Creating Desktop Applications in Dabo

Ed Leafe
September 2007

Table of Contents

Introduction 3
Target Audience for These Sessions 3
Creating an Application 3
Dabo Naming and Programming Conventions 6
Database Connectivity 7
Business Objects 9
User Interface 11
    Common UI Properties 11
    Common UI Objects 12
        Forms 12
        Panels 12
        Paged Controls 13
        List Controls 13
        The Grid 14
UI Programming Strategies 17

Introduction

Dabo is a framework written in Python that enables you to easily create powerful desktop applications. These can range from those that interact heavily with databases to those that are purely user interface. Like any powerful tool, Dabo can be intimidating and confusing when you first start using it, so these sessions are designed to guide you through your first simple application and beyond.

Target Audience for These Sessions

I am assuming that you have experience programming, and have at least a basic understanding of Python. Obviously, the more you know Python, the easier it will be to follow along. I'm also assuming that you have Python and Dabo installed on your computer.

I am going to start with examples that use only code; later in the tutorial I will introduce the visual tools and the files they create.

Creating an Application

Let's start by creating the most basic app. We will be using the command line and simple text scripts for now; later on we will use the visual tools to make a lot of this easier. So for each example script, copy it into a plain text file, give it a name like 'sample.py', and run it by entering the command 'python sample.py' at the command line.

This is about as simple a script as you can get:

```
    import dabo
    app = dabo.dApp()
    app.start()
```

When you run this, you get a blank form displayed, along with a basic menu. Close the form, and the application exits.

Now let's replace that boring blank form with something a little more interesting:

```
    import dabo
    dabo.ui.loadUI("wx")

    class HelloWorldForm(dabo.ui.dForm):
    def afterInit(self):
    self.Caption = "Hello World Form"
    self.lblHello = dabo.ui.dLabel(self, Caption="Hello World",
FontSize=24, ForeColor="blue")
```

```
app = dabo.dApp()
app.MainFormClass = HelloWorldForm
app.start()
```

Here we defined a form class that inherits from the base Dabo form class: **dForm**. All the base UI classes are in the **dabo.ui** module. Since Dabo is designed to support multiple UI toolkits, we need to tell Dabo which one we want to use, so we add the command `dabo.ui.loadUI("wx")` to load the wxPython toolkit into the dabo.ui namespace. Right now Dabo only support wxPython, but someday we'd like to integrate Tkinter, Qt, and possibly other UI toolkits.

We customize the form in its `afterInit()` method. This method is automatically called by the framework for every object in order to provide you with a place to do this sort of customization without having to override (and potentially mess up) the native `__init__()` method. There is another useful 'after' method available, named `afterInitAll()`. The difference between the two is that `afterInitAll()` is called after the object and all contained objects have been created. So in a form with several objects, there is a distinct order in which these various things happen and when the corresponding methods are called. In this example, there is a form that contains an editbox, a panel, and a textbox. The panel contains label. Here is the order that the various events happen when this form is created:

1. form creation
2. form.afterInit()
    a. editbox creation
    b. editbox afterInit()
    c. editbox afterInitAll()
    d. panel creation
    e. panel afterInit()
        i. panel label creation
        ii. panel label afterInit()
    f. panel afterInitAll()
    g. textbox creation
    h. textbox afterInit()
    i. textbox afterInitAll()
3. form.afterInitAll()

So the basic structure of an application is to define the MainFormClass for the app, and start the event loop by calling app.start(). You can also subclass the dApp class to add your own custom behaviors, and then reference them from anywhere in your app using the `obj.Application` reference that's available to all objects in Dabo.

Ready for some really cool stuff? Run the Hello World app again, and then from the File menu, select 'Command Window'. You'll see a new window pop up; it will most likely be too small to work with, so resize it to something more reasonable, and postion it so that you can see the main form, too. Dabo will remember the size and position of your various windows when you close them, so that the next time you run the app, the windows appear as you left them.

