ereditate dalla classe object, ma continuiamo con l'esempio per capire ancora meglio.
Poi abbiamo creato una classe "*figlio*" e l'abbiamo fatta ereditare dalla classe "*padre*", in questo modo "**class(figlio,padre)**", a questo punto, secondo il concetto di ereditarietà, la classe "*figlio*" godrà delle funzioni proprie e di quelle della classe padre.
In seguito, infatti, abbiamo creato un oggetto *%Prova* utilizzando la classe "figlio"(**%prova=$new(figlio)**) e poi usato le funzioni *$testEredita()* che in realtà appartiene alla classe "*padre*" e la funzione *$testFiglio()* che invece appartiene ad essa stessa.
Non basta, come vedete abbiamo anche usato la funzione *$classname()* che serve a ritornare il nome di una classe e che appartiene alla classe "*object*", infatti, dato che la classe "*padre*" eredita dalla classe "*object*", la classe "*figlio*" godrà anche delle funzioni di questa (geneticamente sarebbe un nonno =P).
A questo punto sarebbe d'obbligo andare a vedere, sul manuale del KVIrc le funzioni della classe object, anche per sapere un po' cosa ereditiamo =P (dal manuale *Object Classes-> object class*).
adesso passiamo alla grafica.
Intanto dobbiamo capire cosa è una "*widget*", diciamo semplicemente che per widget si intende un oggetto grafico, facciamo subito una prova:

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

incolliamo nello script tester ed eseguiamo.
Come vedrete apparirà come una finestra, più precisamente appare un

oggetto grafico vuoto, se andiamo a vedere la classe widget nel manuale notiamo che è forse quella più ricca di funzioni (Dal manuale dell' Help *Object Classes->widget class*) dato che questa è la classe base per gli oggetti grafici e, come vedremo, tutte le classi grafiche ereditano da questa, quindi ciascuna classe grafica, *button*, *listview*, *label* o *lineedit* che sia, godrà oltre che delle proprie funzioni anche di quelle della classe *widget.*
Cominciamo a prendere un po' di familiarità con l'oggetto grafico widget e le sue funzioni arricchendo il codice di prima con qualche altra funzione di quelle che si trovano sull'help del KVIrc.

```
%oggettoGrafico = $new(widget)
%oggettoGrafico->$setGeometry(250,150,500,155)
%oggettoGrafico->$setWindowTitle("Qt Eccomi\!")
%oggettoGrafico->$setWindowIcon(217)

%oggettoGrafico->$show()
```

Come possiamo vedere abbiamo usato 4 delle tantissime funzioni che ci da la classe widget per "modellare" il nostro oggetto grafico:
$setGeometry(<x>,<y>,<largehzza,<altezza>)
che ci permette di settare la posizione e le dimensioni del nostro oggetto,
$setWindowTitle(<text>)
che ci permette di settare il titolo del nostro oggetto
$setWindowIcon(<image_id>)
che ci permette di settare l'icona, il cui numero lo ricaviamo dalla tabella delle icone del kvirc (S*trumenti->Mostra tabella delle icone*)
$show()
che è la funzione più importante poiché mostra l'oggetto e tutti i sui eventuali figli.
I figli di un oggetto sono gli oggetti che ad esso sono legati e non dobbiamo confonderci col concetto di ereditarietà visto prima, perché se un oggetto è figlio di un altro non eredita nessuna funzione da quest'ultimo, gli è solamente legato ...un esempio chiarirà le idee.

```
%oggettoGrafico = $new(widget)
%oggettoGrafico->$setGeometry(250,150,200,155)
%oggettoGrafico->$setWindowTitle("Qt Eccomi\!")
%oggettoGrafico->$setWindowIcon(217)
%bottone=$new(button,%oggettoGrafico)
%bottone->$setGeometry(50,50,90,40)
%bottone->$setText("Cliccami\!")
%oggettoGrafico->$show()
```

Come al solito le aggiunte sono in grassetto, andiamo ad esaminare cosa abbiamo fatto a parte ridurre da 500 a 200 la larghezza della widget principale:
%bottone=$new(button,%oggettoGrafico)
qui abbiamo creato un oggetto di tipo "*button*" (andate a vedere la classe button nel manuale del KVIrc per vedere di quali funzioni essa gode) e come vedete l'abbiamo creata come figlio di "**%oggettoGrafico**", cioè l'abbiamo legata alla widget principale, infatti se eseguiamo il codice il pulsante si troverà all'interno di quest'ultima.
Le altre 2 funzioni usate credo che si spieghino da sole.
Notate che lo $show() è stato fatto solo dell'oggetto principale, non è stato necessario farlo anche del figlio.

