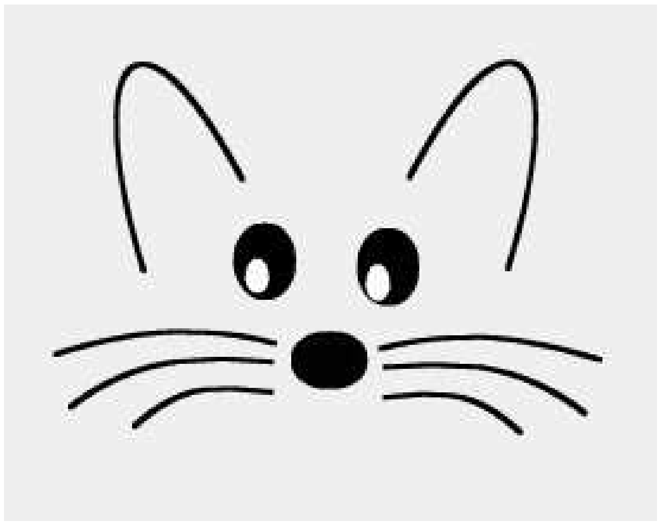
A GUIDE TO WORK WITH SQUEAK MORPH CLASSES

Author:
Onur Aytar
Northeastern University
CSIS

Date:
November 2002



Squeak !3.0

Complimentary file:
morphguide.zip
includes all classes as stationary files

can be downloaded at:
www.ccs.neu.edu/home/aytar/

First revision, November 2002 by Onur Aytar, Northeastern Universty School of Computer and Information Science.

Table of Contents:

Syntax:

_ assignment

“comment”

< > the reader will select the text inside the tags

^ return

Terms:

Action	:	code to evaluate
Event	:	a user interrupt
Morph class	:	a class in Squeak! With display properties.
Morphic class	:	any subclass of Morph class.
Timer	:	a clock interrupt at a specified interval

Part I – Working with Squeak Morph Classes

1. What is the Morph Class?

Morph class is an object with a two dimensional display and visual properties. The root of all morphic objects can be found at the directory “Morphic – Kernel” in Squeak System Browser (the green window)

2. What does a morphic object do?

A morphic object can handle user events and can contain other morphs. It also “steps” meaning that it can act according to time. Besides these, have I mentioned that it is displayed on the screen?

3. How to display a Morph?

A morph must be opened in a Squeak World in order to be displayed. The most common way to do this is to use the “openInWorld” method. The following is an example written in the Work Space (the pink window)

```
m _ Morph new.            "Create a new morph instance"
m openInWorld.            "Show the instance"
```

This will display a morph when done. To do it: select all the text above in the Workspace, right click and click on ‘do it’.

4. How to change the visual properties of a morph?

A morph has visual properties regarding to its color, size and position. The following methods can be used to alter these properties:

- **color: aColor**
- **extent: aPoint**
- **height: aNumber**
- **width: aNumber**
- **position: aPoint**
- **top: aNumber**
- **left: aNumber**

Let’s change the properties of **m**, which we opened in the previous section. Again in the WorkSpace:

```
m color: Color black.     "Make the morph black"
m extent: 200@150.        "Make it this big"
m height: 170.            "I didn't like it, let's make it taller"
m width: 170.             "On the second thought, a square would seem nicer"
m position: 100@100.      "Move, I can't see a thing"
m left: 50.               "A little more to the left, okay"
```

```
m top: 50.                    "A little more upwards, that's good"
```

5 – What is a submorph?

A submorph is a morph which is contained by another morph. This sounds better in the other direction: A morph can nest other morphs and the morphs it nests are called its submorphs.

All morphs have an instance variable called submorphs. The variable submorphs is initialized to be an OrderedCollection and used to track the morphs contained.

All submorphs are depended to their owner. Their positions change when the owner's position changes. Their visibility is also dependent to their owner. They are deleted when the owner is deleted.

Another issue in submorphs is their z-order, meaning which morph is on the front and which is at the back. The z-order follows the opposite order in the OrderedCollection submorphs. First item on the front, last item on the back.

The following are some common submorph methods of the Morph object:

- **addMorph: aMorph**
- **addMorphFront: aMorph**
- **addMorphBack: aMorph**
- **aMorph delete**
- **removeAllMorphs**
- **submorphs**

Let's add some submorphs to **m**. Yeah, m is getting popular. Again in the Work Space:

```
b _ BorderedMorph new.    "Create a new BorderedMorph instance"
m addMorph: b.            "Add the instance to m"
```

Now, probably b is not inside m. Try to move m, you will see b is moving along with it. But we definitely want b to be inside m.

```
b position: m position + (20@20).
```

Allright, it is done.

Now, let's add another morph.

```
r _ RectangleMorph new.
m addMorph: r.
r position: m position + [20@20).
```

Okay, we added it but where did b go? b is actually at the back of r now. I don't know you, but I wanted it to be in the frontline. Hence,

```
r delete.                        "Goodbye r"
r _ RectangleMorph new.
m addMorphBack: r.
r position: m position + [20@20).
r extent: 70@70.
```

Now this was what I wanted to do. Why? … I just did.

6 – This is it? Are we done?

Nope, we are not done. But this is all for the Work Space action. The next step will be "The Making of a Button in Squeak" and we will be creating an object which can handle events.

Part II – The Making of a Button with Squeak

In this section we will create a new button object for our use. Our new object will demonstrate inheritance both through extension and mapping. However we would like to keep everything organized. So we start with the basics.

1 – How to create a new object category in Squeak

In order to accomplish this task, we should first open the green window, the System Browser. On the top left pane of the System Browser:

- Right click
- Select “add item…”
- Enter a category name.

2 – How to create a new object in Squeak

To create a new object, we must be browsing a category. So, click on the category you just created. The big bottom pane will change to the following:

```
Object subclass: #NameOfSubclass
        instanceVariableNames: ‘’
        classVariableNames: ‘’
        poolDictionaries: ‘’
        category: ‘<Your category-name here>’
```

Here we can enter the name of our object and decide which object it is extending. Also we can declare instance variables.

First lets determine what we need for our button.

- A name for the button object
- A label
- A border
- An action for click event
- An action for double click event

We will call our button: SimpleButton.

The name is okay, but we still need borders and a label.

In order to have borders we can extend BorderedMorph class. It will give us all we need for background.

As for label, our object must have a submorph which can display strings. Therefore we will have an instance variable called label.

For actions we will have two other instance variables called clickAction and doubleClickAction.

Finally we can create our button.

```
BorderedMorph subclass: #SimpleButton
        instanceVariableNames: 'label clickAction doubleClickAction '
        classVariableNames: ''
        poolDictionaries: ''
        category: '<your category name here>'
```

And press ALT + s to accept the changes we made. Our Button takes its place on the object list. Well done.

Now that we extended the BorderedMorph, our Button has all of its methods. Moreover we will declare label as a StringMorph, which will allow us to use all of its methods too.

3 – Adding methods to the SimpleButton

We already have a SimpleButton with lots of methods inherited from the super class BorderedMorph. However this is not enough. To have a fully functional button we will add the following methods to our SimpleButton.

• clickAction	"get the clickAction"
• clickAction: aBlock	"set the clickAction"
• doubleClickAction	"get the doubleClickAction"
• doubleClickAction: aBlock	"set the doubleClickAction"
• foreColor	"get foreColor – map to StringMorph"
• foreColor: aColor	"set foreColor – map to StringMorph"
• label	"get label – map to StringMorph"
• label: aString	"set label – map to StringMorph"
• refresh	"center label"

Moreover we will need to override the methods listed below

• click: evt	"do clickAction if exists"
• doubleClick: evt	"do doubleClickAction if exists"
• extent: aPoint	"set extent and refresh"
• handlesMouseDown: evt	"yes we handle it"
• height: aNumber	"set height and refresh"
• width: aNumber	"set width and refresh"

To add methods, we have to select the SimpleButton from the System Window and then select a classification from the next pane. Then we can enter the method code at the bottom pane.

