

## Flash Tools for Developers: Function Grapher A Guide

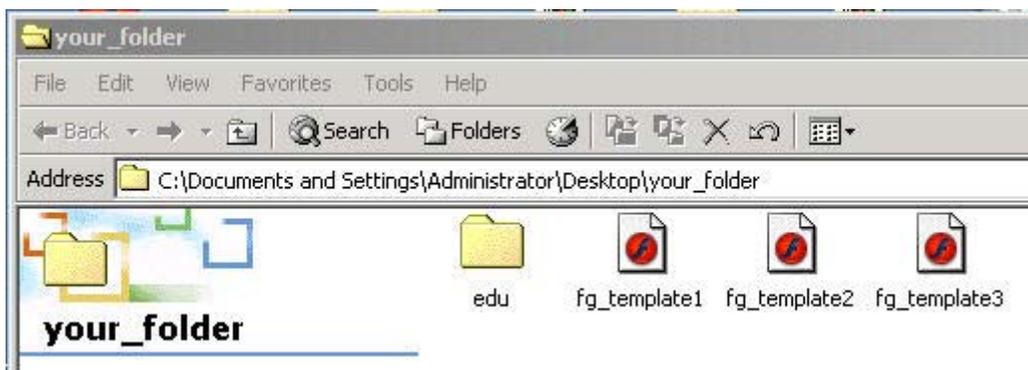
*This paper is a companion to the online article at the MathDL Digital Classroom Resources "Flash Tools for Developers: Function Grapher" by Barbara Kaskosz and Douglas E. Ensley. This paper provides instructions on how to use the three function grapher templates and the ActionScript classes presented in the article. The templates and the classes discussed below can be downloaded from the article through the link [fun\\_grapher.zip](#).*

### Getting Started

Download **fun\_grapher.zip** file and unzip it in a folder on your computer. You will see a **fun\_grapher** folder which contains all the files related to the article. You will be particularly interested in the following:

- The folder **edu** contains all the necessary *ActionScript* classes in a nested sequence of folders.
- The three source files for the templates that will be discussed are **fg\_template1 fla**, **fg\_template2 fla**, and **fg\_template3 fla**.

Working right from the folder **fun\_grapher**, you can open one of the three template **.fla** files in *Flash MX 2004 Professional* or *Flash 8 Professional* and begin customizing the template to your liking. Alternatively, if you prefer a clean working environment, you can copy the **edu** folder and the templates into a new folder. **Note:** For *Flash* to be able to find the classes, the folder **edu** must reside in the same folder as the template you are working on.



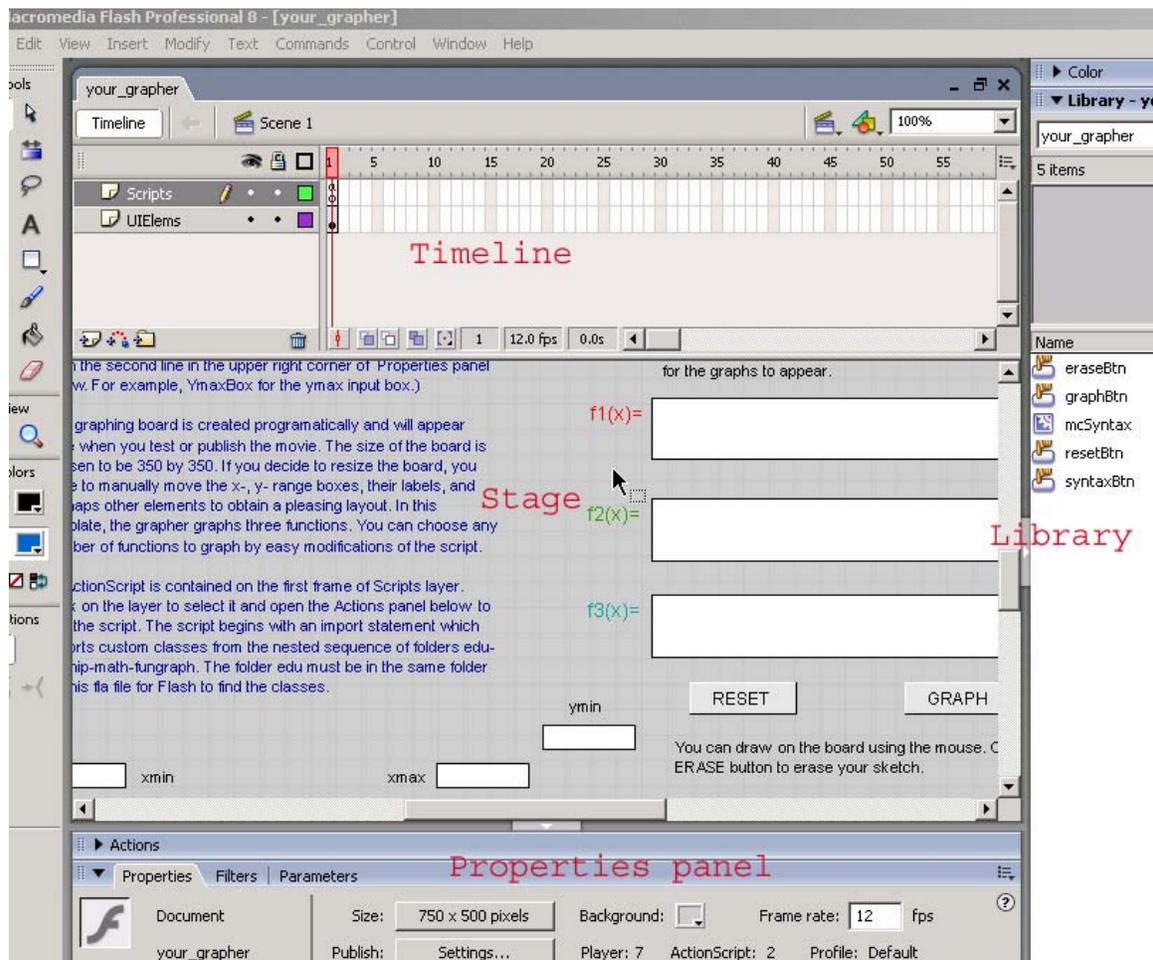
Open your version of *Flash*, navigate to, and open one of the templates. For the purposes of this demonstration we will work first with **fg\_template1 fla**. (The screen shots below show *Flash 8* but everything looks very similar in *Flash MX 2004*.) After you open the template, you may want to save it under a new name as your working file, for example, **your\_grapher fla**. To do that you go to **File**→**Save As**, click on **Save As**, type the new name in the dialog box, and click **Save**. If you are in *Flash 8*, a dialog box will appear asking you if you want to convert the file (which was originally created in *Flash MX*

2004) to *Flash 8* format. If you plan to continue working in *Flash 8*, click Save. If you are in *Flash 2004*, no dialog box will appear.

We now have in front of us the file **your\_grapher.fla** open. Let us examine the elements of the file.

## Elements of the Grapher's fla file

The main elements you see are the **Timeline**, the **Stage**, which is not visible in its entirety because of the movie's size, and the **Properties** panel. The **Actions** panel tag is visible but the panel is closed. A portion of the open **Library** shows on the picture below. If the **Library** is not open, go to the **Window** item in the uppermost menu and click **Library** in the dropdown menu. The **Library** contains assets for our movie. (A few buttons and a movie clip, nothing to be concerned about at this moment.) All the code for the Grapher can be found in the **Actions** panel. We will open it and examine it later. First, let's look at other elements.