Adesso però dobbiamo far fare qualche cosa al pulsante, cioè dobbiamo farlo reagire ai nostri click.
Per fare questo useremo la funzione *privateimpl* la cui sintassi è:
privateimpl(<oggetto>,<functione>){<implementazione>;};
In pratica questa funzione serve per reimplementare una funzione che appartiene all'oggetto, ovvero a fargli eseguire la serie di comandi **<implementazione>**(in parole povere viene eseguita la funzione reimplementata al posto di quella "originale"), facciamo un esempio per capire meglio:

```
%oggettoGrafico = $new(widget)
%oggettoGrafico->$setGeometry(250,150,200,155)
%oggettoGrafico->$setWindowTitle(217)

privateimpl(%oggettoGrafico, setCaption) {
echo Prova
}

%oggettoGrafico->$setCaption("Qt Eccomi\!")
%oggettoGrafico->$show()
```

come potete vedere abbiamo reimplementato la funzione *setCaption* dell'oggetto *%oggettografico*, infatti se eseguiamo il codice noteremo che quello che farà adesso la funzione $setCaption() è dare l'output a schermo della parola prova e non la sua funzione normale, cioè quella di settare il titolo nella barra dell'oggetto (vi ricordo che l'echo verrà emesso sulla finestra della *console* del KVIrc quindi darà li che lo vedrete).
Dopo questa piccola digressione, andiamo a usare questa funzione per reimplementare la funzione
$mousePressEvent(<bottone_premuto>,<x>,<y>)
(notate che questa funzione di default non fa niente, essa viene richiamata in modo automatico quando il pulsante del mouse viene premuto ed il mouse si trova nella widget <bottone_premuto> sarà 0 se è stato premuto il pulsante sinistro del mouse, 1 per quello destro e 2 per quello centrale, x e y sono le coordinate relative alla widget su cui abbiamo premuto il pulsante e sono espresse in pixel)
ora modifichiamo il nostro codice:

```
%oggettoGrafico = $new(widget)
%oggettoGrafico->$setGeometry(250,150,200,155)
%oggettoGrafico->$setWindowTitle("Qt Eccomi\!")
%oggettoGrafico->$setWindowIcon(217)
%bottone=$new(button,%oggettoGrafico)
%bottone->$setGeometry(50,50,90,40)
%bottone->$setText("Cliccami\!")
%Label=$new(label,%oggettoGrafico)
%Label->$setText("Sono una label")
%Label->$setGeometry(50,20,120,20)
privateimpl(%bottone,mousePressEvent)
{
      if($0==0)
      {
            %Label->$setText("Tasto sinistro premuto");
      }
      if($0==1)
      {
            %Label->$setText("Tasto destro premuto");
```

```
	}
	if($0==2)
	{
	%Label->$setText("Tasto centrale premuto");
	}
}
%oggettoGrafico->$show()
```

Ecco che abbiamo messo in pratica il tutto =D, come potete vedere abbiamo aggiunto una label, e l'abbiamo chiamata *%Label* (w la fantasia) e come potete notare l'abbiamo creata globale (comincia per lettera maiuscola) poiché, dato che dobbiamo cambiarla tramite la pressione del tasto, ci serve che "persista" e non muoia dopo la creazione (provate a chiamarla %label e vedetene gli effetti).
Dopo abbiamo usato la funzione *privateimpl* e, ricordandoci che 0 è il tasto sinistro del mouse, 1 quello destro e 2 quello centrale, abbiamo creato la nostra implementazione della funzione *mousePressEvent*, che se il tasto premuto sarà quello sinistro (**if($0==0)**) setterà il testo della label a **"Tasto sinistro premuto"** se sarà quello destro a **"Tasto destro premuto"** mentre in caso sia quello centrale a **"Tasto centrale premuto"**.