4 – The methods of SimpleButton

click: evt
```
"When clicked do clickAction if it is not nil"
clickAction ~= nil
ifTrue: [clickAction value]! !
```

clickAction
```
^clickAction
```

clickAction: aBlock
```
clickAction _ aBlock
```

doubleClick: evt
```
"When double clicked do doubleClickAction if it is not nil"
doubleClickAction ~= nil
ifTrue: [doubleClickAction value]
```

doubleClickAction
```
^doubleClickAction
```

doubleClickAction: aBlock
```
doubleClickAction _ aBlock
```

extent: aPoint
```
"Let the super class do the work then center the label"
super extent: aPoint.
self refresh
```

foreColor
```
^label color
```

foreColor: aColor
```
label color: aColor
```

handlesMouseDown: evt
```
"Yes, I handle mouse press events"
^ true
```

height: aNumber
```
super height: aNumber.
self refresh.
```

initialize
```
"Initialize my properties and instance variables"
super initialize.
"initialized all properties and instance variables inherited from super class"
"Now initialize my specific properties and instance variables"
self color: Color lightGray.
self borderColor: Color black.
```

```
    label _ StringMorph new.
    label contents: ‘SimpleButton’.
    self addMorph: label.
    self extent: 80@25
```

label

```
    ^label contents
```

label: aString

```
    “Let the StringMorph do the work then center the label.
      StringMorh automatically resize to fit contents.”
    label contents: aString.
    Self refresh
```

mouseDown: evt

```
    “wait for other events”
    evt hand waitForClicksOrDrag: self event: evt
```

refresh

```
    “center label”
    label top: self top + ((self height – label height) // 2).
    label left: self left + ((self width – label width) // 2)
```

width: aNumber

```
    super width: aNumber.
    self refresh
```

That’s all the methods we need to write for SimpleButton

5 – Testing the SimpleButton

We now have a SimpleButton, why not use it. We can use the Work Space for testing our button.

```
s _ SimpleButton new.
s openInWorld.
s label: ‘Window’.
s clickAction: [SystemWindow new openInWorld].
```

Okay, the button is there and it opens a new SystemWindow when we click on it.

Can we change the border width of the button?

```
s borderWidth: 1.
```

Yes, it really changes. Although we haven’t written a borderWidth method, SimpleButton inherited it from its super class BorderedMorph.

6 – Are we done?

Well, we are close to complete the basic information required for using morphic objects in Squeak. Next chapter is about steps and time intervals. It is an easy concept to cover. Yet, there is a lot more to be learned in order to be able to say we are done.

Part III – The Making of a Spectrum

1 – Are we really making a spectrum analyzer?

No, we are not. I wish I knew a lot more about sound patterns, unfortunately I don't. What we are going to do is a Morph which acts like a Spectrum analyzer using random values.

Why are we doing this? To demonstrate the step method of course. In this part we will examine how the timer works for a Morph object.

2 – The regular stuff first

This part is really an easy one, we will create a simple DemonstrationMorph object, which has two instance variables:

• interval	"How frequent the display will change"
• rand	"A random number generator"

interval is a number between 0 an 1000. rand will be a Random object.

Here are the class definition and the initialize method

```
Morph subclass: #DemonstrationMorph
        instanceVariableNames: 'interval rand '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Aytar – Common Controls'
```

initialize

```
        super initialize.
        rand _ Random new.
        super extent: 10@60.
        self color: Color black.
        self interval: 200.
```

Now, if you noticed the bold line in the initialize method, there is something wrong about it. It says: "super extent: 10@60" Why do we call the super class. Simply because we will disable resizing, which is our next discussion.

3 – Not allowing direct resizing

Our new spectrum will be a fixed object lacking flexibility. So we would better disable some of its methods. Since we extended Morph class, our new object has a lot of methods by default. Look at the following methods.

extent: aPoint

```
        "do nothing"
```

height: aNumber
```
    "do nothing"
```

width: aNumber
```
    "do nothing"
```

We have overridden extent, height and width methods, and they now do nothing. So, if anyone calls them directly, nothing will happen.

This was why we used "super extent: 10@60" in the initialize method. This way we called the extent method from the Morph class.

4 – More overriding for timer

In the previous section, we have overridden methods to disable them. Now we will override other methods to enable them.

The step and stepTime methods are used for timing by Morph classes. In the Morph class they are not working by default. We are going to override them as below:

stepTime
```
    ^interval
```

step
```
    self drawRectangles
```

stepTime method specifies how frequently the step method will be invoked. The measurement units are milliseconds, so if interval is 1000, then the step method will be invoked once every second.

We have overridden the step method to call drawRectangles method every interval * milliseconds.

5 – Some other timer methods that we don't need in this object

There are some other important timer methods, which are useful to know.

- startStepping – starts the timer
- stop – stops the timer
- isStepping – returns whether the timer is active or not.

All these methods are inherited from Morph class, as we extend it and there is no need to override them.

6 – What is left for the spectrum?

We still need to implement the visual parts and property set/get methods. Here they are with comments.

```
drawRectangles
"redraw the spectrum"
        | r n currentTop|
"first clear all current rectangles"
        self removeAllMorphs.
"now redraw them"
        currentTop _ self top + self height – 5.
        N _ rand nextInt: 12.
        (1 to: n)
        do: [:i |
                r _ self newElement: i.
                r top: currentTop.
                R left: self left.
                Self addMorph: r.
                currentTop _ currentTop – 5.
        ].

newElement: n
"Create a new RectangleMorph with the color according to its position"
        | r |
        r _ RectangleMorph new.
        R extent: 10@4.
"if n is less then 5, rectangle is light green"
        n < 5
        ifTrue: [
                r color: Color green.
                R borderColor: Color lightGreen.
                ^r
        ].
"if n is greater than 8, rectangle is orange"
        n > 8
        ifTrue: [
                r color: Color orange.
                R borderColor: Color orange.
                ^r
        ].
"Otherwise rectangle is yellow"
        r color: Color yellow.
        R borderColor: Color yellow.
        ^r

interval
        ^interval

interval: aNumber
        interval _ aNumber
```

And there we are. We have a useless but nice looking object. Our aim was to learn the stepping mechanism anyway. As long as we reach our goal, writing a little object is not a loss.

7 – Are we done yet?

You now have all the basic information you need to use Morph classes. Congratulations.

The tutorial will continue with making of a ShapeButton and a ListBox. Both those controls are much more sophisticated but they have the same idea behind them.

You have two choices: either you can continue reading the tutorial, or you can go look at the classes yourself. If you are going on your own do not forget to click on “?” in the System Browser. It contains valuable information about the design of objects.

Part IV – The Making of a ShapeButton

In this part we are getting down to the business. We will create a Morph subclass with 25 instance variables, over 70 instance methods and 4 class methods of its own. The numbers may seem scary but, our path is smooth and clean. Nevertheless, first things first.

1 - What will the ShapeButton do?

Before creating an object we must define a specification document for that object, which points the requirements the object meets.

ShapeButton Specifications

1. **Visual Aspects:**
   - The ShapeButton should be able have <u>rectangular, oval or rounded shapes.</u>
   - The ShapeButton should be able to have a <u>border.</u>
   - The ShapeButton should be able to have a <u>hover (mouse over) state.</u>
   - The ShapeButton should be able to have a <u>down (mouse down) state.</u>
   - The ShapeButton must have a <u>label.</u>
   - The label must have <u>alignment options.</u>
   - <u>Hover and down states</u> must enable the user to <u>switch them off and on.</u>

2. **Functionality:**
   - The ShapeButton must be able to <u>handle all mouse events except drag&drop.</u>
   - The ShapeButton must have an <u>index value</u> in case if it is needed.

Here we go, if we take the underlined expressions from the specifications, we will see that there is a lot to do with this object.

2 – Design Decisions

2.1. Which class to extend?

Now, this is a tough question. We know that the ShapeButton requires borders, but a BorderedMorph or a RectangleMorph is out of question, since ShapeButton may have an oval shape too. From the same point of view, extending EllipseMorph class is also impossible.

