

## Classes in the package `flashandmath.as3.*`

**Note:** All methods and properties of the classes listed below are instance methods and properties. None of the classes has static methods or properties.

### `flashandmath.as3.parsers.CompiledObject`

#### Description

`CompiledObject` is a helper class for `MathParser`. It creates a convenient datatype to be returned by `doCompile` method of `MathParser`. This datatype is an object with three properties listed below which comprise the results of compiling a mathematical formula into a form suitable for evaluation.

#### Constructor

The constructor is evoked by the keyword “new” and takes no parameters:

```
new CompiledObject();
```

You shouldn’t encounter the need to use the constructor since the only instances of `CompiledObject` that can conceivably be useful are those returned by `doCompile` method of `MathParser`.

#### Public Methods

None.

#### Public Properties

`CompiledObject` has three public instance properties.

```
instance.PolishArray : Array
```

When the instance is returned by `doCompile` method of `MathParser`, the array represents a mathematical formula in the Polish notation.

```
instance.errorStatus : Number
```

When the instance is returned by `doCompile` method of `MathParser`, the property has value 1 if an error is found and 0 otherwise.

```
instance.errorMessage : String
```

When the instance is returned by `doCompile` method of `MathParser`, the string contains a message to the user indicating where in the input a mistake was found.

### `flashandmath.as3.parsers.MathParser`

#### Description

An instance of `MathParser` (you create an instance using the class’s constructor described below) will compile a string that represents a mathematical formula (usually the user’s input) and then calculate the values of the compiled formula for given values of variables that are recognized by the instance. “Compiling” consists of

rewriting a formula in a form suitable for evaluation; that is, in the Polish notation. Compiling will be successful if the user obeys by the simple syntax rules described at the end of this section.

## Constructor

The constructor is evoked with the word “new”:

```
new MathParser(parameter1:Array)
```

The constructor takes one parameter which is an array. For the MathParser’s instance to do what you want it to do, the parameter has to be an array of strings. The strings represent the names of variables that the instance will recognize. For example:

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

The instance “procFun” will recognize the variables x and y.

```
var procFormula:MathParser = new MathParser(["t"]);
```

The instance “procFormula” will recognize t as a variable. MathParser knows the constants e and pi. Do not enter them into the constructor.

**Note:** Variables have to be entered as strings. It is:

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

and not:

```
procFormula:MathParser = new MathParser([x]);
```

Variables can be composed of more than one letter, e.g.:

```
procFormula:MathParser = new MathParser(["tension"]);
```

It is important to remember the order in which you pass your variables to the constructor since the evaluator method of the parser will expect values for those variables in the same order.

## Public Methods

MathParser has two public instance methods.

```
instance.doCompile(parameter1:String): CompiledObject
```

The method takes a string (typically a mathematical formula entered by the user) and returns an instance of CompiledObject. If no mistakes in syntax are found, the PolishArray property of the returned CompiledObject instance represents the formula in the Polish notation, errorStatus=0, errorMes= “”. If a mistake is found, errorStatus=1, errorMes contains a message indicating where the mistake was found, PolishArray = [ ];

```
instance.doEval(parameter1:Array,parameter2:Array): Number
```

The method takes two parameters, both arrays. For the method to be useful, the first array must be the PolishArray property of a CompiledObject returned by doCompile method. The second parameter represents an array of numerical values for the variables recognized by the instance of MathParser. (The same variables that you passed to the constructor of your MathParser instance.) Under these conditions, doEval will return the value of the formula represented by the PolishArray for the specified values of the variables.

## Public Properties

None.