Introduciamo il concetto di layout, il layout è una classe fantastica, essa ci permette di non impazzire dietro le coordinate geometriche e ci permette di allineare e posizionare in modo pulito gli elementi di una widget.
Come dice il manuale (tradotto ed adattato):
"Il layout è una classe per gestire la geometria degli oggetti figli di una widget. Tu crei un layout, gli dai degli oggetti da gestire, e lui li dispone in modo geometricamente pulito.
Il padre del layout deve essere la widget di cui si vogliono sistemare geometricamente i figli. Esso (il layout) è una griglia virtuale di NxM celle nella quale puoi inserire gli oggetti grafici con la funzione ***$addwidget().***
Oppure se essi devono occupare più di una cella con la funzione ***$addmulticellwidget().***"
Vediamone subito un esempio:

```
// Creo i tre oggetti
%oggettoGrafico = $new(widget)
%layout=$new(layout,%oggettoGrafico)
%bottone=$new(button,%oggettoGrafico)
// Oggetto principale
%oggettoGrafico->$setWindowTitle("Qt Eccomi\!")
%oggettoGrafico->$setWindowIcon(217)
// Label
%Label=$new(label,%oggettoGrafico)
%Label->$setText("Sono una label")
// Bottone
%bottone->$setText("Cliccami\!")
privateimpl(%bottone,mousePressEvent)
{
	if($0==0) {%Label->$setText("Tasto sinistro premuto");}
	if($0==1) {%Label->$setText("Tasto destro premuto");}
	if($0==2) {%Label->$setText("Tasto centrale premuto");}
}
// Aggiungo gli oggetti figli al layout
%layout->$addWidget(%Label,0,0)
```

```
%layout->$addWidget(%bottone,1,0)
// Mostro il tutto
%oggettoGrafico->$show()
```

Come potete vedere abbiamo eliminato tutti i *$setGeometry*, demandando al layout la gestione delle geometrie.
La funzione utilizzata è $addWidget() che ha la sintassi:
$addWidget(<oggetto_grafico>,<riga>,<colonna>)
Ricordando che il layout è come una griglia virtuale, nel nostro esempio, abbiamo disposto la label alla riga 0 e alla colonna 0 mentre il bottone proprio sotto, cioè alla riga 1 della colonna 0 della nostra griglia virtuale, notate anche che se allarghiamo o stringiamo la finestra, gli oggetti grafici si adatteranno, mantenendo le proporzioni e la posizione.
Se volessimo ad esempio che il pulsante fosse la metà della label, cioè prendesse 1 cella virtuale, mentre la label ne prendesse 2, potremmo usare la funzione:
$addMultiCellWidget(<oggetto>,<from_row>,<to_row>,<from_col>,<to_col>)
E il codice risulterebbe essere così:

```
%oggettoGrafico = $new(widget)
%layout=$new(layout,%oggettoGrafico)
%bottone=$new(button,%oggettoGrafico)

%oggettoGrafico->$setWindowTitle("Qt Eccomi\!")
%oggettoGrafico->$setWindowIcon(217)

%Label=$new(label,%oggettoGrafico)
%Label->$setText("Prendo 2 Celle!!!!!!!!!!!!!!!!!!!")

%bottone->$setText("Cliccami\!")
privateimpl(%bottone,mousePressEvent)
{
      if($0==0) {%Label->$setText("Tasto sinistro premuto");}
      if($0==1) {%Label->$setText("Tasto destro premuto");}
      if($0==2) {%Label->$setText("Tasto centrale premuto");}
}

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

%oggettoGrafico->$show()
```

Come vedete, adesso il pulsante occupa 1 cella (**%bottone,1,1,0,0** una riga [la 1,1] ed una colonna [la 0,0]) mentre la label ne occupa 2 (**%Label,0,0,0,1** una riga [la 0,0] ma due colonne vicine [la 0,0 e la 1,1 infatti (%Label,0,0,**0,1**)).
Proviamo a fare una finestrella più complessa.

```
# Creo i tre oggetti
%oggettoGrafico = $new(widget)
%layout=$new(layout,%oggettoGrafico)
# Oggetto principale
%oggettoGrafico->$setWindowTitle("Qt Eccomi\!")
%oggettoGrafico->$setWindowIcon(217)
# Creo le Label
%Labelfrasebase=$new(label,%oggettoGrafico)
```

```
%Labelfrasebase->$setText("Frase Base")
%Labelfrasereversata=$new(label,%oggettoGrafico)
%Labelfrasereversata->$setText("Frase Reversata")
# Creo i Quadranti
%QuadranteFraseBase=$new(multilineedit,%oggettoGrafico)
%QuadranteFraseReversata=$new(multilineedit,%oggettoGrafico)
%QuadranteFraseReversata->$setReadOnly(1)
# Creo i pulsanti
%bottone=$new(button,%oggettoGrafico)
%bottone->$setText("Reversa\!")
privateimpl(%bottone,mousePressEvent)
{
      if($0==0)
      {
            %testoDareversare = %QuadranteFraseBase->$text()
            %idx=$($str.len(%testoDareversare)+1)
            while(%idx!=-1)
            {
                  %sztmp=$str.mid(%testoDareversare,%idx,1)
                  %szTestoreversato=%szTestoreversato%sztmp;
                  %idx--;
            }
            %QuadranteFraseReversata->$setText(%szTestoreversato)
      }
}
# Aggiungo gli oggetti figli al layout
%layout->$addMultiCellWidget(%Labelfrasebase,0,0,0,6)
%layout->$addMultiCellWidget(%Labelfrasereversata,0,0,7,13)
%layout->$addMultiCellWidget(%QuadranteFraseBase,1,1,0,6)
%layout->$addMultiCellWidget(%QuadranteFraseReversata,1,1,7,13)
%layout->$addMultiCellWidget(%bottone,2,2,0,0)

%oggettoGrafico->$show()
```