The command window is an interactive Python interpreter that you can use to investigate and modify your running apps. To make it easy to use, we've mapped the name 'self' behind the scenes to refer to the form that was frontmost when you brought up the command window. You can test this out by typing `self.Caption` and pressing Enter; on the output frame on the bottom you should see 'Hello World' displayed. You will also have noticed the code completion popup as soon as you typed the period after 'self': all the known attributes, methods and properties of the object referred to by 'self' (in this case, the HelloWorld form) are displayed, and as you type successive letters of the desired property ('Caption', in this case), the selected item in the list changes to match what you've typed. Once you have the one you want selected, press Enter to have it entered into the command for you.

You can also change things on the form, too. To see this, press Control-UpArrow once to go to the previous command, and then at the end add: `= "I changed this"`, and press Enter to execute that command. If you look at the form, its caption has now been changed interactively!

Let's not stop here; let's create a control on the fly! Type this command:

```
dabo.ui.dButton(self, Caption="Click Me")
```

You should see the new button appear, but unfortunately it's right over the label. To make matters worse, we didn't save a reference to the button, so how can we move it? Well, there is a handy UI method called `getMouseObject()`, and this will return a reference to whatever Dabo object is below the mouse cursor when the command is exectuted. So place your mouse cursor over the new button without clicking in the Hello World form, and in the command window type:

```
btn = dabo.ui.getMouseObject()
```

Press Enter, and now 'btn' is a reference to the new button. Note to Windows users: some versions of Windows act odd with this command, moving the command window to the bottom of the stack while still displaying it normally. Just click the button for it in the Start bar, and it will return to normal.

Now that we have the reference, we can play with the button. Try entering commands like this:

```
btn.Left = 50
btn.Bottom = 150
```

...and you should see the button moving as the commands execute. Of course, if you click the button, it looks like it's being clicked, but it doesn't do anything. Let's create a simple method on the fly and bind it to the button's Hit event, which is fired when the button is clicked. Type the following code in the command window:

```
def clicker(evt):
import time
self.Caption = time.ctime()
```

```
btn.bindEvent(dabo.dEvents.Hit, clicker)
```

Now when you click the button, the Hit event will be raised, and since it is bound to the clicker() method, that method fires. All event handlers receive an event object parameter; while we don't use it here, we still have to accept it. And since this method was defined in the Command Window, 'self' refers to the form. So each time you click the button, the Caption of the form will change to the current time!

This is still pretty basic stuff, but you can make much more complex forms by adding more and more controls. Getting them to all appear nicely will require that you specify the size and position of each explicitly, or use sizers to do all that for you. Sizers are a whole topic unto themselves, and we'll cover them later. For now I cannot stress enough that although they may seem difficult to grasp at first, the investment in learning how to use sizers will pay off greatly in the long run.

(You introduce sizers here, and make value judgements for using them, without really explaining them. I would try to add some drama:

“Getting them to all appear nicely will require you to specify the size and position of each explicitly, just like in VFP. Or you can use dSizer, which will size and position your controls for you automatically, so you don't need to think about the user's screen resolution or how big the form is. Well cover dSizer later, but needless to say, having positioning and sizing done automatically is much better than specifying absolute coordinates.”

This should open up questions in people's minds, and when they ask them, you can be ready to clarify what sizers bring to the table, the most important thing being of course, cross-platform layout consistency.

Dabo Naming and Programming Conventions

We've tried to be as consistent as possible in creating Dabo, so that developers using Dabo can readily learn the framework and use it effectively. Here are some of the conventions you will come across as you use Dabo:

- The 'd' prefix for classes: the base classes in the Dabo framework are all prefixed with a lower-case letter 'd' (e.g., dTextBox, dBizobj, etc.). We've done this to make the framework classes readily distinguishable from derived classes you may create in your application.
- Class names begin with a capital letter. E.g.: Wizard, ClassDesigner, etc.
- All attributes and methods begin with a lower-case letter. E.g.: dabo.ui.areYouSure(), self.requery(), self.sortOrder
- Property names begin with a capital letter. E.g.: self.Form, biz.KeyField, self.Caption
- Long names are made legible by using inter-capitalization instead of underscores. E.g.: requeryAllChildren(), self.AutoPopulatePK
- Properties and attributes that begin with a leading underscore are considered "for internal use only" by the class that they belong to; in other words, they are implementation details

that should not be accessed from outside of the class in which they are defined.
- No more than one blank line within a method; two blank lines between methods; three blank lines between classes.
- Use tabs for indentation. It Just Makes Sense™.