## Syntax Accepted by MathParser

The parser expects calculator-like syntax: e.g.:

```
sin(2*x^2)-e^-x+tan(pi*x)/2
```

Multiplication must be entered as \*. Arguments of functions must be enclosed in parentheses. The parser is case-insensitive and blind to white spaces. It recognizes the constants e and pi. Here is the list of functions that the parser knows:

$\sin(\cdot)$ ,  $\cos(\cdot)$ ,  $\tan(\cdot)$ ,  $\text{asin}(\cdot)$ ,  $\text{acos}(\cdot)$ ,  $\text{atan}(\cdot)$ ,  $\ln(\cdot)$ ,  $\text{sqrt}(\cdot)$ ,  $\text{abs}(\cdot)$ ,  $\text{ceil}(\cdot)$ ,  $\text{floor}(\cdot)$ ,  $\text{round}(\cdot)$ ,  $\text{max}(\cdot, \cdot)$ ,  $\text{min}(\cdot, \cdot)$ .

Addition, multiplication, division and exponentiation are denoted by the usual symbols +, \*, /, ^, subtraction and unary minus by - .

## flashandmath.as3.parsers.RangeObject

### Description

RangeObject is a helper class for RangeParser. It creates a convenient datatype to be returned by parseRangeTwo and parseRangeFour methods of RangeParser. This datatype is an object with three properties listed below which comprise the results of compiling the user's range input for two variables, say x and y, or for one variable, say a parameter t. In most of our applets the range boxes allow for numerical entries as well as for entries containing pi, like pi/4, 2\*pi etc.. Such entries have to be parsed and checked for validity.

### Constructor

The constructor is evoked by the keyword "new" and takes no parameters:

```
new RangeObject();
```

You shouldn't encounter the need to use the constructor since the only instances of RangeObject that can conceivably be useful are those returned by parseRangeTwo or parseRangeFour methods of RangeParser.

### Public Methods

None.

### Public Properties

RangeObject has three public instance properties.

```
instance.Values:Array
```

When the instance is returned by one of the methods of RangeParser, the array represents four range values (if ranges for two variables are being parsed) or two range values if the range for one variable is being parsed.

```
instance.errorStatus:Number
```

When the instance is returned by one of the RangeParser methods, the property has value 1 if an error is found and 0 otherwise.

```
instance.errorMes:String
```

When the instance is returned by one of the RangeParser methods, the string contains a message to the user indicating where in the input a mistake was found.

## flashandmath.as3.parsers. RangeParser

### Description

In most of our applets the range boxes for x and y or for a parameter, say t, allow for numerical entries as well as for entries containing pi, like pi/4, 2\*pi etc.. Such entries have to be parsed and checked for validity.

### Constructor

The constructor is evoked with the word “new” and takes no parameters:

```
new RangeParser();
```

For example

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

### Public Methods

RangeParser has two public instance methods.

```
instance.parseRangeTwo(par1:String, par2:String):RangeObject
```

The method takes two strings (typically the user’s entries in range boxes for a parameter t, for example) and returns an instance of RangeObject. If the entries are found to be valid numerical entries (or valid entries containing pi), and the first entry is less than the second entry, the Values property of the returned RangeObject contains the two range values. In that case, errorStatus=0, errorMes= “”. If a mistake was found, errorStatus=1, errorMes contains a message indicating where the mistake was found, Values=[];

```
instance.parseRangeFour(par1:String, par2:String, par3:String,  
par4:String):RangeObject
```

The method takes four strings (typically the user’s entries for x and y ranges) and returns an instance of RangeObject. If the entries are found to be valid numerical entries (or valid entries containing pi), the first entry is less than the second entry, the third entry is less than the fourth entry, then the Values property of the returned RangeObject contains the corresponding four range values. In that case, errorStatus=0, errorMes= “”. If a mistake was found, errorStatus=1, errorMes contains a message indicating where the mistake was found, Values=[];

### Public Properties

None.

## flashandmath.as3. boards.GraphingBoard

### Description

GraphingBoard is the main visual class for creating customizable planar graphers: function graphers, parametric curves graphers, etc.. An instance of GraphingBoard draws a square graphing board (at runtime), the x and y axes as well as graphs of functions or parametric curves. Any instance of GraphingBoard contains and controls an error display text field where messages to the user can be displayed. It also contains a coordinate display text field in which x and y coordinates are displayed when the user mouses over the graphing board. An instance of GraphingBoard can enable the user to draw within the graphing board with the mouse.