Non credo ci sia molto da aggiungere a quanto già detto, il codice parla chiaro e la routine per capovolgere la frase è la solita che abbiamo visto negli altri tutorial e tutte le altre funzioni usate le abbiamo già viste, a parte *$setReadOnly()* che come potete immaginare serve per settare il secondo quadrante come di sola lettura, la funzione vuole un valore booleano (la sua sintassi è *$setReadOnly(<bool_value>))* quindi 1(VERO) oppure 0(FALSO) a seconda che vogliamo impostare la proprietà di sola lettura oppure no, considerando che di default è falso.
Per concludere il discorso sul layout vi consiglio di guadarvi le classi hbox e vbox, che nn sono altro che layout ottimizzati per allineare oggetti in vericale (vbox) ed in oriziontale (hbox).
Quando creerete, ad esempio, due oggetti figli di un vbox, appariranno posizionati in verticale, il secondo sotto il primo.
Vi propongo qualche esempio per darvi l'idea del loro utilizzo

```
// Primo esempio:
%hbox=$new(hbox,0)
%ledit=$new(lineedit,%hbox)
%btn=$new(button,%hbox)
%btn->$setText("Premi")
%hbox->$show()
```

```
 // Secondo esempio:
%vbox=$new(vbox,0)
%ledit=$new(lineedit,%vbox)
%btn=$new(button,%vbox)
%btn->$setText("Premi")
%vbox->$show()


// Terzo esempio: due vbox in un hbox per creare una griglia
%hbox=$new(hbox,0)
%left_vbox=$new(vbox,%hbox)
%right_vbox=$new(vbox,%hbox)
%ledit=$new(lineedit,%left_vbox)
%btn=$new(button,%left_vbox)
%btn->$setText("Left")
%ledit=$new(lineedit,%right_vbox)
%btn=$new(button,%right_vbox)
%btn->$setText("Right")
%hbox->$show()
```

Ponete l'attezione sull'ultimo esempio: questo metodo è comodissimo, molto di più del multicell, perchè vi create una griglia "incastrando" tra di loro I vbox e gli hbox, allo stesso tempo potete utilizzare nomi di variabili che, mnemonicamente, vi aiutino a posizionare gli oggetti nel posto giusto, è come se creaste una scaffalatura e la suddivideste in piccoli box nei quali sistemare I vostri oggetti.
Respiroooo.....
Passiamo ad un altro approfondimento, rendendomi conto che, in questi tutorial che feci ben 5 anni fa, manca una parte in cui si parla dell'eredita in relazione agli oggetti grafici ed al loro utilizzo, ho deciso di aggiungere questa nuova parte.
Anche per gli oggetti grafici valgono i principi dell'eridatarietà.
Facciamo qualche esempio, creando una classe "mybutton" che eredita dalla classe button fornita dal KVIrc:

```
class(mybutton,button)
{
}
%mybutton=$new(mybutton)
%mybutton->$show()
```

eseguiamo e ...funziona, ora vediamo se la nostra classe ha ereditato le funzioni della classe button, proviamone alcune.
Dato che la classe è già stata creata e la ritrovate nel class-editor, nello script tester non è necessario rimettere il codice relativo alla classe

```
%mybutton=$new(mybutton)
%mybutton->$setText("Premi quì!")
%mybutton->$setImage(218)
%mybutton->$show()
```

Eseguiamo e...bene, funziona, la nostra classe ha ereditato tutte le funzioni della classe button.
Ma a cosa ci serve "clonare" una classe che già esiste? Perchè farci una mybutton?