Therefore, we are going to extent the Morph class itself, and use mapping for the shape. What was mapping? Mapping is linking an object's methods to some of its sub-objects.

2.2. Deciding on submorphs.

So far so good; now that we have decided to use mapping for most of the visual properties, we should select the submorphs and create instance variables for them.

As we did in SimpleButton, we will use a StringMorph for the label.

In addition we will have an instance variable called "mask", which will become a RectangleMorph or an EllipseMorph on request.

So we have two instance variables for submorphs:

- label
- mask

2.3. Problems in mapping due to the states

Normally, we could map methods like color, borderColor to the mask we are using as below:

color: aColor

```
mask color: aColor
```

color

```
^ mask color
```

However, this will not work due to the hover and down states. We must keep the original color and other variables of the ShapeButton. So we need the following instance variables:

- normalBorderColor
- normalBorderWidth
- normalColor
- normalForeColor

2.4. Issues about hover and down states

From the specifications we know that these states can be turned on and off. Yes, we need two additional instance variables which are □eystro values:

- hasHoverEffect
- hasMouseDownEffect

Besides we will need to store the visual properties of these states. Yes, even more instance variables:

- hoverBorderColor
- hoverBorderWidth
- hoverColor
- hoverForeColor

- downBorderColor

- downBorderWidth
- downColor
- downForeColor

Wow, this is a lot. But it can't be helped.

2.5. Actions upon mouse events

I guess you remember what we did for SimpleButton. We had a clickAction and a doubleClickAction instance variables. This time we need a lot more, since the specifications state that the ShapeButton must be able to handle all mouse events except drag&drop. Below is the list of action variables, which will be blocks.

• clickAction	Action to be taken when clicked
• doubleClickAction	Action to be taken when double clicked
• mouseDownAction	Action to be taken when mouse is pressed
• mouseUpAction	Action to be taken when mouse is released
• mouseStillDownAction	Action to be taken if mouse is still pressed
• mouseOverAction	Action to be taken when hover
• mouseLeaveAction	Action to be taken when roll off.

These events are all implemented in Morph class, however, we will need to override them to use these instance variables.

2.6. Did we miss something?

Unfortunately we did. We still need an index and an alignment variable. So the final two instance variables are:

- index
- alignment

index will simply be a number, alignment will be a number between 1 and 3.

Well done, we have completed the instance variables.

2.7. What about methods?

Well, we have 25 instance variables. That means we will have 50 methods for setting and retrieving their values.

Plus, we will need to override the event methods and sizing methods as we did in SimpleButton.

And of course we will have initialization and private methods, some of which we will put into the class rather than the instance.

2.8. Why put methods in the class?

First of all we don't (I don't know you, but I don't) want these methods called by the user in runtime easily.

Besides we already have a lot of methods, we really should not populate around with unusable methods.

2.9. Method categories

Since we have a lot of methods, it will be useful to categorize them. To create a new category for methods, we need to right-click on the third pane from the left in the System Browser and select "new category…". We will have the following categories:

- event handling
- events – processing
- hover effect
- initialization
- mouse down effect
- visual properties

Naturally we will create these categories after we create the ShapeButton object, speaking of which we now are ready for.

3 – The creation of ShapeButton

```
Morph subclass: #ShapeButton
        instanceVariableNames: 'label hasHoverEffect hasMouseDownEffect
normalColor normalForeColor normalBorderColor normalBorderWidth
hoverColor hoverForeColor hoverBorderColor hoverBorderWidth downColor
downForeColor downBorderColor downBorderWidth mouseOverAction
mouseLeaveAction mouseDownAction mouseStillDownAction mouseUpAction
clickAction doubleClickAction mask alignment index '
        classVariableNames: ''
        poolDictionaries: ''
        category: '<your category here>'
```

There we go, by accepting this we officially create the ShapeButton.

4 – Instance methods of ShapeButton

Assuming that you already read the previous chapter, I believe that you already know how we map methods, how we set and get variables. Therefore, we will only discuss the important methods, which are introduced for the first time. All of the methods can be found at the end of this chapter.

4.1. Handling mouse over

```
handlesMouseOver: evt
        ^ true.
```

By overriding this method of Morph class, we are stating that the ShapeButton handles mouse over event.

4.2. Handling mouse still down

```
handlesMouseStillDown: evt
        ^ true.
```

The same as mouse over, we are stating that ShapeButton handles this event too.

4.3. What to do upon events

We will override the methods below:

• click: evt	what to do when clicked
• doubleClick: evt	what to do when double clicked
• mouseDown: evt	what to do when mouse is pressed
• mouseEnter: evt	what to do when mouse is over
• mouseLeave: evt	what to do when mouse leaves
• mouseStillDown: evt	what to do if mouse is still pressed
• mouseUp: evt	what to do when mouse is released

For mouseDown, mouseUp, mouseEnter and mouseLeave events we will need to implement the hover and down effects. Here is the code and explanations.

```
mouseDown: evt
        hasMouseDownEffect
        ifTrue: [
                self color: downColor.
                Self borderColor: downBorderColor.
                Self foreColor: downForeColor.
                Self borderWidth: downBorderWidth.
        ].
        mouseDownAction ~= nil
        ifTrue: [mouseDownAction value].

        Evt hand waitForClicksOrDrag: self event: evt
```

If hasMouseDownEffect is true, then the visual properties are set to the down state settings.

After that, if an action has been assigned to the event, that action is invoked.

```
mouseUp: evt
    hasMouseDownEffect
    ifTrue: [
        self color: normalColor.
        Self borderColor: normalBorderColor.
        Self foreColor: normalForeColor.
        Self borderWidth: normalBorderWidth.
    ].
    mouseUpAction ~= nil
    ifTrue: [mouseUpAction value].
```

When mouse is released, then the button is not in down state anymore. So we set the original values back.

And of course we invoke the action if it exists.

This is the same for the mouseEnter and mouseLeave events, as it can be seen in their methods. The only difference is that we are switching the hover and normal state.

```
mouseEnter: evt
    hasHoverEffect
    ifTrue: [
        self color: hoverColor.
        Self borderColor: hoverBorderColor.
        Self foreColor: hoverForeColor.
        Self borderWidth: hoverBorderWidth.
    ].
    mouseOverAction~= nil
    ifTrue: [mouseOverAction value].
```

```
mouseLeave: evt
    hasHoverEffect
    ifTrue: [
        self color: normalColor.
        Self borderColor: normalBorderColor.
        Self foreColor: normalForeColor.
        Self borderWidth: normalBorderWidth.
    ].
    mouseLeaveAction ~= nil
    ifTrue:[mouseLeaveAction value]
```

The states are complete but we still have other actions. Here they are:

```
click: evt
    clickAction ~= nil
    ifTrue: [clickAction value]
```

```
doubleClick: evt
        doubleClickAction ~= nil
        ifTrue: [doubleClickAction value]

mouseStillDown: evt
        mouseStillDownAction ~= nil
        ifTrue:[mouseStillDownAction value]
```

Here we are, we completed all events.

4.4. Overcoming the mapping problem

As we discussed earlier, direct mapping does not work in our case. Therefore we will write multiple methods and keep our variables.

```
color: aColor
        mask color: aColor.

color
        ^ mask color.
```

These and related methods such as borderWidth will be overridden to avoid problems from inheritance.

```
normalColor: aColor
        normalColor _ aColor.
        Self color: aColor.

normalColor
        ^ normalColor

hoverColor: aColor
        hoverColor _ aColor

hoverColor
        ^ hoverColor

mouseDownColor: aColor
        downColor _ aColor

mouseDownColor
        ^downColor
```

This is the way we will store our variables. And note that setting normalColor automatically sets the color of the mask. Other related methods can be found at the end of this chapter.

4.5. The instance methods cooperating with class methods

Some of our instance methods will cooperate with class methods. As mentioned earlier, class methods are same for all instances. You can also think as they are static methods in a Java object class.

initialize
"initialize the control with default properties."

```
    Super initialize.
    **Super color: Color transparent.**
    Self shape: 3.
    Alignment _ 3.
    Label _ StringMorph new.
    Label initialize.
    Self label: 'ShapeButton'.
    Self addMorph: label.
    **ShapeButton defaultProperties: self.**
    Self refresh.
```

The bottom bold line is where the class method is called. The instance simply passes itself to the method and the class shapes up the instance.