Another convention is the use of Python properties instead of get/set methods wherever possible. Properties give lots of control over the implementation of various behaviors, and are also discoverable by documentation tools. We've also created the ability to set any properties directly in the constructor of any object. For example, let's say you were creating a simple label, but wanted to customize how it looked. You could write:

```
lbl = dabo.ui.dLabel(self)
lbl.Caption = "This is some text"
lbl.FontSize = 14
lbl.FontFace = "Tahoma"
lbl.RegID = "theTextLabel"
```

Or you could simply pass all those property values to the constructor:

```
lbl = dabo.ui.dLabel(self, Caption="This is some text",
FontSize=14, FontFace="Tahoma", RegID="theTextLabel")
```

Either way will work, but the second version is cleaner and easier to understand. Also note another programming convention: in regular statements, spaces are inserted around the '=' operator, but in parameter lists, there is no space. This is a standard Python style, and we have followed that in Dabo.

Database Connectivity

Dabo works with many database backends; currently we support the following:

- MySQL
- PostgreSQL
- Microsoft SQL Server
- Firebird
- SQLite

There are two main classes used for connections: dConnectInfo, and dConnection. Typically you create a dConnectInfo object, populate it with the required information, and then call its getConnection() method, which returns an instance of dConn ection. You then pass that dConnection object to create a business object, and work with that bizobj, or you can call dConnection.getDaboCursor() to do direct interaction with your database.

Here's a sample session using cursors directly:

```
import dabo
ci = dabo.db.dConnectInfo(Host="dabodev.com", DbType="MySQL",
Database="webtest", User="webuser",
PlainTextPassword="foxrocks")
conn = dabo.db.dConnection(ci)
# Get a cursor to work with
crs = conn.getDaboCursor()
# Execute a query
crs.execute("select * from zipcodes where czip like '1452%' ")
print crs.RowCount
# prints '7'
# Print out all the cities and their zipcodes
crs.first()
while True:
try:
rec = crs.Record
print crs.RowNumber, rec.ccity, rec.czip
crs.next()
except dabo.dException.EndOfFileException:
# we're done
break
```

Note that we first define the connection info, and then create the connection. This may seem like an unnecessary step, but the dConnectInfo object encapsulates the differences between various backend adapters way of specifying connection parameters. Also, there is a 'PlainTextPassword' field; by default, Dabo assumes that anything passed in a 'Password' parameter is encrypted, and runs the default decrypting on it. So we use 'PlainTextPassword' so that Dabo knows to use the string as-is.

We then iterate through the cursor's rows using the cursor navigation methods first() and next(). (There are also last() and prior() methods, if you were wondering). You can also set the RowNumber property to directly move the current record pointer. The cursor (and bizobj, as you'll see shortly) have a Record object that reflects the data in the current row. You then refer to each column in the result as simple attributes of the Record object.

Business Objects

Business objects, or 'bizobjs', as they are commonly called, are the glue that binds together the UI elements and the database backend in a Dabo application. This is what is known as a 'three-tier' design, which consists of UI, bizobjs and data. In this design, the UI can 'talk' to the bizobj tier, but would never work with the data tier directly. This ensures that your business rules regarding data access and integrity are always enforced.