Proviamo a dare un senso a questa cosa...
Noi sappiamo anche che le funzioni possono ricevere dei parametri giusto?
Quindi forniamo, per prima cosa, la nostra classe mybutton di un costruttore, io vi consiglio di farlo dal class-editor per cominciare a distaccarvi dallo script-tester per la creazione delle classi.

```
class(mybutton,button)
{
	constructor()
	{
		/* Il nostro costruttore dovrà ricevere due parametri
		il primo $0 contenente il testo del pulsante ed il
		secondo, $1, l'icona. La @, come ricorderete
		sta ad indicare la stessa classe e corrisponde a
		$this()-> ed a $$->, in pratica chiama una funzione
		($setText) di se stessa                          */
		@$setText($0)
		@$setImage($1)
	}
}
%mybutton=$new(mybutton,0,0,”Premi Quì!”,218)
%mybutton->$show()
```

...ottimo, da questo momento in poi, utilizzando la nosta classe possiamo risparmiarci di scrivere un po' di codice, invece di

```
%mybutton=$new(button)
%mybutton->$setText("Premi quì!")
%mybutton->$setImage(218)
%mybutton->$show()
```

basterà

```
%mybutton=$new(mybutton,0,0,”Premi Quì!”,218)
%mybutton->$show()
```

Oppure, potremmo crearci un oggetto “costruito da più oggetti, vi faccio un esempio, la label del KVIrc non permette di settare un icona, cioè, in teoria, non posso avere un icona associata ad una label.
Rimediamo:

```
class(iconlabel,hbox)
{
	constructor()
	{
		%icon=$0
		%text=$1
		%ico_lb=$new(label,$$)
		@$setAlignment(%ico_lb,"right")
		@$setStretchFactor(%ico_lb,10)
		%ico_lb->$setImage(%icon)

		%txt_lb=$new(label,$$)
		@$setAlignment(%txt_lb,"left")
		@$setSpacing(2)
		@$setStretchFactor(%txt_lb,90)
```

```
            %txt_lb->$setText(%text)

            if($2) @$setTooltip($2)
      }
}
```

Ho creato una classe che eredità da hbox, che, vi ricordo,non è altro che un layout costituito da celle che si sviluppa in orizzontale, e nel proprio costruttore ho creato una label contenente del testo ed una contenente una icona, perchè le label, come vi ho già detto, non possono contenerle entrambe contemporaneamente.
Alla mia classe posso passare 3 parametri, $0 che contiene l'icona, $1 che contiene il testo e $2 che contiene un testo per l'eventuale tooltip che vogliamo far apparire al passaggio del mouse sopra la nostra label.
Le altre funzioni ve le andrete a vedere dalla doc delle relative classi, hbox e label.
Ora non ci resta che aprire uno script tester e provare quanto codice risparmieremo da adesso in poi tutte le volte che vorremo creare un label con icona associata:

```
%mylabel=$new(iconlabel,0,0,218,Nuova Label, Creazioneeee)
%mylabel->$show()
```

bene... una riga invece di 12, direi che può andare bene ;).
Inoltre vi faccio notare nell'Help di KVIrc, quando andate su una classe, ad esempio prendete la classe label, all'inizio troverete:

label class

Displays text or an image

Inherits

object widget

Questo significa che la classe label eredita a sua volta tutte le funzioni delle classi object e widget, quindi oltre alle sue proprie potrete anche usare quelle ereditate dalle due classi, facciamo un esempio:
nella classe object, guardando nella documentazione troviamo queste funzioni:

$timerEvent(<timerId>)
Handler for the timer events. The default implementation does nothing. See also $startTimer() and $killTimer().
$startTimer(<timeout>)
Starts a builtin timer for this object and returns its timer id as a string or '-1' if the <timeout> was invalid. The $timerEvent() handler function will be called at each <timeout>. The <timeout> is in milliseconds.
$killTimer(<timer id>)
Stops the timer specified by <timer id>.