Another important issue here is that we set the color of the super class to transparent. Why do we do that? Well, since we map color to the mask, and the mask may not cover all of the ShapeButton area, it is good for the ShapeButton itself to be transparent. Don't you think so?

Shape: anInteger
" Set the shape of the button. 1 – Rectangle, 2 – Oval, 3 – Rounded Rectangle"

```
    anInteger < 1
    ifTrue: [^false].
    anInteger > 3
    ifTrue:[^false].
    Mask ~= nil
    ifTrue: [mask delete].
    anInteger = 1
    ifTrue:[
        **mask _ ShapeButton shapeRectangle: self.**
        Self addMorphBack: mask.
        ^ true
    ].
    anInteger = 2
    ifTrue: [
        **mask _ ShapeButton shapeOval: self.**
        Self addMorphBack: mask.
        ^ true
    ].
    anInteger = 3
    ifTrue: [
        **mask _ ShapeButton shapeRoundedRectangle: self.**
        Mask cornerStyle: #rounded.
        Self addMorphBack: mask.
        ^ true
    ].
    Label comeToFront.
```

Once again the bold lines call class methods. However shape method is an important method as we discussed in design decisions.

Shape: method deletes current mask and replaces it with the new one using delete and addMorphBack methods, which we discussed in the first chapter.

I do not know why I used “label comeToFront”, since I used “addMorphBack”, but “comeToFront” sets z-order of a submorph to 1.

5. Class methods

We are almost done. There are not many class methods and they are simple enough. They basically do some regular tasks for all instances.

defaultProperties: aShapeButton
```
        | s |
        s _ aShapeButton.
        s normalColor: Color gray.
        s normalBorderColor: Color black.
        s normalForeColor: Color black.
        s normalBorderWidth: 1.
        s hoverColor: Color gray.
        s hoverBorderWidth: 2.
        s hoverForeColor: Color black.
        s hoverBorderColor: Color black.
        s mouseDownColor: Color black.
        s mouseDownForeColor: Color gray.
        s mouseDownBorderWidth: 2.
        s mouseDownBorderColor: Color white.
        s turnOnHoverEffect.
        s turnOnMouseDownEffect.
        s extent: 80@30.
```

defaultProperties simply gives an instance a regular set of values, and make it look good enough when initialized and opened.

shapeOval: aShapeButton
```
        | m |
        m _ EllipseMorph new.
        m extent = aShapeButton extent.
        m position = aShapeButton position.
        aShapeButton extent = m extent.
        ^ m
```

shapeRectangle: aShapeButton
```
        | m |
        m _ RectangleMorph new.
        m extent: aShapeButton extent.
        m position: aShapeButton position.
        ^ m
```

shapeRoundedRectangle: aShapeButton
```
        | m |
        m _ RectangleMorph new.
        m cornerStyle: #rounded.
        m extent: aShapeButton extent.
        M position: aShapeButton position.
        ^ m
```

These three methods create new RectangleMorph or EllipseMorph instances and send them back to the ShapeButton instance to wear as a mask.

See, there is really nothing sophisticated about class methods except that they are called by the class name.

6 – Chapter summary

In this chapter we coped with mapping and extending problems. We covered all mouse events except drag&drop and we handled two submorphs with z-order. Finally, we made an introduction to class methods.

Now, we have a versatile ShapeButton, which in the next chapter we will use for a totally different purpose. If there will be a next chapter.

This part ends with the methods of ShapeButton, but you will probably look at them from the System Browser. Just in case they are here.

7 – All methods of ShapeButton

Instance Methods

methodsFor: 'event handling'

click: evt
```
    clickAction ~= nil
    ifTrue: [clickAction value]
```

doubleClick: evt
```
    doubleClickAction ~= nil
    ifTrue: [doubleClickAction value]
```

handlesMouseDown: evt
```
    ^ true.
```

handlesMouseOver: evt
```
    ^ true.
```

handlesMouseStillDown: evt
```
    ^ true.
```

mouseDown: evt
```
    hasMouseDownEffect
    ifTrue: [
        self color: downColor.
        self borderColor: downBorderColor.
        self foreColor: downForeColor.
        self borderWidth: downBorderWidth.
    ].
```

```
   mouseDownAction ~= nil
   ifTrue: [mouseDownAction value].
   evt hand waitForClicksOrDrag: self event: evt
```

mouseEnter: evt

```
   hasHoverEffect
   ifTrue: [
      self color: hoverColor.
      self borderColor: hoverBorderColor.
      self foreColor: hoverForeColor.
      self borderWidth: hoverBorderWidth.
   ].
   mouseOverAction~= nil
   ifTrue: [mouseOverAction value].
```

mouseLeave: evt

```
   hasHoverEffect
   ifTrue: [
      self color: normalColor.
      self borderColor: normalBorderColor.
      self foreColor: normalForeColor.
      self borderWidth: normalBorderWidth.
   ].
   mouseLeaveAction ~= nil
   ifTrue:[mouseLeaveAction value]
```

mouseStillDown: evt

```
   mouseStillDownAction ~= nil
   ifTrue:[mouseStillDownAction value]
```

mouseUp: evt

```
   hasMouseDownEffect
   ifTrue: [
      self color: normalColor.
      self borderColor: normalBorderColor.
      self foreColor: normalForeColor.
      self borderWidth: normalBorderWidth.
   ].
   mouseUpAction ~= nil
   ifTrue: [mouseUpAction value].
```

methodsFor: ‘events-processing’

clickAction

```
   ^clickAction
```

clickAction: aBlock

```
   clickAction _ aBlock
```

doubleClickAction

```
   ^doubleClickAction
```

doubleClickAction: aBlock

```
   doubleClickAction _ aBlock
```

mouseDownAction
```
^mouseDownAction
```

mouseDownAction: aBlock
```
mouseDownAction _ aBlock
```

mouseEnterAction
```
^mouseOverAction
```

mouseEnterAction: aBlock
```
mouseOverAction _ aBlock
```

mouseLeaveAction
```
^mouseLeaveAction
```

mouseLeaveAction: aBlock
```
mouseLeaveAction _ aBlock
```

mouseStillDownAction
```
^ mouseStillDownAction
```

mouseStillDownAction: aBlock
```
mouseStillDownAction _ aBlock
```

mouseUpAction
```
^ mouseUpAction
```

mouseUpAction: aBlock
```
mouseUpAction _ aBlock
```

methodsFor: 'hover effect'

hasHoverEffect
```
hasHoverEffect = nil
ifTrue: [hasHoverEffect _ false].
^ hasHoverEffect
```

hoverBorderColor
```
^hoverBorderColor
```

hoverBorderColor: aColor
```
hoverBorderColor _ aColor
```

hoverBorderWidth
```
^ hoverBorderWidth
```

hoverBorderWidth: aNumber
```
hoverBorderWidth _ aNumber
```

hoverColor
```
^ hoverColor
```

hoverColor: aColor
```
    hoverColor _ aColor
```

hoverForeColor
```
    ^hoverForeColor
```

hoverForeColor: aColor
```
    hoverForeColor _ aColor
```

normalBorderColor
```
    ^normalBorderColor
```

normalBorderColor: aColor
```
    normalBorderColor _ aColor.
    self borderColor: aColor
```

normalBorderWidth
```
    ^ normalBorderWidth
```

normalBorderWidth: aNumber
```
    normalBorderWidth _ aNumber.
    self borderWidth: aNumber
```

normalColor
```
    ^ normalColor
```

normalColor: aColor
```
    normalColor _ aColor.
    self color: aColor
```

normalForeColor
```
    ^normalForeColor
```

normalForeColor: aColor
```
    normalForeColor _ aColor.
    self foreColor: aColor
```

turnOffHoverEffect
```
    hasHoverEffect _ false
```

turnOnHoverEffect
```
    hasHoverEffect _ true
```

methodsFor: ‘initialization’