The **Timeline** has only one frame and two layers. The layer named **Scripts** is a holder for all *ActionScript* code in the movie; the layer **UIElements** holds all elements you see on the

Stage. (**Note:** Layers in *Flash* serve mostly as a convenience at the authoring time; they help you organize the elements of your application. In most cases, layers do not matter much for your final compiled file.) If you click on the **UIElements** layer, all elements on the **Stage** become selected which is indicated by blue outline boxes around them. Deselect them by clicking on any empty spot on the **Stage**.

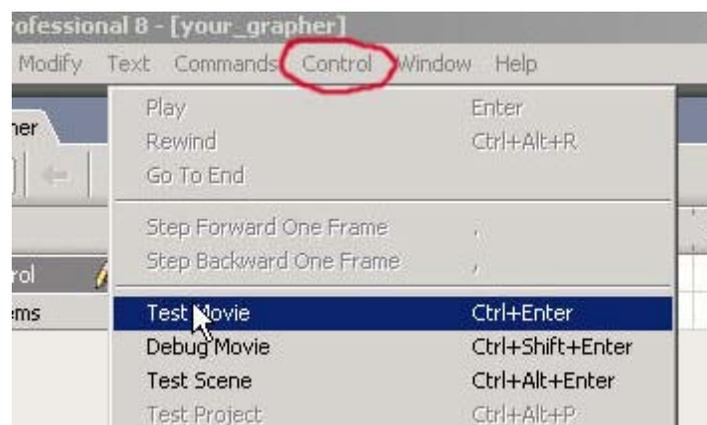
Throughout this paper we are assuming that the **Selection Tool** (the left arrow in the top row of the **Tools** panel) is on unless otherwise specified:



The **Stage** contains all elements of the Grapher which were created manually: input boxes, buttons, and static text boxes. You can see all of them by sliding vertical and horizontal sliders on the right and on the bottom of the **Stage**.

Below the **Stage** you see the **Actions** panel tag and the **Properties** panel which is open. If no elements on the **Stage** are selected, the **Properties** panel shows properties of our movie: its size, background color, publish settings, and other properties. If you select an object on the **Stage** (by clicking on it), the **Properties** panel will display important information about the selected object.

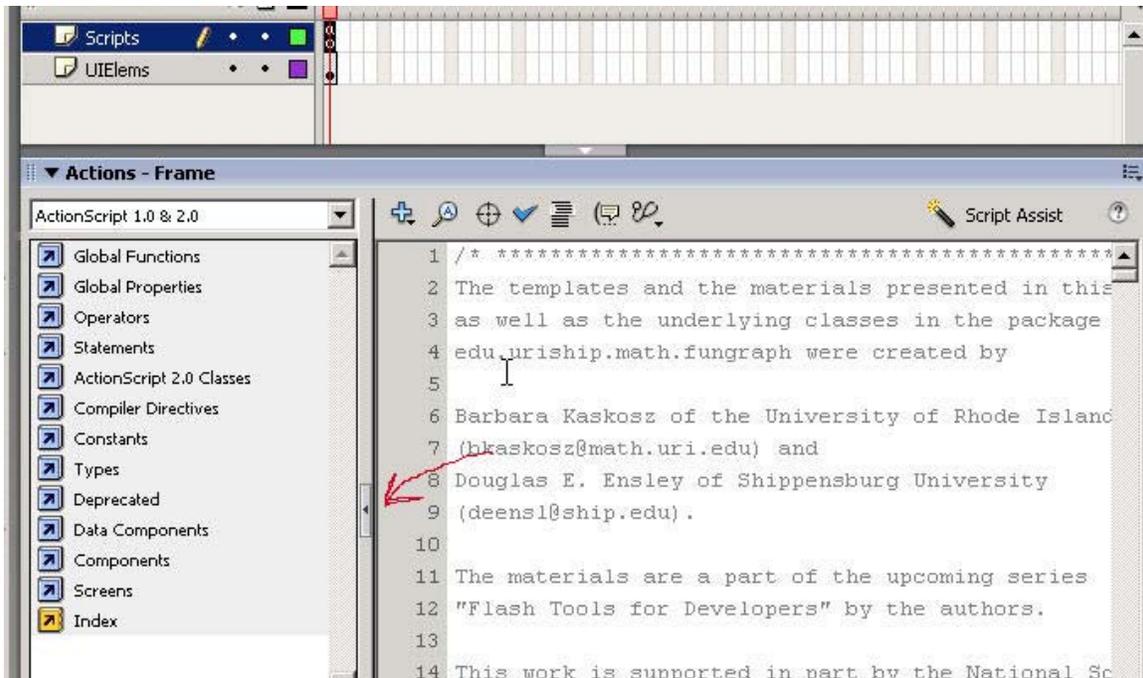
Before you do anything else, you may want to see how the compiled movie works. To do this, click on the **Control** item located at the uppermost menu. In the dropdown menu that opens, click on **Test Movie**:



A new window opens with a compiled **your\_grapher.swf** file playing in the *Flash Player*. Familiarize yourself with the Grapher's functionality and then close the *Player's* window. (**Note:** In *Flash 8*, typically, the *Player* window floats on top of your **.fla** file; in

*Flash 2004*, it covers the **.fla** file. In either case, close it by clicking on the **X** at the top right corner of the Grapher's window.) You are back now to the authoring environment.

All the code needed for the Grapher to run (besides the code in the classes contained in the folder **edu** which will be imported) is attached to **Frame 1** of the **Scripts** layer of the **Timeline**. (Most **fla** files contain a layer whose only purpose is to hold the code. It is usually named "Control" or "Actions" or "Scripts".) To access the code, select the **Scripts** layer by clicking on it, then open the **Actions** panel by clicking on its name or on the little arrow next to it:

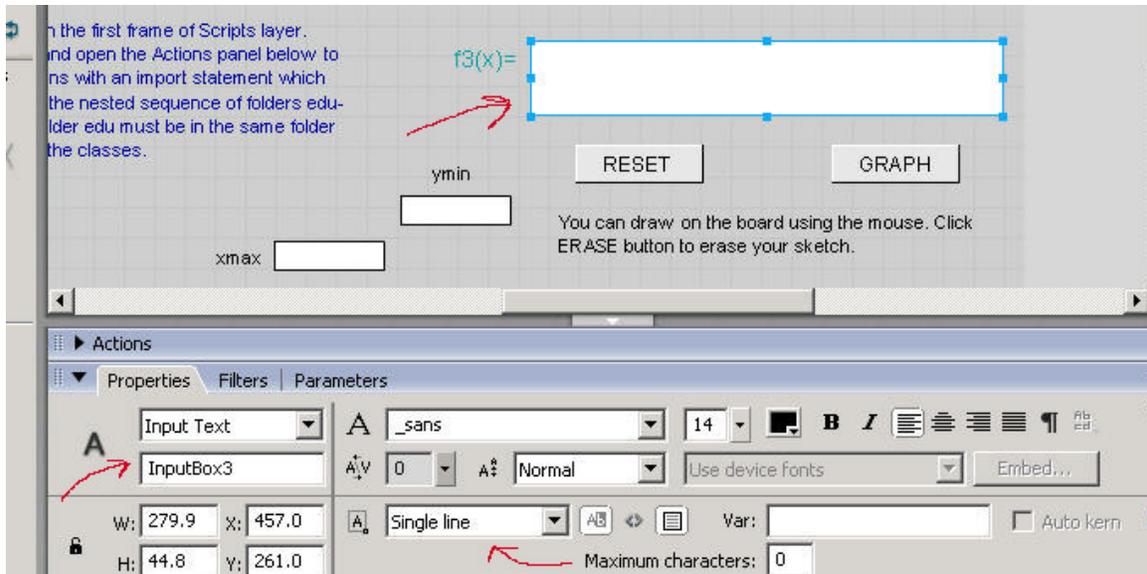


Note: in order to see the code, you must select the **Scripts** layer first as the code is placed on **Frame 1** of this layer.