La classe hbox, che abbiamo utilizzato per creare una icon label eredita anche da object, quindi andiamo a trasformare la nostra classe iconlabel in una timelabel =) :

```
class(timelabel,hbox)
{
	constructor()
	{
		%ico_lb=$new(label,$$)
		@$setAlignment(%ico_lb,"right")
		@$setStretchFactor(%ico_lb,10)
		%ico_lb->$setImage(93)

		/*		%txt_lb ora devo renderla visibile a tutte le funzioni della classe perchè devo vederla nel timerEvent, quindi la dichiaro come @%, quindi variabile di questa classe e visibile all'interno di essa: non locale. Da notare che è possibile leggere/modificare una variabile di classe anche dall'esterno puntando ad essa come, in questo esempio: %Timelabel->%txt_lb. Potrei anche, per realizzare lo stesso scopo, renderla globale ma non c'è motivo di essere "visibile" da tutto il KVIrc, in generale l'uso delle variabili globali deve essere ridotto all'indispensabile
		*/
		@%txt_lb=$new(label,$$)
		@$setAlignment(%txt_lb,"left")
		@$setSpacing(2)
		@$setStretchFactor(%txt_lb,90)
		@%txt_lb->$setText($date("H:M:S"))
		@$setTooltip($date("d/m/Y") )
		/*	Se l'eredità funziona, la nostra classe, ereditando da hbox, che come dice la doc di KVIrc eredita a sua volta da Object, avrà anche la funzione $startTimer() che itroviamo proprio in quest'ultima. Inoltre mi conservo l'ID del timer nella variabile @%timer in modo da poterlo killare all'occorrenza*/
		@%timer=@$startTimer(1000)
	}
	destructor()
	{
		// Per sicurezza, al delete dell'oggetto killiamo il timer
		@$killTimer(@%timer)
	}
	timerEvent()
	{
		@%txt_lb->$setText($date("H:M:S"))
		/* Implementiamo il timerEvent, l'evento tipico della classeditor.load object che si verifica ogni volta che scatta il timer che abbiamo creato */
	}
}
```

creiamo l'oggetto:

```
%Timelabel=$new(timelabel)
%Timelabel->$show()
```

..bello vero?
Attenzione che se chiudete la label dalla X della finestrella il timer non smetterà di scorrere, perchè l'oggetto che avete creato è ancora in memoria, dovrete cancellarlo con
/delete %Timelabel.

Come potete notare ho usato una variabile Globale per crearmi quest'oggetto, proprio per non "perderne" l'ID che mi sarebbe servito per il successivo delete.
Se l'avessimo creato come %timerlabel, quindi come variabile locale, non avremmo avuto poi la possibilità di cancellarlo.
Fate una prova =).
Volendo potreste "intercettare" l'evento $closeEvent() della classe da cui ereditiamo riscrivendolo: un event è una funzione che viene chiamata automaticamente dall'oggetto quando si verificano determinati eventi (come il constructor() alla creazione e destrctor() alla eliminazione), alcuni esempi sono: closeEvent, mousePressEvent, focusInEvent etc etc.. guardate la documentazione della classe widget per averne un panorama.
Riscrivendo la funzione legata all'evento la classe farà esattamente quello che gli direte di fare nel codice, quindi potreste aggingere una cosa del genere:

```
closeEvent()
{
      echo Chiusura.
      @$killTimer(@%timer)
      delete $this()
}
```

Vi faccio notare, anche se credo che questo sia ormai chiaro, che tutte le funzioni di cui abbiamo dotato la nostra classe possono essere chiamate anche dall'esterno con "*%NomeOggetto->$funzione(parametri)*" proprio come si fa con gli oggetti delle classi base di KVIrc, vi faccio un esempio, torniamo alla nostra iconlabel e dotiamola di qualche altra funzioncina utile:

```
class(iconlabel,hbox)
{
      constructor()
      {
            %icon=$0
            %text=$1
            @%ico_lb=$new(label,$$)
            @$setAlignment(@%ico_lb,"right")
            @$setStretchFactor(@%ico_lb,10)
            @%ico_lb->$setImage(%icon)

            @%txt_lb=$new(label,$$)
            @$setAlignment(@%txt_lb,"left")
            @$setSpacing(2)
            @$setStretchFactor(@%txt_lb,90)
            @%txt_lb->$setText(%text)

            if($2) @$setTooltip($2)
      }
      setText()
      {
            @%txt_lb->$setText($0)
      }
      setImage()
      {
            @%ico_lb->$setImage($0)
      }
```

```
	getText()
	{
		return @%txt_lb->$text()
	}
}
```