index
```
    ^ index
```

index: aNumber
```
    index _ aNumber.
```

initialize
```
   “initialize the contol with default properties.”
   super initialize.
   super color: Color transparent.
   self shape: 3.
   alignment _ 3.
   label _ StringMorph new.
   label initialize.
   self label: ‘ShapeButton’.
   self addMorph: label.
   ShapeButton defaultProperties: self.
   self refresh
```

methodsFor: ‘mouse down effect’

hasMouseDownEffect
```
   hasMouseDownEffect = nil
   ifTrue:[hasMouseDownEffect _ false].
   ^ hasMouseDownEffect
```

mouseDownBorderColor
```
   ^downBorderColor
```

mouseDownBorderColor: aColor
```
   downBorderColor _ aColor
```

mouseDownBorderWidth
```
   ^downBorderWidth
```

mouseDownBorderWidth: aNumber
```
   downBorderWidth _ aNumber
```

mouseDownColor
```
   ^downColor
```

mouseDownColor: aColor
```
   downColor _ aColor
```

mouseDownForeColor
```
   ^downForeColor
```

mouseDownForeColor: aColor
```
   downForeColor _ aColor
```

turnOffMouseDownEffect
```
   hasMouseDownEffect _ false
```

turnOnMouseDownEffect
```
   hasMouseDownEffect _ true
```

methodsFor: ‘visual properties’

alignCenter
```
    self alignment: 3
```

alignLeft
```
    self alignment: 1
```

alignRight
```
    self alignment: 2
```

alignment
```
    ^ alignment
```

alignment: aNumber
```
    "1 – Left aligned, 2 – Right aligned, 3 – Center"
    alignment _ aNumber.
    self refresh.
```

borderColor
```
    ^ normalBorderColor
```

borderColor: aColor
```
    mask borderColor: aColor
```

borderWidth
```
    ^ normalBorderWidth
```

borderWidth: aNumber
```
mask borderWidth: aNumber
```

color
```
    ^ normalColor
```

color: aColor
```
    mask color: aColor
```

extent: aPoint
```
    super extent: aPoint.
    self refresh
```

foreColor
```
    ^ normalForeColor
```

foreColor: aColor
```
    label color: aColor
```

height: anInteger
```
    super height: anInteger.
    self refresh
```

label
```
    ^ label contents
```

label: aString
```
    label contents: aString.
    self refresh
```

```
refresh
    label ~= nil
    ifTrue: [
        alignment = 1
        ifTrue: [
            label left: self left + 2
        ].
        alignment = 2
        ifTrue: [
            label left: self left + self width – label width – 2.
        ].
        alignment = 3
        ifTrue: [
            label left: self left + ((self width – label width) // 2).
        ].
        label top: self top + ((self height – label height) // 2).
    ].
    mask ~= nil
    ifTrue: [
        mask extent: self extent.
        mask position: self position.
    ]

shape: anInteger
    “ Set the shape of the button. 1 – Rectangle, 2 – Oval, 3 – Rounded Rectangle”
    anInteger < 1
    ifTrue: [^false].
    anInteger > 3
    ifTrue:[^false].
    mask ~= nil
    ifTrue: [mask delete].
    anInteger = 1
    ifTrue:[
        mask _ ShapeButton shapeRectangle: self.
        self addMorphBack: mask.
        ^ true
    ].
    anInteger = 2
    ifTrue: [
        mask _ ShapeButton shapeOval: self.
        self addMorphBack: mask.
        ^ true
    ].
    anInteger = 3
    ifTrue: [
        mask _ ShapeButton shapeRoundedRectangle: self.
        mask cornerStyle: #rounded.
        self addMorphBack: mask.
        ^ true
    ].
    label comeToFront

width: anInteger
    super width: anInteger.
    self refresh
```

Class Methods

methodsFor: ‘Shapes’

shapeOval: aShapeButton
```
| m |
m _ EllipseMorph new.
m extent = aShapeButton extent.
m position = aShapeButton position.
aShapeButton extent = m extent.
^ m
```

shapeRectangle: aShapeButton
```
| m |
m _ RectangleMorph new.
m extent: aShapeButton extent.
m position: aShapeButton position.
^ m
```

shapeRoundedRectangle: aShapeButton
```
| m |
m _ RectangleMorph new.
m cornerStyle: #rounded.
m extent: aShapeButton extent.
m position: aShapeButton position.
^ m
```

methodsFor: ‘instance creation’

defaultProperties: aShapeButton
```
| s |
s _ aShapeButton.
s normalColor: Color gray.
s normalBorderColor: Color black.
s normalForeColor: Color black.
s normalBorderWidth: 1.
s hoverColor: Color gray.
s hoverBorderWidth: 2.
s hoverForeColor: Color black.
s hoverBorderColor: Color black.
s mouseDownColor: Color black.
s mouseDownForeColor: Color gray.
s mouseDownBorderWidth: 2.
s mouseDownBorderColor: Color white.
s turnOnHoverEffect.
s turnOnMouseDownEffect.
s extent: 80@30
```

Part V – The Making of a ListBox

In this final part we will create ourselves a ListBox control which detects most mouse events through its submorphs but responds to keyboard events. We will cover the cursorPoint method, even though it is not really safe to use in a list box, and we will discover some scrollbar facts.

Although the ListBox is more sophisticated then the ShapeButton, you will be amazed that it takes shorter to write than writing ShapeButton. The reason is simple: we are going to use ShapeButtons inside the ListBox and have them do all the dirty work for us. We put a lot of effort into that ShapeButton afterall. Now it is payback time.

1 – The ScrollBar

Bad news: The built-in scrollbar in Squeak works only with ModelMorph subclasses.

Evidentially, the creators of Squeak had reasons to do it this way. However we may still want to use ScrollBars with other morphic objects as well.

There is an obvious thing to do: we will extend the ScrollBar, and override its method where it actually does the scrolling of a ModelMorph.

```
ScrollBar subclass: #ScrollBarForNonModels
        instanceVariableNames: ‘’
        classVariableNames: ‘’
        poolDictionaries: ‘’
        category: ‘<your category-name here>’
```

And the method to be overridden is:

```
setValue: newValue
“send the value to your owner”
        | v |
        v _ newValue roundTo: scrollDelta.
        owner scrollBarValue: v.
```

If you are curious, here is the original method with comments

```
setValue: newValue
        “Using roundTo: instead of truncateTo: ensures that scrollUp will scroll the
same distance as scrollDown.”
        ^ super setValue: (newValue roundTo: scrollDelta)
```

Ehm, it’s not here. We have to look at the Slider class.

```
setValue: newValue
“Called internally for propagation to model”
```

```
        self value: newValue.
        self use: setValueSelector orMakeModelSelectorFor: 'Value:'
                in: [:sel | setValueSelector _ sel.  Model perform: sel with: value]
```

What??? What does the command in bold mean???

It is a method inherited from ModelMorph class.

use: cachedSelector orMakeModelSelectorFor: selectorBody in: selectorBlock

It is indeed a complicated method. It compiles a new method, with a null return value if the model does not support certain actions, if it supports the actions, it does the desired action.

Anyway, with overriding this method we now have a ScrollBarForNonModels which sends its value to its owner whenever changed. Neat, eh?

2 – Specifications for ListBox

Visual Specifications
- ListBox should <u>highlight its items upon mouseover</u>
- ListBox should <u>give a different color to the selected item</u>
- ListBox' <u>size must be adjustable</u>

Events
- ListBox must be able to be <u>scrolled using arrow keys</u>
- When a <u>key is pressed,</u> ListBox must <u>select the next item starting with the pressed key's value</u>, if such an item exists.
- ListBox must <u>respond to click and double click events</u>

Functionality
- ListBox must be able to <u>contain string and numeric data</u>
- ListBox must be able to <u>contain an additional data for each item</u>
- ListBox must be able to <u>add items as a whole</u>
- ListBox must be able to <u>remove items</u> if necessary

3 – Design Decisions

A list box is simply a collection of items placed in a container. We can think that all rows in the list box is a ShapeButton. Our ShapeButton is versatile enough to act as a row. MoreOver ShapeButton already has hover effect and it handles click and double click events.