There are two main things you need to configure in a bizobj: its interface to the data, including SQL Builder or direct SQL statements, and validation routines for the data. For the data interface, there are two main properties that need to be set: DataSource, which is the name of the underlying table in the database, and KeyField, which is the primary key used to update changes

back to the database (If you are using compound keys as your PK, you can specify the KeyField as a tuple of field names). You then need to specify what sort of data you want to get from the database. You can do this by writing the SQL directly, which is a good option for those who are comfortable with SQL, or you can use the SQL Builder methods to specify the parts of the SQL you want to customize. By default, the bizobj will run:

select * from <self.DataSource> limit 1000

The SQL Builder methods are:

- **addField(exp, alias=None)** – Add the field (and optional alias) to the fields in the select statement.
- **setFieldClause(clause)** – Sets the field clause, erasing anything previously specified.
- **getFieldClause()** – Returns the current field clause.
- **addFrom(exp)** – Adds the specified table to the 'from' clause
- **setFromClause(clause)** – Sets the from clause, erasing anything previously specified.
- **getFromClause()** – Returns the current from clause.
- **addJoin(tbl, exp, joinType=None)** – Adds a 'join' to the statement. You specify the table to join, and the expression to use for the join. 'joinType' should be one of 'inner', 'outer', 'left', 'right'. Default is 'inner'
- **setJoinClause(clause)** – Sets the join clause, erasing anything previously specified.
- **getJoinClause()** – Returns the current join clause.
- **addGroupBy(exp)** – Adds the expression to the 'group by' clause
- **setGroupByClause(clause)** – Sets the group by clause, erasing anything previously specified.
- **getGroupByClause()** – Returns the current group by clause.
- **setLimit(val)** – Sets the limit value on the number of returned records. Default=1000. Set this to None to return all records.
- **getLimit()** – Returns the current limit clause.
- **addOrderBy(exp)** – Adds the expression to the 'order by' clause
- **setOrderByClause(clause)** – Sets the order by clause, erasing anything previously specified.
- **getOrderByClause()** – Returns the current order by clause.
- **addWhere(exp, comp="and")** – Adds the expression to the 'where' clause. If there is already something in the where clause, the new expression is added with a default of 'and', but you can also specify 'or' if that is what you need.
- **setWhereClause(clause)** – Sets the where clause, erasing anything previously specified.
- **getWhereClause()** – Returns the current where clause.
- **getSQL()** – This will take all the 'pieces' specified by the various SQL Builder methods and construct the actual SQL statement to be used.

So let's continue with our simple app, adding a bizobj into the mix.

```
import dabo
ci = dabo.db.dConnectInfo(Host="dabodev.com", DbType="MySQL",
Database="webtest",
User="webuser", PlainTextPassword="foxrocks")
conn = dabo.db.dConnection(ci)
# Create the bizobj
biz = dabo.biz.dBizobj(conn)
# These are the two required properties to be set
biz.DataSource = "zipcodes"
biz.KeyField = "iid"
# Add some fields
biz.addField("iid")
biz.addField("ccity")
# Add an alias for this field
biz.addField("czip", "postalcode")
# Add a WHERE clause
biz.addWhere("czip like '1452%' ")
# Run the query
biz.requery()
print biz.RowCount
# prints '7'
# Show the SQL that was run
print "SQL:", biz.LastSQL
# Iterate through the records, and print the city/zip
for rownum in biz.bizIterator():
rec = biz.Record
print rownum, rec.ccity, rec.postalcode
```

Bizobjs also have first(), next(), etc., methods for navigation, but it's handy to have a way to process all the rows in the current result set. That's what the bizIterator class does; as we iterate through the records in the bizobj, it moves the pointer as if we called next(), and when we reach the end of the results set, the iteration ends.

User Interface

Dabo has a rich set of UI controls and a robust event model. Dabo is designed to be UI toolkit-agnostic, meaning that we plan on supporting any and all UI toolkits, such as Tkinter, Qt, and wxPython. We do this by wrapping the toolkit classes so that they use a common naming and API, so no matter what the toolkit calls, say, a control for entering a line of text, you will use a dTextBox; and no matter how the toolkit implements getting the contents of that control, you would use dTextBox.Value. This common API simplifies UI programming, and makes for more consistent coding.

Realistically, wrapping a UI toolkit is a very intensive undertaking, so for the foreseeable future, we will only be supporting the wxPython toolkit. We chose this as our primary toolkit because it uses each platform's native drawing routines to create controls that don't just 'look' native; they *are* native.