Unless you want to use *ActionScript* help as you modify the code, click on the little arrow marked on the picture to close the left panel and have a bigger window within which to work with code. Now that we know where all the elements are, let's learn how to work with them. Close the **Action** panel by clicking on its name and let's get back to elements on the **Stage**.

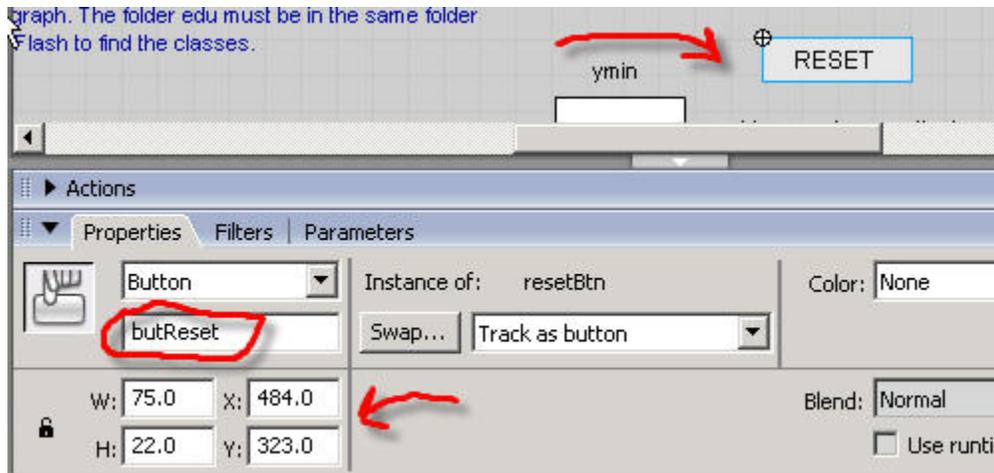
### Elements on Stage: Properties and Instance Names

The elements on the **Stage** were created manually and consist of input text fields, static text fields, and buttons. Input text fields and buttons are controlled by *ActionScript* so they must have instance names. Click, for example, on the box next to the label "f3 (x) =". The blue outline appears indicating that the text field is selected and its description shows in **Properties** panel. (Make sure the panel is open.)



The description tells us that the text field is of the type **Input Text** in which the user will enter text. Just below the type **Input Text**, we see the instance name, **InputBox3**, which was given to the field. When you place a new input or dynamic text field on the **Stage** you must assign an instance name to it by typing the name in the instance name box as shown on the picture. (The same holds true for buttons and movie clips placed on the **Stage**.) For input text fields it is also important to select the **Single line** option marked above to prevent the user from entering hard breaks in a compiled movie. The font should not be the same as the font of static text boxes. Since our static text on the **Stage** is in **Arial**, we chose the **generic \_sans** font for all input boxes. The size of the font is chosen to be 14, the color black, the alignment left. The clicked button to the left of "Var" gives our box background and border. The default colors are white and black, respectively. We will change the color of the border for **InputBox3** programmatically.

*Failure to assign instance names to objects on the **Stage** is a very common reason for your code not to work.* Below is a screen shot with **RESET** button selected. The **Properties** panel tells us that **RESET** is a button, an instance of **resetBtn**. The name **resetBtn** was given to the button when the button was first created, and the name which the button's prototype has in the **Library**, but this is not the name that matters to us at this point. (**Note: We will talk more about instances and Library assets in the [Creating New Objects on Stage](#) section.**) The name by which the button will be accessed by *ActionScript* is its instance name, **butReset**, typed in the box encircled in red. You could create more instances of **resetBtn** by dragging it from the library window to the **Stage**, give each instance its own unique name and control each of them separately through *ActionScript*.



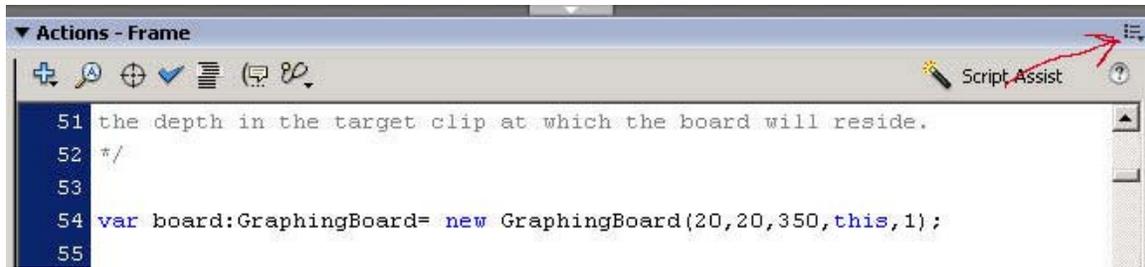
You can move elements on the **Stage** by selecting them and then pressing arrow keys; you can also drag them with the mouse. But you can achieve the most precise positioning by assigning x- and y- coordinates to an element by typing the coordinates in the **X** and **Y** boxes marked on the picture above. Our **RESET** button's position is x=484.0, y=323.0. You can move it by typing new numbers into the boxes.

**Note:** In *Flash*, the x-coordinate is the horizontal coordinate, in pixels, measured from the upper left corner of the movie (whose x-coordinate is 0) and increasing to the right. The y-coordinate is the vertical coordinate, in pixels, measured from the upper left corner of the movie (whose y-coordinate is 0) and increasing when you go down.

We will learn later how to create new buttons, text fields, and other user interface elements. For now, please read the explanations in the static text box on the **Stage** in blue font (you see the very bottom of it in the upper left corner of the picture above), then select the box (note that static boxes do not have instance names) and delete the box from the **Stage** by pressing the delete key.

### Modifying the Script: Changing Appearance

In this section, we will show how to customize the appearance and the properties of the Grapher through simple modifications of the script. Select the **Scripts** layer and open the **Actions** panel. Most of the text you see are comments. Comments are enclosed between `/*...*/` or following `//`, and they appear in faded grey. The comments are design to walk you step-by-step through the script. For the purpose of our little exercise, scroll to line 54. (**Note:** If line numbers are not showing click on the little menu icon marked by the arrow on the picture below and check [Line Numbers](#).)

A screenshot of an IDE's Actions panel. The panel title is "Actions - Frame". It contains a toolbar with icons for adding, deleting, undo, redo, and search. Below the toolbar is a text area with code. Line 54 is highlighted in blue. A red arrow points to a "Script Assist" button in the top right corner of the panel.

```
51 the depth in the target clip at which the board will reside.  
52 */  
53  
54 var board:GraphingBoard= new GraphingBoard(20,20,350,this,1);  
55
```

The line 54 that you see is instantiating the custom class, **GraphingBoard**, contained in the package **edu.uriship.math.fungraph** (the package was imported on line 40) and storing the instance in the variable "board". The instance "board" will create a square graphing board where all your graphs will appear. The instance "board" will also be responsible for creating graphs, for formatting and displaying the error box, the coordinates display box, and for performing other tasks. The constructor of the class, evoked by the word "new", takes several parameters. The first three are responsible for the location and the size of the board within the target movie clip. The target movie clip above is the main movie, referred to as "this", and "board" is placed at depth 1 which is specified by the last parameter.