We can change the color of ShapeButton as we wish, therefore, re-coloring the selected item is no problem, but we have to store the colors.

We can easily make the box resizable.

We also have to keep another collection for item data, but it is just a collection, it has no visual representation whatsoever.
Adding and removing items to a collection is a piece of cake.

There are some topics about resizing, sorry size in general. The items may not fit the box. Hence we need three variables to store

1. Maximum number of visible rows
2. Top row index as in item collection
3. And for our convenience last row index as in item collection

We are left with keyboard events and we need to figure out a way to select items. Naturally they will be selected when they are clicked, but, how will the list box know which one it is.

Keyboard events are the subject of this chapter, so don't worry about them. For the selection, I say we use the the position of the row. Yes, it is not a good idea, but I want to demonstrate the cursorPoint method.

And which class should we extend? I say we extend BorderedMorph, it will look good.

4 – The ListBox Class

```
BorderedMorph subclass: #ListBox
        instanceVariableNames: 'items itemData selectedItem normalColor
normalForeColor highlightColor selectedColor selectedForeColor lastListIndex
listIndex clickAction doubleClickAction containers maxVisibleItems topItem
lastItem sBar '
        classVariableNames: ''
        poolDictionaries: ''
        category: '<your category-name here>'
```

items: an OrderedCollection of items to be listed

itemData: an OrderedCollection of item data

containers: may be rows were a better name. An OrderedCollection of ShapeButtons.

listIndex: current selected item

lastListIndex: this will let us refresh faster. Rather than removing all submorphs and redrawing them, we will address two ShapeButtons upon selection changes.

5 – The methods of ListBox

Don’t worry; I won’t bother you with all methods anymore. We have just a little left to cover. The methods are provided at the end of chapter and you probably have them installed in Squeak too.

5.1 – Handling keyboard events

We are going to override again. As usual Morph class provides these methods for us, but they are not functional, waiting for us to override.

The first method to override is:

```
hasFocus
        ^ true.
```

If the user can’t focus on an object, that object can’t handle keyboard events. E.g., you can only type in the active window.

The second step is:

```
handlesKeyboard: evt
        ^true
```

Just like the mouse event, isn’t it?

Finally, the last one:

```
□eystroke: event
        | aChar |
        (self scrollByKeyboard: event) ifTrue: [^self].
        aChar _ event keyCharacter.
        ^ self keyPressed: aChar
```

Whenever there is a key stroke while listbox is focused this method will be invoked. But what does the bold line mean?

The bold line tells ListBox to go and check if arrow keys has been pressed. How does it tell that? We are going to write that method too.

If scrollByKeyboard method returns false, mentioning some other key has been pressed, then “keypressed” method will be called.

So, here are the two methods to finalize keyboard event.

```
scrollByKeyboard: event
        (listIndex = nil)
        ifTrue:[listIndex _ 0].
        event keyValue = 30 “up arrow”
```

```
                ifTrue: [
                        listIndex = 1
                        ifTrue:[^true].
                        self listIndex: listIndex – 1.
                        ^ true].
        Event keyValue = 31 "down arrow"
                ifTrue: [
                        listIndex = items size
                        ifTrue:[^true].
                        self listIndex: listIndex + 1.
                        ^ true].
        ^ false
```

```
keyPressed: aChar
        | s |
        s _ listIndex.
        s = 0
        ifTrue:[s _ 1].
        (s to: items size)
        do: [:i |
                (aChar = (((items at: i) at: 1) asCharacter))
                ifTrue:[
                        self listIndex: i.
                        ^ true.
                ].
        ].
```

As you can see, we change the listIndex variable and do nothing. So how come the list gets updated? The listIndex method does that.

```
listIndex: aNumber
        lastListIndex _ listIndex.
        listIndex _ aNumber.
        selectedItem _ items at: aNumber.
        listIndex < topItem
        ifTrue:[
                sBar setValue: (listIndex * (sBar scrollDelta)) asFloat.
                ^ true
        ].
        listIndex > lastItem
        ifTrue:[
                sBar setValue: ((listIndex – maxVisibleItems + 1) * (sBar scrollDelta))
asFloat.
                ^ true
        ].
        self updateList.
```

Whenever the listIndex gets changed it calls the updateList method, which changes the colors of the rows.

The important point here is what happens if the selected item is not in display range. listIndex method considers that too. It sets the value of the scrollbar so that the selected item can be seen.

5.2. Mapping mouse events

Just look at the bold lines, this is a long method.

```
showList
        | b currentTop sD pD l|
        sBar top: self top.
        sBar left: self left + self width – sBar width.
        sBar height: self height.
        containers = nil
        ifTrue:[^ false].
        Items = nil
        ifTrue:[^ false].
        topItem = nil
        ifTrue:[^ false].
        currentTop _ self top + 2.
        L _ lastItem.
        ((items size) < maxVisibleItems)
        ifTrue: [l _ items size].
        (topItem to: l)
        do: [:i |
                ((items at: i) ~= nil)
                ifTrue: [
                        b _ ShapeButton new.
                        b initialize.
                        b shape: 1.
                        b alignment: 1.
                        b label: (items at: i).
                        self addMorph: b.
                        b normalColor: normalColor.
                        b normalBorderColor: normalColor.
                        b normalBorderWidth: 0.
                        b normalForeColor: normalForeColor.
                        b hoverColor: highlightColor.
                        b hoverBorderColor: highlightColor.
                        b hoverBorderWidth: 0.
                        b hoverForeColor: normalForeColor.
                        b height: 14.
                        b width: self width – 2 – sBar width.
                        b top: currentTop.
                        b left: self left + 2.
                        b index: i.
                        b clickAction: [self gotClicked: (self cursorPoint y – self top)].
                        b doubleClickAction: [self gotDoubleClicked].
                        b doubleClickAction: doubleClickAction.
                        b turnOnHoverEffect.
                        b turnOffMouseDownEffect.
                        currentTop _ currentTop + 14.
                        containers add: b
                ].
        ].
        ((items size) > maxVisibleItems)
        ifTrue: [
                sD _ (1 / (items size)) asFloat.
                pD _ ((items size – maxVisibleItems) / (items size)) asFloat.
                sBar scrollDelta: sD pageDelta: pD.
                sBar interval: (1 – pD) asFloat.
                sBar value: (topItem / (items size)) asFloat.
        ]
        ifFalse: [
                sBar scrollDelta: 0.02 pageDelta: 0.2.
                sBar interval: 1.0
        ].
        self updateList
```

As you can see in the bold lines we are setting the ShapeButtons to call ListBox methods upon events.

gotClicked: aNumber

```
        | n |
        n _ (aNumber // 14) + 1.
        (n > (items size))
        ifTrue: [n _ items size].
        self listIndex: (n + topItem – 1).
        clickAction ~= nil
        ifTrue: [clickAction value].
```

gotDoubleClicked

```
        doubleClickAction ~= nil
        ifTrue: [doubleClickAction value].
```

Here we have the method I was mentioning

cursorPoint y

cursorPoint returns a point value of the position of the cursor. As a row has 14 pixels height, we divide the y of the cursorPoint minus ListBox position, to find the index of the row clicked.

I actually added the index property to the ShapeButton specifically for these kind of situations, but I realized I haven't covered cursorPoint, so there was a change of plans.

6 – Cool, now what?

Nothing. I am not going to explain the whole ListBox. You have completed all essential Morph properties, submorphs and almost all events. From now on, everything depends on your design and implementation.

7 – What is next for me to do?

You can still have a look at my “Farewell” and you can examine the Morph reference.

After that, well, the guide is over and I am leaving your life.