The layout, the colors, and the sizes of all elements of an instance of GraphingBoard are all easily customizable via instance methods of the class.

An instance of GraphingBoard sets x and y ranges (usually based on the user's input), and provides public methods for translating pixel coordinates of a point into its functional coordinates and vice versa.

GraphingBoard extends Sprite. Thus, it inherits from Sprite. In particular, you can control the position of your instance of GraphingBoard within the main movie with the Sprite methods and properties:

```
instance.x  
instance.y
```

These properties set the x and the y coordinates of the upper left corner of your instance of GraphingBoard with respect to the upper left corner of the main movie. (All coordinates are measured in pixels.) Recall that in Flash, the x coordinate increases to the right, the y coordinate increases as you go down.

### Constructor

The constructor is evoked with the word "new" and takes two numerical parameters. The parameters are the width and height, in pixels, of the rectangular graphing board which will be drawn:

```
new GraphingBoard(w:Number, h:Number);
```

We pass to the constructor the size of our graphing board; all other attributes will be set using the methods of the class.

### Public Methods – Graphing Board Appearance

```
instance.changeBackColor(h:Number): void
```

The method controls the background color of the graphing board created by the instance. The numerical parameter should be the desired color in the hexadecimal form.

**Default:** white.

```
instance.changeBorderColorAndThick(h:Number, t:Number): void
```

The method controls the color and thickness (in pixels) of the border of a graphing board created by the instance. The parameter passed to the method should be the hexadecimal form for the desired color.

**Default:** black.

```
instance.setAxesColorAndThick(h:Number, t:Number): void
```

The method sets the color and the thickness, in pixels, of the x and y axes. The color should be passed to the method in its hexadecimal form. Thickness set to 0 produces a line 1 pixel wide whose thickness will not change with rescaling.

**Default:** black, 0

**Note:** Colors and thickness of graphs of functions or curves are passed directly to the graphing method of GraphingBoard as shown later in this guide. That way those attributes can vary from graph to graph.

The `changeBackAlpha` method allows one to create a graphing board with no background by setting the “back alpha” to be 0.

```
instance.changeBackAlpha(alph:Number): void
```

The following methods allow the setup of grid lines, tick marks and labels. Note that labels must fall next to tick marks, but grid lines can be set independently.

```
instance.setGridColor(colo:Number):void  
instance.setGrid(xsize:Number, ysize:Number):void  
instance.drawGrid():void  
instance.setTicks(xsize:Number, ysize:Number, xheight:Number, ywidth:Number):void  
instance.drawTicks():void  
instance.setLabelFormat(f:String, n:Number, c:uint):void  
instance.addLabels():void  
instance.clearLabels():void
```

### Public Methods – Drawing by the User

```
instance.enableUserDraw(h:Number, t:Number): void
```

The method enables the user to draw on the graphing board with the mouse. It sets the color and the thickness, in pixels, of the user’s drawing.

**Default:** enabled in red with thickness 0.

If you want to disable the drawing capability, use the method:

```
instance.disableUserDraw(): void
```

### Public Methods – Error Display

An instance of GraphingBoard controls the text field for displaying error messages to the user. You can control the appearance and the position of the error text field with the following methods.

```
instance.setErrorBoxSizeAndPos(w:Number, h:Number, xpos:Number, ypos:Number): void
```

The parameters determine: the width, the height (in pixels) of the error text field, and its x and y position relative to your instance of GraphingBoard.

**Default:** The text field is positioned over the upper half of the graphing board created by the instance.

You can set visual attributes of the error box with the method:

```
instance.setErrorBoxFormat(c1:Number, c1:Number, c3:Number, s:Number):void
```

The parameters determine: the background color, the border color, the text color, and the text size. All colors should be passed in hex.