For the purposes of this demonstration, we begin our customization by changing colors and properties of some of the elements controlled by "board". Suppose we want a white graphing board with a black border instead of the dramatic black board with white border. It takes only a few simple changes to the code.

Scroll to line 180. You see there:

```
board.changeBorderColor(0xFFFFFFFF);
```

Change it to:

```
board.changeBorderColor(0x000000);
```

(As you see, the method takes as a parameter a hex number of a color.)

Scroll to line 187. You see there:

```
board.changeBackColor(0x000000);
```

Change it to:

```
board.changeBackColor(0xFFFFFFFF);
```

Go to **Control** → **Test Movie**. As you see, the board is white, with black border. The axes appear awfully light, though. If you mouse over the board, the coordinate display box still has a black background. If you enter a formula for one of the functions with a syntax error and click GRAPH, the error box that appears has a black background. (To see this, try entering the function "x +" and clicking GRAPH.) We need to change all that. Close the compiled movie and go back to the **fla** file.

Scroll to line 194. You see there:

```
board.setErrorBoxFormat(0x000000,0x000000,0xCCCCCC,12);
```

Change it to:

```
board.setErrorBoxFormat (0xFFFFFFFF,0xFFFFFFFF,0x000000,12);
```

The method takes parameters that control your error box's background color, border color, text color, and text size.

Scroll to line 212. You see there:

```
board.setCoordsBoxFormat(0x000000,0x000000,0xCCCCCC,12);
```

Change it to:

```
board.setCoordsBoxFormat (0xFFFFFFFF,0xFFFFFFFF,0x000000,12);
```

The method takes parameters that control your coordinate display box's background color, border color, text color, and text size.

Test the movie now. Mouse over the board and try to graph "x+". The error box and coordinate display box appear in proper colors. Try to draw on the board with the mouse. Your drawing is in light yellow. We need to change that, as well as axes color. Go back to the **fla** file and scroll to line 222:

```
board.enableUserDraw(0xFFFF00,1);
```

The line instructs "**board**" to enable the user to draw in yellow with the line thickness 1. Change it to whatever color and thickness you want, for example, dark gold:

```
board.enableUserDraw(0xFF6600,1);
```

Scroll to line 229:

```
board.setAxesColor(0xCCCCCC);
```

If you want axes to be black, change it to:

```
board.setAxesColor(0x000000);
```

Test the movie now. Graph three functions, say "x", "sin(x)", "tan(x)". The red of the first graph seems fine but the green and the cyan of the other two graphs are too light.

The instance "**board**" also controls the color of graphs it produces. To change them go back to the **fla** file and scroll to line 496:

```
board.drawGraph(2,f2Array,0x00FF00);
```

The method is evoked within the function `makeGraphs` which has already parsed the user's input for `f2(x)` and produced an array of points `f2Array` to be graphed. The first parameter of the method is the number of the graph (it also is the depth of the graph within a movie clip internal to "board"). The last parameter determines the color of the graph. Let's darken it a bit. Change the line to:

```
board.drawGraph( 2 , f2Array , 0x009900 ) ;
```

Scroll to line 523 and change:

```
board.drawGraph( 3 , f3Array , 0x00FFFF ) ;  
to  
board.drawGraph( 3 , f3Array , 0x0000FF ) ;
```

which will produce a blue graph (or enter any other color you want).

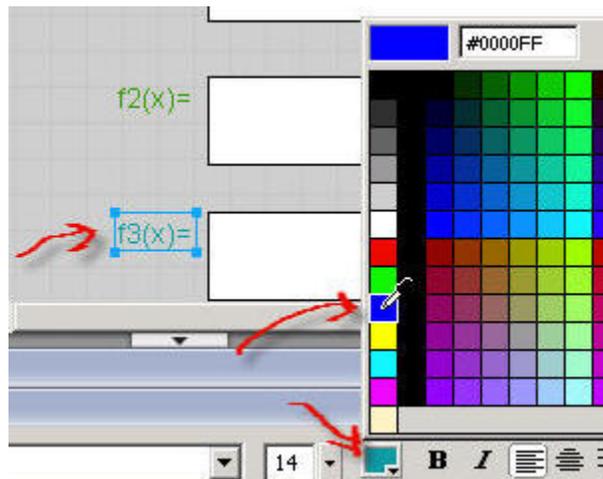
Test the movie, graph a few functions. All the colors appear pleasing, except for the colors of the two last input boxes' borders and their labels which do not correspond any more to colors of the graphs. The colors of the borders for those boxes are set programmatically. Go to line 139 and change it to:

```
InputBox2.borderColor=0x009900 ;
```

Go to line 143 and change it to:

```
InputBox3.borderColor=0x0000FF ;
```

Labels "f2" and "f3" are static text boxes. They cannot be controlled programmatically; we have to adjust their colors manually on the **Stage**. Close the **Actions** panel by clicking on its name. Make sure that the **Properties** panel is open. On the **Stage** click on "f3=" label. The blue outline box around the static text box and its description in the **Properties** panel appear. In the **Properties** panel, click on the small colored rectangle to the right of the font size. A color palette shows. Click on the blue rectangle:



After you click, the color of the font within the selected text field changes to blue. Similarly, you can change the color of "f2=" label by clicking on the rectangle #009900 in the palette although the color of the label as is seems fine. Test the movie again. You have successfully changed the coloring of the Grapher.

### Modifying the Script: Resizing and Repositioning

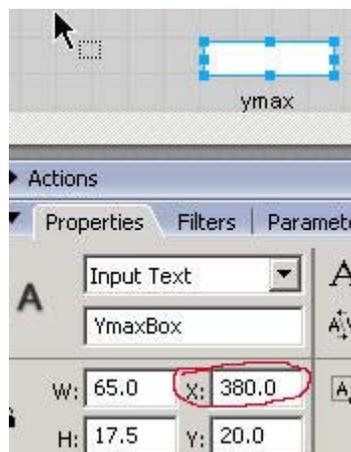
If you want to change the layout of the Grapher, the first thing you may want to change is the size of the graphing board. This is done very easily. Go to line 54 again where "board" is instantiated:

```
var board:GraphingBoard = new GraphingBoard(20,20,350,this,1);
```

The first two numbers, 20, 20, give the x- and the y-position, in pixels, of the upper left corner of the graphing board with respect to the main movie. "350" is the size of the board. Change 350 to 250:

```
var board:GraphingBoard = new GraphingBoard(20,20,250,this,1);
```

Go to Control → Test Movie. As you see, the board is much smaller now but it has the same functionality as before. (If you forgot to delete the static box in blue font, it will stick out now from underneath the board. Delete it after you go back to the **fla** file.) After you resized the board, you may want to change the location of the range input boxes and their labels. You can do it manually in the **fla** file. Close the *Flash Player* and go back to the **fla** file. Select the **y**max input box right above the **y**max label. You have decreased the size of the board by 100 pixels, so you want to move the box 100 pixels to the left. You can drag the box with the mouse, but you will achieve more precision (particularly not seeing the graphing board at the authoring time) by changing the box's coordinates in the **Properties** panel:



Change the x-coordinate marked on the picture above to 280.0. (The change takes effect when you click anywhere outside the **X** box.) Select the static text box **y**max and change its x-coordinate in the **Properties** panel from 397.0 to 297.0. Test the movie now. The **y**max input box and its label are in the right position. Reposition similarly other range

boxes and their labels. (For `ymin` and `xmax` boxes and their labels, you have to decrease both coordinates by 100. For `xmin`, only the y-coordinate.) Test the movie. Mouse over the graphing board. The coordinate display box is now outside the board.