Common UI Properties

- **Caption**: many controls have a small bit of text displayed on them. Buttons, labels, tabbed pages, etc. All of these can be set using the Caption property.
- **Value**: while some controls are display only, others, such as text controls, check boxes, list boxes, sliders, etc., all have a value associated with them. The Value can be of any data type.
- **DataSource**, **DataField**: these are the basis of data binding. When these are set, the control gets its Value from the specified DataSource/Field, and changes to the control are propagated back to the DataSource/Field. Typically in a database application, the DataSource is a bizobj, and the DataField is the name of the column in the data set. But a DataSource can be any object, and the DataField can be a property of that object, so that changes to one control can immediately affect another control. The possibilities are endless!
- **Width**, **Height**, **Size**: these control the actual size of the control when sizers are not in use. Width and Height are integer values, while Size is a 2-tuple of (Width, Height). When sizers are in use, these control the minimum size that the sizer will be allowed to size the control at runtime.
- **Left**, **Right**, **Top**, **Bottom**, **Position** – These control the placing of the control in its parent container. The first 4 are integers, while Position is a 2-tuple of (Left, Top). When using sizers, setting these is pointless, because the sizer will control the positioning of the object; essentially, they are read-only with sizers.
- **Parent**: a reference to the object that contains the control.
- **Form**: the top-level window that contains the control. For a form itself, the Form property returns None.
- **BackColor**, **ForeColor**: the background and foreground color used by the control. These can be set with RGB tuples, such as (255,128,0), or with common color names such as 'blue', 'RED', 'lightGreen'. Note that capitalization is ignored for colors.
- **Children**: returns a list of all objects contained by the control.
- **Enabled**: determines if a control can be interacted with by the user.
- **Visible**: determines whether the control is shown or not.
- **FontBold**, **FontFace**, **FontItalic**, **FontSize**, **FontUnderline**: used to set aspects of the displayed font on the control.
- **Name**: uniquely identifies a control among other controls with the same parent container. All objects have a non-blank Name.
- **RegID**: optional. If set, uniquely identifies an object within a given form, and a reference can be obtained by treating the RegID as an attribute of the form. For example, if a particular control has a RegID="theThing", any other control on the form can refer to it as self.Form.theThing, no matter how deeply buried it is in any containership hierarchy.
- **ToolTipText**: holds the text of the tool tip displayed when the user hovers the mouse over the control.

There are many other properties that can be used, as well as some that are specific to certain types of controls, but these will cover 80% of what you need.

Common UI Objects

Forms

All UI elements exist in a top-level window on the screen; in Dabo, we call these windows 'forms'. Every UI control must be created with some sort of 'container' control as its Parent; the first parameter of every UI object creation statement is that required Parent object. Forms are the only UI object that can have None as a parent, although you can create forms (and dialogs) as children of other forms, so that when the parent form is destroyed, the child form is also destroyed.

Panels

Panels are convenient container objects for grouping related controls. Normally they have no visible appearance, but you can set their BackColor, or give them a border, etc., if you want them to be visible. You can nest panels within each other to any degree you like.
Note to Windows users: in order for forms to look 'normal' for the given theme, you must add a panel to the form, and then add all your controls as children of this main panel. Otherwise, the background appears as a dark grey, and your form will not look right.
A great use of panels is to save encapsulated designs as classes, so that everywhere you need to get standard contact information from the user, you just instantiate and place your PnlContactInfo subclass of dPanel.

Paged Controls

Dabo offers several 'paged' controls; these are containers that have one or more pages, with only one of the pages visible at any given time. The main difference is how the active page is set:

- **dPageFrame**: each page has a 'tab' with identifying text. Clicking the tab brings that tab's page forward. The position of the tabs is specified at creation with the TabPosition property (default=Top).
- **dPageList**: presents the captions for the various pages as a list. Selecting one of the captions from the list selectes the associated page.
- **dPageSelect**: similar to dPageList, but the various page captions are displayed as a dropdown list control. Selecting the caption from the dropdown brings the associated page forward.
- **dPageFrameNoTabs**: this is a paged control with no user interface for changing the active page. Page selection must be done programmatically, usually in response to some user-generated event, such as clicking a “next” button that you've defined.