**Default values:** white, white, black, 12.

**Note:** The error display text field is a public property of GraphingBoard:

```
instance.ErrorBox
```

Hence, you can control its attributes directly through methods of Flash's TextField class. It is easier, however, to use the methods of GraphingBoard to set properties of the error box.

We made ErrorBox property public to give you easy control over the visibility of ErrorBox and the text displayed in it. (The error box is visible when the user made an error; its text is an error message to the user determined by the kind of error found.)

**Note:** The initial visibility of the error box is set to false.

### Public Methods – Coordinates Display

An instance of GraphingBoard controls the text field for displaying x and y coordinates when the user mouses over the graphing board. You can control the appearance and the position of the coordinate text field with the following methods.

```
instance.setCoordsBoxSizeAndPos(w:Number,h:Number,xpos:Number,ypos:Number):void
```

The parameters determine: the width, the height (in pixels) of the coordinate text field, and its x and y position relative to your instance of GraphingBoard.

**Default:** The text field is positioned in the lower left corner a graphing board created by the instance.

You can set visual attributes of the coordinate box with the method:

```
instance.setCoordsBoxFormat(c1:Number,c1:Number,c3:Number,s:Number):void
```

The parameters determine: the background color, the border color, the text color, and the text size. All colors should be passed in hex.

**Default values:** white, white, black, 12.

You can disable and enable coordinate display box using the methods:

```
instance.disableCoordsDisp():void
```

```
instance.enableCoordsDisp():void
```

**Default:** enabled.

### Public Methods – Tracing Cursors

GraphingBoard provides methods helpful for building graph-tracing mechanisms into your applet. The class allows two styles for the tracing cursor: "cross" and "arrow". The corresponding method is

```
instance.setTraceStyle(s:String):void
```

The string parameter "s" has two values that will produce a cursor: "cross" and "arrow".

**Default value:** "cross"

For each of the two styles you have much control over the size and the color of the tracing cursor. Below are the corresponding methods.

```
instance.setCrossSizeAndThick(s:Number,t:Number): void
```

The method sets the size and the line thickness for the cross cursor.

**Default:** 6, 1.

```
instance.setCrossColor(c:Number): void
```

The method sets the color for the cross cursor. The color should be passed in its hex form.

**Default:** black.

```
instance.setCrossPos(s:Number,t:Number):void
```

The method sets the position of the cross cursor, in pixels, relative to the instance's upper left corner.

**Default:** The upper left corner.

```
instance.crossVisible(b:Boolean): void
```

The method sets the visibility of the cross cursor to "false" or "true" as in:

```
board.crossVisible(true);
```

**Default:** false.

The two last methods are used to move the cursor along a given graph and make it visible or invisible depending on the user's actions.

```
instance.getCrossSize(): Number
```

The method returns cross' size in case you need it for positioning purposes. We have similar methods for the arrow cursor.

```
instance.setArrowSize(s:Number): void
```

The method sets the size for the arrow cursor.

**Default:** 10.

```
instance.setArrowColor(c:Number): void
```

The method sets the color for the arrow cursor. The color should be passed in its hex form.

**Default:** black.

```
instance.setArrowPos(s:Number,t:Number,r:Number): void
```

The method sets the x and y coordinates, in pixels, of the arrow cursor relative to the upper left corner an instance,  $x=s$ ,  $y=t$ . The last parameter represents the rotation of the arrow, counterclockwise, in degrees. The rotation parameter allows the arrow to move along a curve and rotate accordingly.

**Default:** The upper left corner, pointing upwards.

The last method is used to build a tracing mechanism in which an arrow traces a parametric curve when the user slides a slider.

```
instance.arrowVisible(b:Boolean): void
```

The method sets the visibility of the arrow cursor to “false” or “true”.

**Default:** false.

```
instance.getArrowSize(): Number
```

The method returns arrow’s size in case you need it for positioning purposes.