Although the coordinate display box and the error box are both a part of "`board`" and are controlled by "`board`", they do not have to be positioned within the square where you want the graphs to appear. You can place them anywhere you want. Open the **Actions** panel and scroll to line 214:

```
board.setCoordsBoxSizeAndPos(60,40,20,300);
```

The parameters control, respectively: width, height, x-coordinate, and y-coordinate of the box. The x- and the y-coordinates are relative to the graphing board; that is, relative to the upper left corner of the graphing board, which, in this example, resides at the point (20,20) of the main movie (see line 54). Let's position the box below the graphing board in approximately in the middle between `xmin` and `xmax` labels. Change line 214 to:

```
board.setCoordsBoxSizeAndPos(60,40,100,290);
```

(The exact middle corresponds to the third parameter 105 but it does not produce a pleasing appearance.) Since the box is outside of the board, it will be nicer to give it a black border. Change line 212 to:

```
board.setCoordsBoxFormat(0xFFFFFFFF,0x000000,0x000000,12);
```

Test the movie. The coordinate box looks fine but when you enter formula with a syntax error the error box looks unsightly and sticks outside the board. Similarly as the coordinate box, the error box is a part of and is controlled by "`board`". Again, however, we can position it within or outside the graphing board. This time let's leave it on top of the graphing board. We will only adjust the error box's dimensions. Scroll to line 202 and change it to:

```
board.setErrorBoxSizeAndPos(210,160,20,20);
```

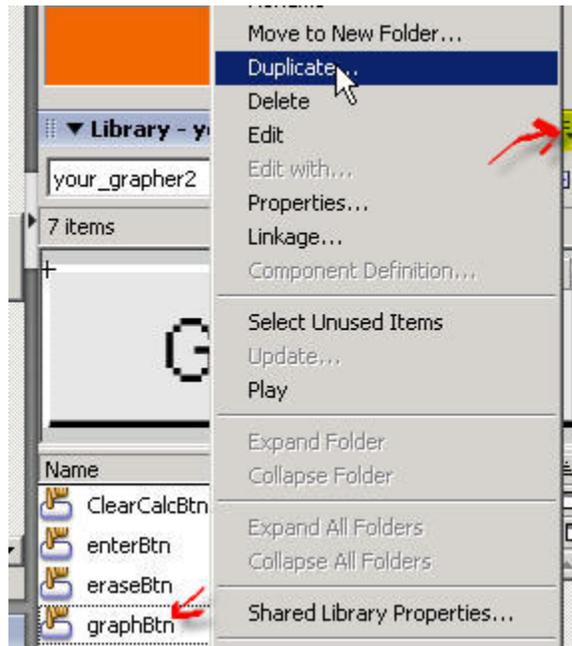
We changed the width and the height of the box and left its position at (20,20). The position is relative to "`board`" so the position of the box in the main movie is (40,40).

### **Modifying the Script: Changing Functionality**

To change functionality of the graphers, we have to delve a little deeper into the code. In this section we will work with **fg\_template3 fla**. Close the previous file and open **fg\_template3 fla**. Again, you may want to save it under your own working name. We want to change the Grapher as follows. The user enters only one function. The function is graphed when the GRAPH button is clicked. Then, when a button `df/dx` (which we will create) is clicked the graph of the derivative of the function is displayed.

Let's begin from creating the derivative button. Instead of creating a button from scratch, we will create a duplicate of an existing button in the **Library** and edit it appropriately.

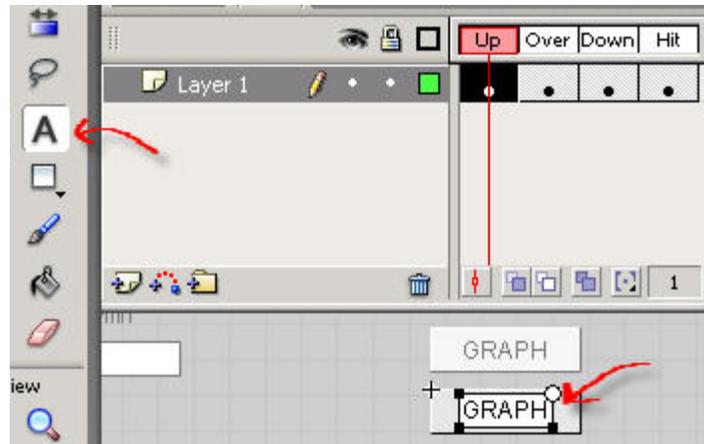
Make sure that the **Library** panel is open. (If it is not, go to **Window** in the uppermost menu and click **Library** in the dropdown menu.) In the **Library**, select any of the existing button symbols, say, **graphBtn**. After selecting the symbol, click the little menu icon in the upper right corner of the Library panel. (The icon is highlighted in yellow on the picture below and a red arrow is pointing at it.) After you click the icon a menu opens. Click **Duplicate**....



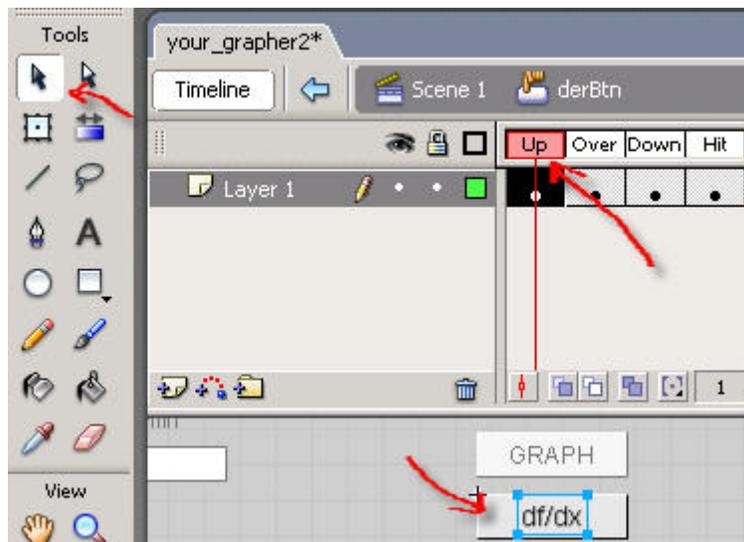
Then a dialog box opens:



Click the **Backspace** key to clear the highlighted field and type the new name **derBtn**. Click **OK**. The window closes and a new button symbol "**derBtn**" appears in the **Library**. Click on it to select it. Its image (which looks the same as the **GRAPH** button) appears in the **Library** window. Select the **FunUI** layer on the **Timeline** and drag the image of the button from the **Library** window onto the **Stage** and place it, say, under the **GRAPH** button. The button is selected. In the **Properties** panel it says that the button is an instance of **derBtn**; the instance name field is blank. Type **butDer** in the instance name field. Now we need to edit **butDer** so it has the right label. Right-click on the button and in the menu that opens click **Edit in Place**. The button's timeline opens in which you can edit the button.