Adesso creiamo l'ggetto

```
%Iconlabel=$new(iconlabel,0,0,218,Nuova Label, Creazioneeee)
%Iconlabel->$show()
```

e poi, proviamo le funzioni che abbiamo aggiunto, una alla volta:

```
%Iconlabel->$setText(Floppy)

%Iconlabel->$setImage(10)

/echo %Iconlabel->$getText()
```

Ovviamente possiamo aggiungerne molte altre ..a seconda di quello che dovesse servirci.
Un'altra cosa da notare è che potremmo richiamare le funzioni degli oggetti interni alla nostra classe attraverso gli stessi oggetti, un esempio:
OggettoCreato->oggettoInterno->funzioneOggettoInterno()
nel nostro caso:

```
%Iconlabel->%txt_lb->$setText(Floppy)
```

ed ancora

```
%Iconlabel->%ico_lb->$setImage(10)
```

senza la necessità di creare le due funzioni interne

```
	setText()
	{
		@%txt_lb->$setText($0)
	}
	setImage()
	{
		@%ico_lb->$setImage($0)
	}
```

quindi arrivare alle funzioni dei due oggetti attraverso questi stessi.
Questo, però, comporterebbe la necessità di doverci riguardare, ogni volta, i nomi degli oggetti interni che abbiamo creato (in questo caso %txt_lb, %ico_lb), ...concorderete che è sicuramente molto più comodo crearci delle funzioni specifiche come abbiamo fatto nel nostro esempio per utilizzarle in modo diretto.

```
%Iconlabel->$setText(Floppy)
%Iconlabel->$setImage(10)
```

è sicuramente più comodo e più facile da ricordare di:

```
%Iconlabel->%txt_lb->$setText(Floppy)
```

```
%Iconlabel->%ico_lb->$setImage(10)
```

Ovviamente se dovessimo avere la necessità di richiamare una funzione particolare e specifica degli oggetti potremmo sempre ricorrere alla formula

```
%Iconlabel->%ico_lb->$funzione()
```

l'importante è creare un richiamo diretto con quelle da noi più utilizzate.
Ribadisco che ciò è comunque possibile fino a quando, nella classe, abbiamo dichiarato le variabili come variabili di classe (**@%txt_lb)** e non semplicemente come variabili locali tempornaee (**%txt_lb).**
Infatti queste ultime si "perderebbero" una volta create e non sarebbero più raggiungibili.
Respiro...
Prima di leggere il codice seguente forse sarebbe meglio leggersi il capitolo 6 di A.O.S., io comunque ho deciso di spiegarlo quì perchè fa capire ancora meglio quanto codice si possa risparmiare utilizzanzo l'ereditarietà delle classi e creandoce di personlizzate.
Guardiamo il seguente codice:

```
class("mybtn","button")
{
	function constructor
	{
		@$setText($0)
		$1->$setGeometry(40,40,100,50)
		objects.connect $$ clicked $1 $2
	}
}

class("mywidget","widget")
{
	function myexit
	{
		delete $$
	}

	function constructor
	{
		@%btn=$new(mybtn,$$,0,"Exit",$$,"myexit")
	}
}
```

Il codice è abbastanza semplice: sono due classi, la prima **mybtn** eredita dalla classe **button** e la seconda **mywidget** eredita dalla classe **widget.**
Considerate *mybtn* come la classe che vi siete creati per risparmiare codice in futuro e la classe mywidget come lo script che state facendo attualmente e nel quale utilizzate la vostra classe personale *mybtn*.
Il costruttore della classe *mybtn* riceve tre parametri:
$0 = che non è altro che il testo che avrà il pulsante, ed infatti guardando nel costruttore della seconda classe lo ritroviamo quì:

```
		@%btn=$new(mybtn,$$,0,"Exit",$$,"myexit")
```

$1 = questo non è altro che il riferimento ad un oggetto, nel caso particolare passo il riferimento alla classe mywidget e lo ritroviamo quì:

```
@%btn=$new(mybtn,$$,0,"Exit",$$,"myexit")
```

$2 = questo è invece il nome di una funzione, nel nostro caso "myexit" appartenente all'oggetto che abbiamo passato attraverso $1:

```
@%btn=$new(mybtn,$$,0,"Exit",$$,"myexit")
```

e che nel nostro caso non fa altro che cancellare l'oggetto di tipo mywidget creato, in pratica cancella se stesso:

```
function myexit
{
	delete $$
}
```