8 – Methods of ListBox

methodsFor: ‘accessing’

addItem: aString

```
   items = nil
   ifTrue: [items _ OrderedCollection new].
   itemData = nil
   ifTrue: [itemData _ OrderedCollection new].
   items add: aString asString.
   itemData add: nil.
   self clearList.
   self showList.! !
```

clear

```
   items _ OrderedCollection new.
   itemData _ OrderedCollection new.
   self clearList
```

itemData

```
   ^ itemData
```

itemData: anOrderedCollection

```
   itemData _ anOrderedCollection
```

itemData: anObject at: anIndex

```
   anIndex <= items size
   ifTrue: [
      itemData at: anIndex put: anObject
   ]
```

itemDataAt: anIndex

```
   anIndex <= items size
   ifTrue: [
      ^ itemData at: anIndex
   ]
```

items

```
   ^items
```

items: anOrderedCollection

```
   items _ anOrderedCollection.
   itemData _ OrderedCollection new.
   (1 to: anOrderedCollection size)
   do: [:i |
      itemData add: nil
   ].
```

```
    self clearList.
    self showList.! !
```

listIndex
```
    ^ listIndex
```

listIndex: aNumber
```
    lastListIndex _ listIndex.
    listIndex _ aNumber.
    selectedItem _ items at: aNumber.
    listIndex < topItem
    ifTrue:[
        sBar setValue: (listIndex * (sBar scrollDelta)) asFloat.
        ^ true
    ].
    listIndex > lastItem
    ifTrue:[
        sBar setValue: ((listIndex – maxVisibleItems + 1) * (sBar scrollDelta)) asFloat.
        ^ true
    ].
    self updateList
```

removeItem: anIndex
```
    items = nil
    ifTrue: [^ false].
    anIndex <= items size
    ifTrue:[
        items removeAt: anIndex.
        self removeItemDataAt: anIndex.
    ].
    self clearList.
    self showList
```

removeItemDataAt: anIndex
```
    itemData = nil
    ifTrue:[^ false].
    anIndex <= items size
    ifTrue: [
        ^ itemData removeAt: anIndex
    ]
```

selectedItem
```
    ^ selectedItem
```

selectedItem: aNumber
```
    ^ self listIndex: aNumber
```

methodsFor: ‘event handling’

handlesKeyboard: evt
```
    ^true
```

hasFocus
```
    ^ true
```

□eystroke: event
```
| aChar |
(self scrollByKeyboard: event) ifTrue: [^self].
aChar _ event keyCharacter.
^ self keyPressed: aChar
```

scrollByKeyboard: event
```
(listIndex = nil)
ifTrue:[listIndex _ 0].
event keyValue = 30
    ifTrue: [
        listIndex = 1
        ifTrue:[^true].
        self listIndex: listIndex – 1.
        ^ true].
event keyValue = 31
    ifTrue: [
        listIndex = items size
        ifTrue:[^true].
        self listIndex: listIndex + 1.
        ^ true].
^ false
```

methodsFor: 'events-processing'

clickAction: aBloc
```
clickAction _ aBlock
```

doubleClickAction: aBlock
```
| b |
doubleClickAction _ aBlock.
containers ~= nil
ifTrue: [
    (1 to: containers size)
    do: [:i |
        b _ containers at: i.
        b doubleClickAction: aBlock.
    ].
]
```

gotClicked: aNumber
```
| n |
n _ (aNumber // 14) + 1.
(n > (items size))
ifTrue: [n _ items size].
self listIndex: (n + topItem – 1).
clickAction ~= nil
ifTrue: [clickAction value]
```

gotDoubleClicked
```
doubleClickAction ~= nil
ifTrue: [doubleClickAction value]
```

```
keyPressed: aChar
    | s |
    s _ listIndex.
    s = 0
    ifTrue:[s _ 1].
    (s to: items size)
    do: [:i |
        (aChar = (((items at: i) at: 1) asCharacter))
        ifTrue:[
            self listIndex: i.
            ^ true.
        ].
    ]

methodsFor: 'initialization'

initialize
    super initialize.
    sBar _ ScrollBarForNonModels new.
    self addMorph: sBar.
    ListBox defaultProperties: self.
    maxVisibleItems _ (self height // 14).
    topItem _ 1.
    lastItem _ maxVisibleItems.
    self color: normalColor.
    containers _ OrderedCollection new! !

hideOrShowScrollBar
    ^true! !

scrollBarValue: scrollValue
    topItem _ (scrollValue * items size).
    topItem < 1
    ifTrue: [
        topItem _ 1.
        sBar value: sBar scrollDelta.
    ].
    lastItem _ topItem + maxVisibleItems – 1.
    (lastItem > (items size))
    ifTrue: [
        lastItem _ items size.
        topItem _ lastItem – maxVisibleItems + 1.
    ].
    self clearList.
    self showList


methodsFor: 'visual properties'

extent: aPoint
    super extent: aPoint.
    maxVisibleItems _ (self height // 2) + 1.
    lastItem ~= nil
    ifTrue: [lastItem _ topItem + maxVisibleItems].
    self clearList.
    self showList
```

height: aNumber

```
    super height: aNumber.
    maxVisibleItems _ (self height // 2) + 1.
    lastItem ~= nil
    ifTrue: [lastItem _ topItem + maxVisibleItems].
    self clearList.
    self showList
```

highlightColor

```
    ^highlightColor
```

highlightColor: aColor

```
    highlightColor _ aColor
```

normalColor

```
    ^normalColor
```

normalColor: aColor

```
    normalColor _ aColor
```

normalForeColor

```
    ^normalForeColor
```

normalForeColor: aColor

```
    normalForeColor _ aColor
```

selectedColor

```
    ^selectedColor
```

selectedColor: aColor

```
    selectedColor _ aColor
```

selectedForeColor

```
    ^selectedForeColor
```

selectedForeColor: aColor

```
    selectedForeColor _ aColor
```

width: aNumber

```
    super width: aNumber.
    self clearList.
    self showList
```

methodsFor: ‘private’

clearList

```
    | b |
    containers = nil
    ifTrue: [^ false].
    ((containers size) > 0)
    ifTrue: [
        (1 to: containers size)
        do: [:i |
            b _ containers at: i.
            b delete.
        ].
    ].
    containers _ OrderedCollection new.! !
```

showList

```
    | b currentTop sD pD l|
    sBar top: self top.
    sBar left: self left + self width – sBar width.
    sBar height: self height.
    containers = nil
    ifTrue:[^ false].
    items = nil
    ifTrue:[^ false].
    topItem = nil
    ifTrue:[^ false].
    currentTop _ self top + 2.
    l _ lastItem.
    ((items size) < maxVisibleItems)
    ifTrue: [l _ items size].
    (topItem to: l)
    do: [:i |
        ((items at: i) ~= nil)
        ifTrue: [
            b _ ShapeButton new.
            b initialize.
            b shape: 1.
            b alignment: 1.
            b label: (items at: i).
            self addMorph: b.
            b normalColor: normalColor.
            b normalBorderColor: normalColor.
            b normalBorderWidth: 0.
            b normalForeColor: normalForeColor.
            b hoverColor: highlightColor.
            b hoverBorderColor: highlightColor.
            b hoverBorderWidth: 0.
            b hoverForeColor: normalForeColor.
            b height: 14.
            b width: self width – 2 – sBar width.
            b top: currentTop.
            b left: self left + 2.
            b index: i.
            b clickAction: [self gotClicked: (self cursorPoint y – self top)].
            b doubleClickAction: [self gotDoubleClicked].
            b doubleClickAction: doubleClickAction.
            b turnOnHoverEffect.
            b turnOffMouseDownEffect.
            currentTop _ currentTop + 14.
            containers add: b
        ].
    ].
    ((items size) > maxVisibleItems)
    ifTrue: [
        sD _ (1 / (items size)) asFloat.
        pD _ ((items size – maxVisibleItems) / (items size)) asFloat.
        sBar scrollDelta: sD pageDelta: pD.
        sBar interval: (1 – pD) asFloat.
        sBar value: (topItem / (items size)) asFloat.
    ]
    ifFalse: [
        sBar scrollDelta: 0.02 pageDelta: 0.2.
        sBar interval: 1.0
    ].
    self updateList
```

updateList

```
    | b |
    listIndex = nil
        ifTrue: [^ false].
    listIndex > (maxVisibleItems + topItem – 1)
        ifTrue: [^ false].
    listIndex < topItem
        ifTrue: [^ false].
    (lastListIndex ~= nil
            and: [lastListIndex >= topItem
                    and: [lastListIndex <= (lastItem)]])
        ifTrue: [b _ containers at: (lastListIndex – topItem + 1).
            b normalColor: normalColor.
            b normalBorderColor: normalColor.
            b normalForeColor: normalForeColor.
            b turnOnHoverEffect].
    b _ containers at: listIndex – topItem + 1.
    b normalColor: selectedColor.
    b normalBorderColor: selectedColor.
    b normalForeColor: selectedForeColor.
    b turnOffHoverEffect
```

class methodsFor: 'instance creation'

defaultProperties: aListBox

```
    aListBox borderColor: Color lightGray.
    aListBox normalColor: Color white.
    aListBox highlightColor: Color paleBlue.
    aListBox selectedColor: Color blue.
    aListBox normalForeColor: Color black.
    aListBox selectedForeColor: Color white.
    aListBox extent: 100@200.
```