Click the text tool marked by the arrow on the picture above and click right after "H" in GRAPH. The cursor is blinking, you can now edit the text in the text field. Backspace to clear the existing text and enter "df/dx". Click on the **Select** tool in the **Tools** panel (the one we have used all along). The text field is selected. Center it using the arrow buttons, perhaps increase the font size to 14 using the **Properties** panel. When you are satisfied, go to the **Edit** menu item in the uppermost menu and click **Copy** in the dropdown menu.



Move the playhead which is now over the frame **Up** to **Over**. Select the text field containing GRAPH. Delete it by pressing the **Delete** key. Go to the **Edit** → **Paste in Place** and click it. A new text appears. Move the playhead to the next frame **Down** and then to **Hit** and repeat the process. After you are finished, drag the playhead over the four frames to see if your button is animated properly. If yes, go to **Edit** → **Edit Document**. You are back on the main **Stage**. You have a button labeled **df/dx** whose instance name is **butDer**. This is the name that we will use to assign functionality to the button via *ActionScript*.

Select and delete the text field **InputBox2** next to the label "**f2(x)=**". Select and delete the label itself. Choose the text tool in the **Toolbox** marked "**A**" (the one we just used above), click on the "**f1(x)=**" label and change it to "**f(x)=**". Select the **df/dx** button, press the

Shift key and select the GRAPH button. With both of them selected, use the upper arrow key to move them up a few notches for a nicer appearance.

Now we have to modify the script. We begin by deleting the parts of the script which have to do with `InputBox2` (we have just deleted the box) and processing of `f2(x)` and its graph. After we start deleting portions of the script, the line numbers become approximate as they depend on how many empty lines you deleted in the process.

Select the **Scripts** layer and open the **Actions** panel. Scroll to line 128:

```
InputBox2.text="";
```

and delete it. Then scroll to and delete the lines 144 and 146:

```
InputBox2.wordWrap=true;
InputBox2.borderColor=0x00CC00;
```

Inside the function `makeGraphs`, delete the lines 271 and 279:

```
var sFunction2:String="";
var f2Array=[];
```

Do the same with the line 294:

```
compObj2:CompiledObject;
```

and with the line 393:

```
sFunction2=InputBox2.text;
```

Go to line approximately 462 and delete the whole block of code to the line approximately 486:

```
if(sFunction2.length>0){
    .....
    board.drawGraph(2,f2Array,0x00FF00);
}
```

Test the movie. The Grapher should work fine except that it can graph only one function. The `df/dx` button is clickable but does not do anything.

We will augment the script to create the graph of the derivative of the function entered by the user. Again, line numbers at this point are only approximate. Scroll back to the top of the script and under the line 112: `var points:Number=330;` add the line:

```
var derArray:Array=[];
```

The `derArray` variable will store the points to be plotted in order to create a graph of the derivative. It has to be a global variable as it will be used by two different functions. Scroll down all the way to the function `makeGraphs` and on the line approximately 325, right after the line: `var oRange:RangeObject;` add the line:

```
derArray=[];
```

Then on the line approximately 460 right after:

```
board.drawGraph(1,f1Array,0xFF0000);  
    }
```

after the `"}"` shown above but before the final `"}"` closing the function `makeGraphs` add the following block of code:

```
for(i=0;i<points;i++){  
    derArray[i]=[f1Array[i][0],(f1Array[i+1][1]-f1Array[i][1])/xstep];  
    }  
derArray[points]=derArray[points-1];
```

We have created the variable of values to plot the derivative. Now at any point, for example, after the last line of the code add:

```
butDer.onRelease=function(){  
    board.drawGraph(2,derArray,0x00FF00);  
    }
```

At last scroll to line approximately 585 inside the function `butReset.onRelease` and right below `board.eraseGraphs();` add the line:

```
derArray=[];
```

Test the movie now. Enter a function into the `f(x)` box and click the GRAPH button. The graph appears. Now click the `df/dx` button the graph of the derivative appears.

Of course, to provide the user with an opportunity to graph derivatives it would be nice to enable user's draw by adding the line:

```
board.enableUserDraw(0xFFFF00,1);
```

somewhere around line 190, for example, above the line `board.changeBorderColor(0xFFFFFFFF);`. Now the user can draw on the board. To erase the user's sketches, add the line

```
board.eraseUserDraw();
```

somewhere inside the function `butReset.onRelease=function(){...}` (line 590 approximately). Or if you wish, create a dedicated ERASE button by duplicating one of the existing buttons and add the task of erasing the user's sketches to that button.

## The Description of Custom Classes

The package `edu.uriship.math.fungraph` contains five custom classes: `CompiledObject`, `MathParser`, `RangeObject`, `RangeParser`, `GraphingBoard`. Here is the description of the classes.

- `CompiledObject`

This simple class defines a datatype that is returned by `MathParser`. The constructor takes no parameters.

```
var compObj:CompiledObject = new CompiledObject();
```

Every instance of `CompiledObject` has three public properties:

`compObj.PolishArray` -- an array. When `compObj` is returned by `MathParser's doCompile` method, the property represents a parsed mathematical expression in the Polish notation. Default value `[]`.

`compObj.errorMessage` -- a string. When `compObj` is returned by `MathParser's doCompile` method, the property represents a specific syntax error message. Default value `""`.

`compObj.errorStatus` -- a number 0 or 1. When `compObj` is returned by `doCompile`, 0 corresponds to no error found, 1 to error found. Default value 0.

Within the three templates, we only use the instances of the class that are returned by `MathParser's doCompile` method.

- `MathParser`

This class is the engine behind parsing the user's input. The constructor takes an array of strings as a parameter. For example:

```
var procFun:MathParser = new MathParser(["x"]);
```

The array of strings represents names of variables that will be recognized by the instance of `MathParser`. In the example above as well as in our templates, we use only one variable "x". There can be any number of variables, e.g.: `new MathParser(["x", "y"])`, and they can have names longer than one letter. If you do not want your instance of the parser to allow variables (for example for the calculator application in **Template 3**), enter the empty array into the constructor: `new MathParser([])`. Constants "pi" and "e" are automatically recognized and evaluated by the parser; do not enter them into the constructor.

Every instance of `MathParser` has two public methods:

`procFun.doCompile(string)` -- this method takes a string (typically a string entered by the user) and returns an instance of `CompiledObject`. To give an example, we repeat below some of the code from **Template 1**. The code resides within the `makeGraphs` function.

```
var sFunction1:String="";  
  
var compObj1:CompiledObject;  
  
sFunction1=InputBox1.text;  
  
compObj1=procFun.doCompile(sFunction1);
```

If `compObj1.errorStatus=1`, we know that the user has made a syntax error and we can send `compObj1.errorMessage` to our error box for display. If `compObj1.errorStatus=0`, there is no error and we can send

```
compObj1.PolishArray
```

to the evaluator method, `doEval`, of `MathParser`. The method is discussed next.