List Controls

Dabo offers several controls for displaying a list of items; they differ in how the items are displayed, and how the user selects an item or items.

- **dListBox**: this control displays several items at once, depending on the height of the control. If there are more items than can be displayed at once, scrollbars allow the user to see the hidden items. If MultipleSelect is True, the user can select more than one item at a time by shift-clicking, control-clicking, etc.
- **dDropdownList**: this control displays a single item at a time, but clicking on the control causes the list of items to be displayed. Once the user selects an item from that list, the list closes and a single item (the selected one) is displayed.
- **dComboBox**: similar to dDropdownList, except that the user can type in the area where the single item is displayed. Normally you would have to write code to handle what is to be done to the typed value, but if the AppendOnEnter property is True, the text will be added to the list of items in the control if it isn't already present.
- **dListControl**: this is a cross between dListBox and dGrid. Like a listbox, you can only interact with rows, but like a grid it can have multiple columns in each row.

With list controls, you specify the list of items by setting the **Choices** property to a list of the items to display. You can also set an optional **Keys** property; this is a 1:1 list of key values, each of which is associated with the Choices value in the same list position.

You can specify the **Value** of the selected item in several ways. First is the text that is displayed; this is called **StringValue**. Second is the position of the selected item within the list; this is **PositionValue**. Finally, if keys have been set for the control, you can get the key associated with the selection by referencing the **KeyValue** property. There is another property named **ValueMode** that can be one of 'String', 'Position' or 'Key'; how this is set determines which type of value is returned when you reference the Value property.

The Grid

Fully describing the grid control could take an entire chapter – maybe even several chapters. But this will serve as a basic introduction into the **dGrid** class and how to use it effectively.

Grids are used to display data records (called a '**dataset**') in a tabular format, similar to a spreadsheet. If you simply have a dataset named 'ds' and want it displayed, you can call `dGrid.buildFromDataSet(ds)`, and a grid will be constructed for you, with a column for each column in the dataset. However, if you want control over the appearance of each column of data, create a **dColumn** instance for each column in the dataset, and set the dColumn's properties accordingly. We will discuss these shortly.

Grids have lots of cool behaviors built into them, such as sortable columns, in-place editing, auto-sizing column widths, and incremental searching within a column. You can modify or turn off any of these if you wish, either on a column-by-column basis or grid-wide, but in general they make grids very handy for your users.

Generally, you assign the **DataSource** property of the grid to control where the grid gets its data, and assign the **DataField** property to each column to control which column in the dataset is displayed in that grid column.

Here are the grid properties most commonly-used to control the grid's appearance:

- **AlternateRowColoring**, **RowColorEven**, **RowColorOdd**: when AlternateRowColoring is True, it colors alternate rows of the grid according to the two row color properties. This makes it easier to visually separate the rows of the grid.
- **Children, Columns**: these are aliases that both return a list of the dColumn instances that comprise the grid.
- **CurrentRow**, **CurrentColumn**: these determine the location of the selected cell of the grid.
- **HeaderBackColor**, **HeaderForeColor**: these set the default back and forecolor for the column headers. Individual columns can override with their own value.
- **NoneDisplay**: this is the text displayed when null values (None) are present.
- **ResizableColumns**, **ResizableRows**, **SameSizeRows**: the first two determine if the user can change the width of columns or the height of rows by dragging the separator lines. The last one controls whether changing the height of one row changes all the rows to that height.
- **SelectionBackColor**, **SelectionForeColor**: These control the color of any selected cells in the grid.
- **ShowColumnLabels**, **ShowRowLabels**: these determine if the column headers or row labels are displayed.