L'altra funzione:

```
$1->$setGeometry(40,40,100,50)
```

l'ho proposta solo per farvi capire come $1 sia un riferimento all'oggetto creato di tipo *mywidget* e, pertanto, tutto ciò che accadrà a $1 si riperquoterà sul nostro oggetto.
Passiamo all'ultima funzione effettivamente da spiegare:

```
objects.connect $$ clicked $1 $2
```

a questo proposito guardiamo la documentazione del KVIrc sotto la voce commands->objects.connect dove possiamo leggere

Connects the <source_object>'s signal <signal_name> to the <target_object>'s slot <slot_name>. When one of the two objects is destroyed, the signal/slot connection is automatically removed.

In pratica non fa altro che collegare un segnale (nel nostro caso clicked) di un oggetto (nel nostro caso $$ cioè la stessa classe) ad una funzione di un altro oggetto (nel nostro caso alla funzione *$myexit()* -$2- dell'oggetto creato utilizzando la classe *mywidget* -$1-).
Piccola digressione: dovete sapere che ogni volta che si verifica un event, l'oggetto emette un segnale relativo all'evento verificatosi (andate a leggere, a questo riguardo, il volume 6 di A.O.S. sugli SLOT e i Segnali), questo segnale se collegato ad una funzione, ogni volta che viene emesso la eseguirà.
In pratica quando clicco su un oggetto di tipo *button* questo chiamerà automaticamente l'evento *mousePressEvent* che emetterà il segnale *clicked* ...tutto in automatico.
Infatti dall'help del KVIrc alla voce relativa alla classe *button*, possiamo leggere

Signals
$clicked()
This signal is emitted by the default implementation of $clickEvent().

Quindi nel nostro caso, tutte le volte che verrà emesso il segnale clicked, grazie alla connessione che abbiamo creato, verrà chiamata la funzione *myexit*.
Facciamo una prova

```
%x=$new(mywidget)
```

```
%x->$show
```
Provate a clickare sul pulsante ...direi che va bene!
Il risultato è che, una volta creata la nostra classe mybtn con una sola riga di codice:
```
            @%btn=$new(mybtn,$$,0,"Exit",$$,"myexit")
```
ne risparmieremo moltissime.

Ancora una piccola nota prima di concludere con questo corposo AOS 4 =)
Volevo farvi notare la comodissima possibilità di assegnare delle variabili agli oggetti grafici creati, vi faccio subito un esempio:

```
%Lview=$new(listview,0)
%Lview->$addColumn("Nomi")

%LwItem[0]=$new(listviewitem,%Lview)
%LwItem[1]=$new(listviewitem,%Lview)
%LwItem[2]=$new(listviewitem,%Lview)
%LwItem[3]=$new(listviewitem,%Lview)

%LwItem[0]->$setText(0,Tonino)
%LwItem[1]->$setText(0,Alessandro)
%LwItem[2]->$setText(0,Giorgio)
%LwItem[3]->$setText(0,Giovanni)

%LwItem[0]->%telefono=33333333
%LwItem[1]->%telefono=44444444
%LwItem[2]->%telefono=55555555
%LwItem[3]->%telefono=66666666

%LwItem[0]->%compleanno=agosto
%LwItem[1]->%compleanno=ottobre
%LwItem[2]->%compleanno=marzo
%LwItem[3]->%compleanno=aprile

%Lview->$show()

echo %LwItem[2]->%compleanno
```

Come potete notare ho creato delle variabili legate agli oggetti che mi contengono informazioni importanti, questo è molto utile perchè mi permette di poter conservare informazioni in un determinato oggetto, generalmente relative allo stesso, che poi posso richiamare in seguito senza che l'utilizzatore finale dello script debba vederle.
In un eventuale rubrica, ad esempio, potrei, ciclando, ricavarmi tutti i nati nel mese di agosto, o tutti quelli che hanno un numero di telefono fisso e così via.
Bene, direi che anche per questa volta abbiamo finito.
Digitiamo con soddisfazione il nostro...
/ECHO STOP

```
- - - - - - -- - - - - - - -- - - - - - - - - -- - - - - -- - - - - -
```
" Tu vedi cose e ne spieghi il perché, io invece immagino cose che non sono mai esistite e mi chiedo perché no." (George Bernad Shaw)
```
- - - - - - - - -- - - - - - - --  -- - - - - - - - - - - - - - - - -
```
Grifisx (Tonino Imbesi)