`procFun.doEval(array, array)` -- this method takes two arrays as parameters. For the method to do what you want it to do, the first array has to represent a mathematical expression in the Polish notation returned by the `doCompile` method, the second provides values of the variables recognized by the parser listed in the same order as the order in which the variables were passed to the `MathParser` constructor. In our templates, there is only one variable, "x", so the second array contains only one value, for example, `xmin+xstep*i`:

```
procFun.doEval(compObj1.PolishArray,[xmin+xstep*i]);
```

(The values `xmin`, `xstep`, and `i` are numerical variables which were defined earlier in the script, so `xmin+xstep*i` is a number representing a consecutive value for "x".)

The complete list of functions that `MathParser` recognizes as well as all the syntax rules are described in a movie clip that appears in each of the templates when the user mouses over the SYNTAX button.

- `RangeObject`

This simple class defines a datatype that is returned by `RangeParser's parseRange` method. It is very similar as `CompiledObject`. The constructor takes no parameters.

```
var rangeObj:RangeObject = new RangeObject();
```

Every instance of `RangeObject` has three public properties:

`rangeObj.Values` -- an array. When `rangeObj` is returned by `RangeParser's parseRange` method, the property represents the four numerical values for `xmin`, `xmax`, `ymin`, `ymax`. Default value `[]`.

`rangeObj.errorMes` -- a string. When `rangeObj` is returned by `parseRange` method, the property represents a specific syntax error message. Default value `""`.

`rangeObj.errorStatus` -- a number 0 or 1. When `rangeObj` is returned by `RangeParser`, 0 corresponds to no error found, 1 to error found. Default value 0.

Within the three templates, we only use the instances of the class that are returned by `RangeParser's parseRange` method.

- `RangeParser`

The `RangeParser` is a simple utility for parsing the user's input in the range boxes. We need `RangeParser` to allow inputs containing "pi" like `pi/2`, `3*pi/2` `2*pi` etc., in addition to numerical inputs. The constructor does not take any parameters:

```
var procRange:RangeParser = new RangeParser();
```

Each instance has only one method, `parseRange`:

```
procRange.parseRange(string,string,string,string).
```

The method takes four strings as parameters and returns an instance of `RangeObject`. Here is an example how we use it in our templates. The code appears within the `makeGraphs` function.

```
var oRange:RangeObject;  
sXmin=XminBox.text;  
sXmax=XmaxBox.text;  
sYmin=YminBox.text;  
sYmax=YmaxBox.text;  
oRange=procRange.parseRange(sXmin,sXmax,sYmin,sYmax);
```

If `oRange.errorStatus=1`, we send `oRange.errorMes` to our error box for display. If `oRange.errorStatus=0`, we use the `oRange.Values` array to set our range values:

```
xmin=oRange.Values[0];  
xmax=oRange.Values[1];  
ymin=oRange.Values[2];  
ymax=oRange.Values[3];
```

- **GraphingBoard**

This class is responsible for creating all visual elements of our Graphers except for those that were created at the authoring time. Its constructor:

```
var board:GraphingBoard = new GraphingBoard(20,20,350,this,1);
```

and its methods are used and commented exhaustively in the templates' scripts as well as in the tutorial sections above. Particularly, those methods used in **Template 1** and **Template 3**. Below we list them with examples of possible parameters (if they take parameters):

```
board.changeBorderColor(0xFFFFFFFF)
board.changeBackColor(0x000000)
board.enableErrorBox()
board.disableErrorBox()
board.setErrorBoxFormat(0x000000,0x000000,0xCCCCCC,12)
board.setErrorBoxSizeAndPos(300,150,20,20)
board.enableCoordsDisplay()
board.disableCoordsDisplay()
board.setCoordsBoxFormat(0x000000,0x000000,0xCCCCCC,12)
board.setCoordsBoxSizeAndPos(60,40,20,300)
board.enableUserDraw(0xFFFF00,1)
board.disableUserDraw()
board.setAxesColor(0xCCCCCC)
board.setVarsRanges(-10,10,-10,10)
board.drawAxes()
board.drawGraph(1,f1Array,0xFF0000)
board.eraseUserDraw()
board.eraseGraphs()
```

The latter method erases functions' graphs as well as deletes range settings. It does not erase the user's sketches. It should be noted that the method `board.setVarsRanges` should be called before `board.drawAxes` or `board.drawGraph` are called. We do so in all templates, for example, within the `makeGraphs` function:

```
board.setVarsRanges(xmin,xmax,ymin,ymax)
```

after `xmin`, `xmax`, `ymin`, `ymax` are already defined.

In **Template 2**, which employs trace functionality, there is a number of instance methods used that are worth mentioning even though comments in the script should illuminate their functionality. Every instance of **GraphingBoard** has a cursor in the shape of a hairline cross built in. The default size is 12 by 12 pixels, the default color black, the default line thickness 0. By default, the cursor is not visible. You can change the default parameters and manipulate the cross's visibility and position to use it for tracing purposes. The corresponding methods (their usage is illustrated in **Template 2**) are as follows:

```
board.setCrossColor(0x009900)
board.setCrossSizeAndThick(6,0)
board.crossVisible(true)(or board.crossVisible(false))
board.setCrossPos(50,70)
```

(The position in the latter method is in pixels and it is relative to the graphing board.) The last two methods are most instrumental for tracing graphs of functions. We animate the cross using `this.onEnterFrame` event handler so from the outside of the "board" instance, but we set its position and visibility using the "board" methods. To move the cursor along the graph of a given function and to display the corresponding values of the function at the same time we need both an array of points on the graph in functional terms and the corresponding array translated into pixel values. What comes in handy is the fact that, in addition to drawing graphs, the method `board.drawGraph(1, f1FunArray, 0xFF0000)` takes a functional array (`f1FunArray` in this case) as a parameter and returns the corresponding array in pixels:

```
f1PixArray=board.drawGraph(1, f1FunArray, 0xFF0000);
```

This way, we have both arrays `f1FunArray` and `f1PixArray` necessary for tracing and displaying values.

The remaining methods used in Template 2 are:

```
board.getBoardSize()  
board.getCrossSize()
```

which return the size of the graphing board and the size of the cross cursor, respectively. We also use the method:

```
board.isDrawable(a)
```

A given entry "a" usually represents one of the coordinates in pixels of a point to be drawn or the cursor to be moved to. The method checks if "a" is a finite number and if it does not exceed the pixel value beyond which the undesirable effect of "wrapping around" may occur. (This value is set at 5000 pixels).

The only instance method of `GraphingBoard` class never mentioned or used is

```
board.destroy();
```

The method removes listeners, deletes movie clips created by board, etc. You should evoke the method before deleting the variable "board" (should you need to delete it.)

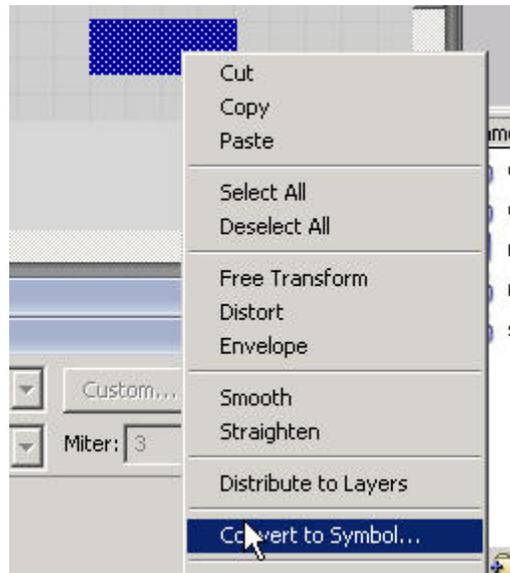
## Creating New Object on Stage

The detailed description of how to create buttons, movie clips, and text fields from scratch and how to edit them is a bit beyond this little guide. We refer the reader to one of the books listed in the next section. The purpose of this section is to show that the process is really easy. For example, let's see how little it takes to create a button.