These are the properties used most commonly to control the grid's behavior:

- **ActivateEditorOnSelect**: when the grid is editable, should the cell go into editing mode as soon as it is selected? Or should it wait to be clicked after being selected in order to display its editor? This property determines the behavior.
- **Editable**, **Searchable**, **Sortable**: these determine, respectively, if anything in the grid can be edited, searched or sorted. If they are False, then that behavior is disabled for the entire grid. However, setting it to True doesn't do anything by iteself; each grid column for which you want that particular behavior must also have its corresponding property set to True before the behavior can occur.
- **Encoding**: if the data in the grid has a different text encoding, this can override the application's setting.
- **HorizontalScrolling**, **VerticalScrolling**: these control if the grid can be scrolled in the respective directions.
- **MovableColumns**: columns can be re-arranged by dragging their headers. Setting this property to False disables this ability.
- **MultipleSelection**: when True, the user can select more than one cell/row/column at a time.
- **SearchDelay**: when running an incremental search, this controls how long a pause must be before the next keypress is considered a new search instead of appending the key to the current search string.
- **SelectionMode**: can be one of either Cells, Row or Column. This determines if clicking on a cell selects just that cell, that cell's row, or that cell's column.

Columns have their own properties; here are the most commonly-used:

- **Caption**: the text displayed in the Header.
- **Order**: this is an integer that controls the order of the column within the grid. You can give your columns numbers like 1,2,34,... or 10,20,30... or even 1,66,732. The grid will display the columns depending on the relative value of their Order property.
- **Editable**, **Searchable**, **Sortable**: See the grid properties of the same name above. When these are enabled for the grid, enabling them for the column allows the behavior to happen.
- **Visible**: this can be toggled to show/hide a column.
- **Width**: this determines the Width of the column.
- **BackColor**, **ForeColor**: these control the color of the cells in the column.
- **FontBold**, **FontFace**, **FontInfo**, **FontItalic**, **FontSize**, **FontUnderline**: these control the appearance of the font for the cells in the column.
- **HeaderBackColor**, **HeaderFontBold**, **HeaderFontFace**, **HeaderFontItalic**, **HeaderFontSize**, **HeaderFontUnderline**, **HeaderForeColor**: like the above, except these control the column header's appearance.
- **HorizontalAlignment**, **VerticalAlignment**, **HeaderHorizontalAlignment**, **HeaderVerticalAlignment**: these determine where in the cell or header, respectively, the text appears. The former can be one of 'Left', 'Center' or 'Right', while the latter can be one of 'Top', 'Middle' or 'Bottom'. Default is Center, Middle.
- **Expand**: when set to True, the column expands/contracts proportionally as the grid is resized.
- **Precision**: determines how many decimal places are displayed when float values are in the column.
- **WordWrap**: when True, long cell contents will be wrapped onto additional lines as long as their is enough height in the row to display those lines.
- **CustomEditorClass**, **CustomEditors**, **CustomRendererClass**, **CustomRenderers**: you can define custom controls to handle how data is rendered and/or edited in a grid. The CustomEditorClass and CustomRendererClass are used when you need data in a column to be treated differently than the default. You can get even finer-grained control by creating a dict with the row number as the key and a custom editor/renderer as the value, and setting the CustomEditors/ CustomRenderers to this dict. Now any row with an entry in this dict will get its own editor/renderer; the rest get the standard CustomEditorClass/ CustomRendererClass.

Common UI Events

GUI programming is all about receiving and responding to events, and Dabo tries to make it as easy as possible to do this effectively. When an event is raised, nothing will happen unless there is an event handler that is bound to the event. An 'event handler' is a method that accepts an event object, and carries out the desired behavior in response to the event.
There are several ways to bind a handler to an event. Let's start with the direct method: the bindEvent() method:

```
    someWindow.bindEvent(dabo.dEvents.Activate, self.activateHandler)
```

```
...
def activateHandler(self, evt):
self.doSomething()
etc....
```

So now when 'someWindow' is activated, the 'activateHandler' method fires, and does whatever you need to happen. Note that 'activateHandler' accepts an event parameter; that is required for all handlers, whether you need the information in the event object or not.
Another way to bind to an event is to create a handler method named '**on*EventName***' in your class; this will automatically bind the event '*EventName*' to that method. This is usually a big convenience, but if for some reason this would interfere with your program, you can turn off this behavior by setting **`dabo.autoBindEvents`** to `False`.
A third way is to pass the handler to the constructor for the object using the keyword '**On*EventName***'. Here's a common example: creating a button and specifying the handler for when it is clicked:

```
btn = dabo.ui.dButton(self, Caption="Click Me", OnHit=self.btnHandler)
```