## Public Methods – Graphing

For every instance of GraphingBoard the maximum number of graphs that can be displayed simultaneously should be set via the method:

```
instance.setMaxNumGraphs(a:int): void
```

**Default:** 3.

Before you can make any of the graphing methods work, you have to set x and y ranges so your instance of GraphingBoard knows how to translate functional values to pixel values and vice versa. Once you decided on the ranges for x and y (based on the user’s input or your own assignment), you have to call the method:

```
instance.setVarsRanges(a:Number,b:Number,c:Number,d:Number): void
```

**Default:** no range is set until you call the method.

Once the variables ranges are set, we can draw the x and y axes via the method:

```
instance.drawAxes(): void
```

The following methods allow the user to draw open and closed points as well as asymptotes. Note that the color of all these components is set by the first method.

```
instance.setCircleColor(colo:Number):void  
instance.addOpenPoint(p:Point):void  
instance.addClosedPoint(p:Point):void  
instance.addVertAsymptote(nx:Number):void  
instance.drawPoints():void  
instance.clearPoints():void
```

Graphs of functions or curves are drawn using the method:

```
instance.drawGraph(num:int,thick:Number,aVals:Array,c:Number):Array
```

The method takes four parameters. The first, an integer, is the number of the graph being drawn. This integer must not exceed the maximum number of graphs to be displayed at one time (set by *instance.setMaxNumGraphs(..)* method. The integer will also determine the depth of the graph being drawn in the internal stacking order of your instance of GraphingBoard. (A graph with a higher number will appear in front of a graph with lower number.) The second parameter will determine the thickness of your graph, in pixels. The third parameter should be an array whose elements are two-element arrays. Each of the two-element arrays represents the x and y coordinates of a point within (or outside) the graphing board. These coordinates are in functional terms; they will be translated to pixel values by the method. For the method to work properly, this array, denoted above by aVals, should consist of consecutive points along a graph or a curve which will be joined by lineal elements to form the actual graph. (Although, the method will join the consecutive points in the array by lineal elements regardless what the points represent.) The last parameter is responsible for the color of the graph. The value for a color should be passed in hex.

The drawGraph method returns an array. If the tracing cursor is “cross” (the default), the returned array is the original aVals array which was passed to the method with all x and y coordinates of all points translated to their pixel equivalents. Being able to retrieve this translated array in your applet is valuable for tracing purposes. The returned array gives you the consecutive positions for the cross cursor when tracing the graph. If you do not have a tracing mechanism in your applet, you can ignore the array returned by the method.

### Public Methods – Clearing the Graphing Board

To clear the graphs you use the method:

```
instance.cleanBoard():void
```

The method erases all graphs, the x and y axes, and resets the x and y ranges to undefined. The method does not erase the user’s drawing. The latter is accomplished by

```
instance.eraseUserDraw():void
```

We separated these two methods since, typically, you want the GRAPH button to clean the graphing board but not to erase the user’s drawing. (The user may possibly be experimenting with drawing functions.)

### Public Methods – Other

Here are some other possibly useful methods of GraphingBoard class.

```
instance.getMaxNumGraphs():int
```

Use this method if you are not sure what the maximum number of graphs is set to. Similarly:

```
instance.getBoardSize():Number  
instance.getVarsRanges():Array  
instance.setNumPoints(np:Number):void  
instance.getNumPoints():Number
```

The next four methods allow you to convert functional coordinates to their pixel equivalents and vice versa. Note: these methods will work only if the x and y ranges are set.

```
instance.xtoPix(a:Number):Number  
instance.ytoPix(a:Number):Number  
instance.xtoFun(a:Number):Number  
instance.ytoFun(a:Number):Number
```

The following methods are identical to the latter two above.

```
instance.xfromPix(a:Number):Number  
instance.yfromPix(a:Number):Number
```

A couple of testing methods:

```
instance.isLegal(a:*) :Boolean
```

The method returns “true” if “a” is of the numerical datatype and it is a finite number. Otherwise, the method returns “false”.

```
instance.isDrawable(a:*) : Boolean
```

The method returns “true” if “a” is of the numerical datatype and it is a finite number, and its absolute value does not exceed 5000. Otherwise, the method returns “false”. The reason you may want to have a test of such kind is that an attempt to draw an object, a portion of a graph or a cursor, located very far away from the graphing board (in pixels), you may encounter unexpected results.