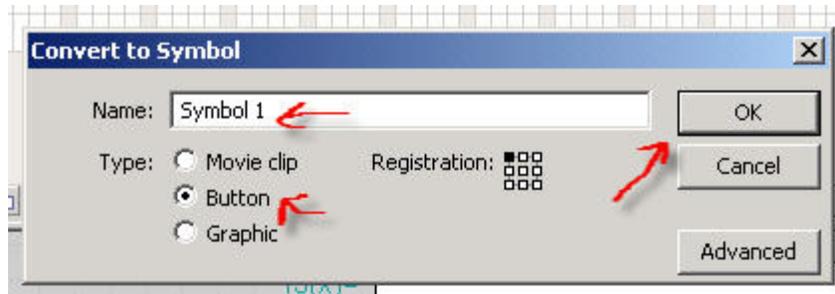
Open **Template 1**. From the **Tools** menu choose the **Rectangle** tool marked on the picture below:



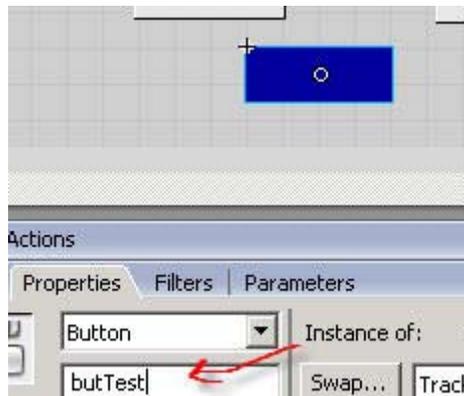
Click on the stroke rectangle (the little rectangle next to the pencil icon) and select the empty stroke. Click on the fill rectangle (the rectangle next to the bucket icon) and choose a color you like. Go to an empty spot on the **Stage** where you want to place your button and by pressing and then dragging the mouse draw a rectangle. Go back to the **Selection tool** (the uppermost right arrow in the **Tools** menu) and click on the rectangle that you have drawn to select it. Then right-click on it. A menu opens:



Click on Convert to Symbol. A dialog box opens:



Make sure that the **Button** option is selected and in the field where you see **Symbol 1**, backspace to erase **Symbol 1** and type **myBtn** instead. Click **OK**. The dialog box disappears. If your new button is not selected, select it. (Observe that the type of selection outline has changed.) In the **Instance Name** box type **butTest** (or any other name you wish):



You have created a new button. Observe that there is a new item in the **Library**: **myBtn**. If you test the movie the new button is clickable except it does not do anything. Open the **Actions** panel and add the following code anywhere you like, say at the end:

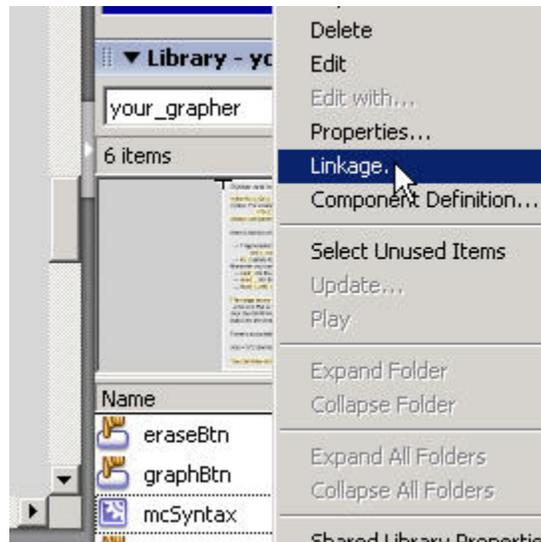
```
butTest.onRelease = function(){
    InputBox1._visible=!InputBox1._visible;
}
```

Test the movie now: when you click the button, **InputBox1** appears and disappears.

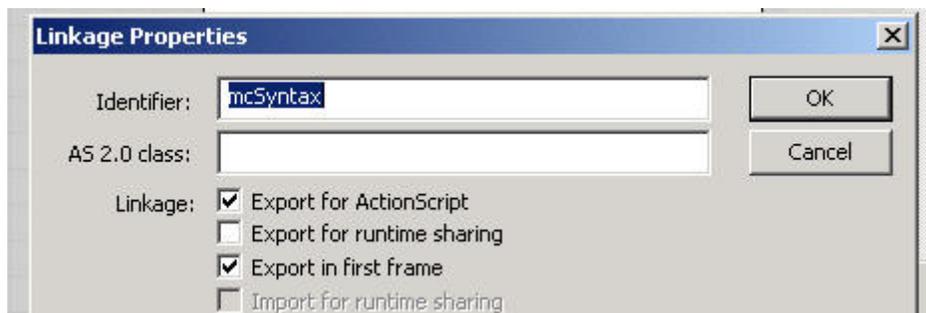
Creating text fields is even easier: select the **Text tool** from the **Tools panel**. (The one marked by "A".) In the **Properties** panel choose **Input Text**. Click at the point on the **Stage** where you want to put your text box. You see a rectangle whose size you can adjust by dragging a tiny square at the bottom of it. To add a background to your field, click a little icon in the **Properties** panel to the left of "Var".

A careful reader may notice that in all our templates there is a movie clip appearing and disappearing when the user mouses over the SYNTAX button. Yet, there is no instance of the movie clip on the **Stage**. There is a way of attaching movie clips at runtime

provided you establish a linkage to their symbol that resides in the **Library**. Select **mcSyntax** in the **Library** and click on the tiny menu icon in the upper right corner of the Library window. A menu opens:



Click on **Linkage**. What you see in the dialog box that opens is that the clip has been imported to *ActionScript* under the name **mcSyntax**:



## Recommended Reading

The complete *Flash 8* documentation, including *Getting Started with Flash*, *Using Flash*, *Learning ActionScript 2.0 in Flash*, *ActionScript 2.0 Language Reference*, *Using Components in Flash*, and more can be downloaded for free from the Macromedia site:

<http://www.macromedia.com/support/documentation/en/flash/>

(After you open the page, click on the link Download Complete *Flash 8* Documentation.)

A few books that we have found immensely helpful are listed below.

To become familiar with the *Flash's* authoring environment:

1. Katherine Ulrich, *Macromedia Flash for Windows and Macintosh: Visual QuickStart Guide*, Peachpit Press, 2004. Flash 8 version is due in December 2005.

2. Robert Reinhardt and Snow Dowd, *Macromedia Flash MX 2004 Bible*, Wiley Publishing, 2004. Flash 8 version is due in February 2006.

To learn *ActionScript 2.0* (which hasn't changed much between *Flash MX 2004* and *Flash 8*):

3. Colin Moock, *Essential ActionScript 2.0*, O'Reilly, 2004.

4. Robert Reinhardt and Joey Lott, *Flash MX 2004 ActionScript Bible*, Wiley Publishing 2004.

5. Colin Moock, *ActionScript for Flash MX: The Definitive Guide*, Second Edition, O'Reilly, 2003.

The latter book was written for an earlier version, *Flash MX*, but we still find it to be a great reference.