The event object that is passed to the handler contains a lot of relevant data about the event in its EventData dictionary. Mouse events will contain information about the location of the event; Key events will contain information about the key and any modifier keys being pressed; tree events will contain information about the particular node involved in the event.
All the main event classes are defined in the dabo.dEvents module; you can define your own events by subclassing the dabo.dEvents.dEvent class. There are over 100 events defined in dEvents; here are the ones you will use most often:

- Hit: Nearly every control has an obviously way that the user interacts with it: a button is clicked, a selection is made from a list, a menu item is selected, a timer fires, a chcekbox is toggled, a slider is dragged, etc. Each of these actions by the user raises a Hit event, so in your coding you don't need to worry about what each control calls its main event; you know that Hit will be the one you want to bind to.
- GotFocus, LostFocus: these are raised when a control gets focus and loses focus, respectively.
- KeyDown, KeyUp, KeyChar: these events allow you to respond to key presses. Important EventData keys are: keyChar, keyCode, unicodeChar, unicodeKey, controlDown, shiftDown, commandDown, altDown, metaDown.
- MouseLeftDown, MouseLeftUp, MouseLeftClick, MouseLeftDoubleClick: note that there are versions of these methods for the right and middle mouse buttons; these are named the same, but with 'Right' and 'Middle' replacing 'Left' in the event name. Important EventData keys are: mousePosition, dragging, controlDown, shiftDown, commandDown, altDown, metaDown.
- ContextMenu: this is raised by either a right click or when the context menu button is pressed. The same EventData keys for the mouse events apply here.
- Activate, Deactivate, Close: these form events are raised when the form becomes frontmost, loses frontmost status, and is closed, respectively.
- Update: an Update event is raised by the framework when data needs to be refreshed.
- Resize: raised when an object is resized.
- Destroy: raised when an object is released.

- Idle: raised continuously when nothing else is happening.

UI Programming Strategies

An application with a graphical user interface differs from non-GUI apps in that there is a main event loop that waits for the user to do things, and then processes those events as they happen. Occasionally, though, you may find that the code you write doesn't seem to do what you were expecting: for example, you may write code that fires when a control's value changes, but which depends on that new value being 'visible' by other objects in the system. Due to the order in which events get processed, though, your code may fire before the control's value has been updated, and you get stale values. This is not an error on your part; it's just a common pitfall of event-driven programming.

To work around this, use the `dabo.ui.callAfter(mthd, *args, **kwargs)` function to add your method call (and any parameters) to the end of the event queue. This will ensure that any previous processing has already been completed before your method is executed. There is an analogous function for delayed setting of properties: `dabo.ui.setAfter(obj, propName, value)`.

Another common problem in a GUI environment is excessive duplicate calls to the same function, such as refresh(). There may be several objects that call refresh() when they are modified somehow, and the refresh() code triggers changes in other objects that end up calling refresh() again, and so what happens is that refresh() can be called dozens of times when once would have sufficed. To solve that problem, use `dabo.ui.callAfterInterval(interval, mthd, *args, **kwargs)`.; there is an analogous function `dabo.ui.setAfterInterval(interval, obj, propName, *args, **kwargs)` for setting properties. Here 'interval' is the number of milliseconds to wait for duplicate calls; the other arguments are just like those in dabo.ui.callAfter(). The way it works is that when this call is received, Dabo check an internal log to see if it already has a call pending with the same signature. If not, it creates a timer that will fire in the specified interval, and adds that call to the log. Now lets assume that another call for the same method with the same args is received. Dabo sees that it is a duplicate, so it resets the timer and discards the duplicate requests. If several dozen similar calls are received, they are likewise discarded. Once the duplicates stop long enough for the timer to fire, the method is called and removed from the internal log.