Finally, if you want to remove an instance of GraphingBoard at runtime, you should call

```
instance.destroy(): void
```

The method removes all listeners set by your instance of GraphingBoard, clears all drawings, and sets all the Sprites created by the instance to null.

## Public Properties

The only public property of GraphingBoard (except for those inherited from Sprite) is

```
instance.ErrorBox
```

It is a dynamic text field in which error messages to the user can be displayed. As we described above, you can set the size, the position, and the formatting for the text field by using GraphingBoard methods. You can also apply Flash’s TextField class properties and methods to ErrorBox, e.g.:

```
board.ErrorBox.visible=true;  
board.ErrorBox.text="Error in f1(x). "+compObj1.errorMes;
```

## flashandmath.as3.utilities.HorizontalSlider

### Description

Flash CS3 has a slider component. Our class provides a light-weight, easily customizable alternative. The class HorizontalSlider extends Sprite. Hence, it inherits from Sprite. Each instance of HorizontalSlider consists of a track, 3 pixels wide, with four tick marks, and a draggable knob. You can control the length, the style, the colors, and the appearance of the slider using the methods listed below.

### Constructor

The constructor is evoked with the word “new” and takes two parameters. The first parameter is the length, in pixels, of the slider to be created. The second, a String parameter, determines the style of the draggable knob:

```
new HorizontalSlider(len:Number, style:String);
```

The available styles for the knob are” “triangle” and “rectangle”.

### Public Methods

Most methods of the class are meant to give you control over your slider’s visual attributes. The names of the properties are quite self-explanatory.

```
instance.changeKnobColor(c:Number): void
```

**Default:** dark gray.

```
instance.changeKnobSize(c:Number): void
```

**Default:** 8 (in pixels).

```
instance.changeKnobOpacity(n:Number): void
```

**Default:** 1.0 – completely opaque.

You may find this method useful with a rectangular knob if you want the tick marks underneath to show.

```
instance.changeKnobLeftLine(c:Number): void  
instance.changeKnobRightLine(c:Number): void
```

**Default:** white, black.

The methods change the colors of the left and the right outline of the knob to create a 3D effect with your coloring scheme.

```
instance.changeTrackOutColor(c:Number): void  
instance.changeTrackInColor(c:Number): void
```

**Default:** dark gray, white.

The methods change the colors of the outline of the track and the line inside to match your coloring scheme.

```
instance.setKnobPos(p:Number): void
```

The method adjusts the x coordinate of the knob along the horizontal track. 0 corresponds to the left end of the slider. The method is usually used to set the initial position of the knob.

```
instance.getKnobPos(): Number
```

This extremely important method allows you to make your applet respond to the changing horizontal position of the knob as the user drags the knob along the track.

Here are the last two methods of lesser importance.

```
instance.getSliderLen(): Number
```

The method returns the slider's length.

Finally, if you want to remove an instance of `HorizontalSlider` at runtime, you should call

```
instance.destroy(): void
```

The method removes all listeners set by your instance of `HorizontalSlider`, clears all drawings, and sets all the Sprites created by the instance to null.

## Public Properties

Each instance of `HorizontalSlider` has one public property (except for those inherited from `Sprite`):

```
instance.isPressed: Boolean
```

The property is set by the class to “true” if the user presses the mouse button over the knob. The property is reset to the default – “false” when the user releases the mouse button.

## flashandmath.as3.utilities.VerticalSlider

### Description

The class is nearly identical to `HorizontalSlider` class. It has the same methods and properties. The only difference is that the track is drawn vertically and all references in the description above to the x (or horizontal) coordinate should be replaced by references to the y (or vertical) coordinate. Since `HorizontalSlider` inherits from `Sprite`, any instance of `HorizontalSlider` can be positioned vertically via `Sprite.rotation` property. The reason we have a separate class has to do with slight differences in the way the vertical slider is drawn which give it a more pleasing appearance.

### Constructor

The constructor is evoked with the word “new” and takes two parameters. The first parameter is the length, in pixels, of the slider to be created. The second, a `String` parameter, determines the style of the draggable knob:

```
new VerticalSlider(len:Number, style:String);
```

The available styles for the knob are” “triangle” and “rectangle”.

### Public Methods

Same as for `HorizontalSlider`.

### Public Properties

Same as for `HorizontalSlider`.

## flashandmath.as3.tools.SimpleGraph

### Description

Here we describe some properties and methods that we will be using in this book.

### Constructor

```
new SimpleGraph(w, h)
```

The constructor creates a new `GraphingBoard` object `w` pixels wide, `h` pixels high. `instance.board` has all of the properties and methods of the `GraphingBoard` class. The property `board` is protected so it cannot be manipulated outside of the class. For this reason, you will need to see the documentation on the `GraphingBoard` class to learn how to customize the `board` property.

### Public Methods

```
instance.setWindow(stxmin, stxmax, stymin, stymax)
```

checks all four strings for errors and, finding none, sets up a clean board with these ranges for x and y. In each of the following methods, `expr` is a string representing an expression in the single variable `stVar`, and the graph is to be placed on the board at level `num`, with the given thickness and color. For polar and parametric graphs, a range (minimum and maximum) for the parameter also must be specified as strings.

```
instance.graphRectangular(expr:String, stVar:String, num:int,
                           thickness:Number, color:int)
instance.graphPolar(expr:String, stVar:String, stMin:String,
                    stMax:String, num:int, thickness:Number, color:int)
```

```
instance.graphParametric(expr1:String, expr2:String, stVar:String,
    stMin:String, stMax:String, num:int, thickness:Number, color:int)
```

The following method detect errors that might have come up in parsing the range or the graph expressions.

```
instance.hasError():Boolean
```

Once created, we can interact with the SimpleGraph object in a number of ways.

```
instance.rectangularValueAt(num:int, xval:Number):Number
```

returns the value of the expression graphed in *instance* at the value *xval* of the variable.

```
instance.polarValueAt(num:int, tval:Number):Number
```

is similar.

```
instance.parametricValueAt(num:int, tval:Number):Array
```

is similar, but returns a two-element array [x(tval), y(tval) ] instead of a number.

For many applications we do not need to evaluate the graphed expression from outside of the graph, but it is helpful to have the list of points (in pixels) that were used to build the graph.

```
instance.getPixData(nGraph:int):Array
```

returns an array of “triples” of the form [xcrd, ycrd, rotat], where (xcrd, ycrd) is the coordinate pair (in pixels) of a point on the graph, and rotat is a rotation angle that can be used for any orientable cursor that is placed at this point.

Similarly, the following methods get other pieces of information that are important for the graphs already drawn. The last one is used to set the number of points sampled for the creation of the graph.

```
instance.getCompiledObject(nGraph:int):CompiledObject
instance.getVariable(nGraph:int):String
instance.getNumPoints():int
instance.setNumPoints(np):void
```

The method

```
instance.addChildToBoard(displayobject)
```

adds the displayobject as a child of the board so that the masking of the graphing board will also apply to *displayobject*.

The following methods are self-explanatory.

```
instance.removeGraph(num:Number):void
instance.destroy():void
```

## Public Properties

The only public property is the `board` for a SimpleGraph instance. That is, `instance.board` is a GraphingBoard object, so we can use all of the GraphingBoard methods to customize appearance, etc. for this object.