Farewell

I hope you enjoyed this tutorial, as much as I enjoyed writing it. I think here I must say that I enjoyed creating this document a lot.

What I really wish is that this document has been useful to you. Smalltalk stems from the idea of having everything as an object. As all approaches, this has advantages, as well as disadvantages.

Object oriented programming is a smart way to follow, but it brings problems about usability along. If we have 1000 objects in the system and each object introduces 10 new methods, it makes 11000 words for a user to handle, which is almost as difficult as learning a foreign language. So many objects and methods take too much time even to scan, no need to mention deciding what to use.

Though this guide aims to demonstrate morphic classes and their uses, I tried to show how to design and create objects as well. Evidentially every programmer has an own style, but knowing other styles wouldn't hurt.

Thank you for reading the tutorial and

Farewell.

MORPH CLASS REFERENCE

Object subclass Morph

Common instance methods in alphabetical order

addAllMorphs: aCollection
adds all morphs in the collection as submorphs.

addAllMorphs: aCollection after: anotherMorph
adds all morphs in the collection as submorphs after the existing submorph, keeping the z-order.

addMorph: aMorph
adds a morph as a submorph to the z-order 1.

addMorph: newMorph after: aMorph
adds the new morph after the existing submorph aMorph regarding the z-order.

addMorph: newMorph behind: aMorph
adds the new morph after the existing submorph aMorph regarding the z-order.

addMorph: newMorph inFrontOf: aMorph
adds the new morph before the existing submorph aMorph regarding the z-order.

addMorphBack: aMorph
adds a morph as a submorph with lowest z-order.

addMorphCentered: aMorph
adds a morph as a submorph and centers it inside itself.

addMorphFront: aMorph
adds a morph as a submorph with highest z-order.

beTransparent
makes the morph transparent.

bottom: aNumber
moves the morph so that its bottom is at aNumber.

bottomLeft: aPoint
moves the morph so that its bottom left corner is on aPoint.

bottomRight: aPoint
moves the morph so that its bottom right corner is on aPoint.

center
centers the morph in the world that it belongs.

click: evt
default method does nothing, must be overridden to provide action. Becomes invoked upon mouse click.

collapse
minimizes the morph.

color: aColor
changes the color of a morph

comeToFront
sets the z-order of the morph to 1.

copy
returns a copy of the morph without its submorphs.

cursorPoint
returns a point indicating where the mouse cursor is.

delete
deletes the morph as a submorph

doubleClick: evt
default method does nothing, must be overridden to provide action. Becomes invoked upon mouse double click.

dragEnabled
returns a boolean value indicating if submorphs can be added by drag&drop the the morph.

dragEnabled: aBool
sets dragEnabled property.

duplicate
returns a clone of the morph.

extent: aPoint
resizes the Morph

goBehind
sets the morph's z-order to the lowest.

handlesKeyboard: evt
default method returns false, must be overridden to provide action. Must return true if keyboard events will be handled by the morph. Related methods: hasFocus, keyDown, keyStroke, keyUp.

handlesMouseDown: evt
default method returns false, must be overridden to provide action. Must return true if mouse down events will be handled by the morph. Related methods: click, doubleClick, mouseDown, mouseUp.

handlesMouseOver: evt
default method returns false, must be overridden to provide action. Must return true if mouse over events will be handled by the morph. Related methods: mouseEnter, mouseLeave.

handlesMouseOverDragging: evt
default method returns false, must be overridden to provide action. Must return true if drag events will be handled by the morph.

handlesMouseStillDown: evt

default method returns false, must be overridden to provide action. Must return true if mouse pressed events will be handled by the morph.

hasFocus
default method returns false, must be overridden to provide action. Must return true if keyboard events will be handled by the morph. Related methods: handlesKeyboard, keyDown, keyStroke, keyUp.

height: aNumber
resizes the morph so that it has the height aNumber

initialize
initializes the morph.

isStepping
returns whether the morph timer is on or off.

keyDown: anEvent
default method does nothing, must be overridden to provide action. Becomes invoked upon a key is pressed. Related methods: handlesKeyboard, hasFocus.

keyStroke: anEvent
default method does nothing, must be overridden to provide action. Becomes invoked upon a keystroke. Related methods: handlesKeyboard, hasFocus.

keyUp: anEvent
default method does nothing, must be overridden to provide action. Becomes invoked upon a key is released. Related methods: handlesKeyboard, hasFocus.

left: aNumber
sets the left of the morph to the given number.

mouseDown: evt
default method does nothing, must be overridden to provide action. Becomes invoked when mouse is pressed. Related methods: handlesMouseDown.

mouseEnter: evt
default method does nothing, must be overridden to provide action. Becomes invoked when mouse enters morph's bounds. Related methods: handlesMouseOver.

mouseEnterDragging: evt
default method does nothing, must be overridden to provide action. Becomes invoked when mouse enters morph's bounds dragging an object. Related methods: handlesMouseOverDragging.

mouseLeave: evt
default method does nothing, must be overridden to provide action. Becomes invoked when mouse leaves morph's bounds. Related methods: handlesMouseOver.

mouseLeaveDragging: evt
default method does nothing, must be overridden to provide action. Becomes invoked when mouse leaves morph's bounds dragging an object. Related methods: handlesMouseOverDragging.

mouseMove: evt

default method does nothing, must be overridden to provide action. Becomes invoked upon mouse movement.

mouseStillDown: evt
default method does nothing, must be overridden to provide action. Becomes invoked if mouse is pressed and hold. Related methods: handlesMouseStillDown.

mouseUp: evt
default method does nothing, must be overridden to provide action. Becomes invoked when mouse is released. Related methods: handlesMouseDown.

openInWindowLabeled: aString
opens morph in the window with the given label.

openInWorld
opens morph in the default world.

openInWorld: aWorld
opens the morph in a desired world.

owner
returns the owner of the morph.

position
returns a point indicating the top left corner of the morph.

position: aPoint
sets top left corner of the morph to a point.

removeAllMorphs
removes all submorphs of the morph.

right: aNumber
moves the morph so that its right is at a number.

showBalloon: msgText
shows balloon containing msgText.

startStepping
starts the timer. Related methods: step, stepAt, stepTime, stop.

step
default method does nothing, must be overridden to provide action. This method is invoked periodically. Related methods: startStepping, stepAt, stepTime, stop.

stepAt: amillisecondValue
invoke the step method in amillisecondValue. Related methods: startStepping, step, stepTime, stop.

stepTime
returns how frequent the step method will be invoked. Related methods: startStepping, step, stepAt, stop.

stop

stops timer. Related methods: startStepping, step, stepAt, stepTime.

submorphs
returns a copy of the submorphs collection.

top: aNumber
moves the morph so that its top is at aNumber.

topLeft: aPoint
moves the morph so that its top left corner is at aPoint.

topRight: aPoint
moves the morph so that its top right corner is at aPoint.

visible: aBoolean
sets whether the morph is visible or not.

width: aNumber
resizes the morph so that it has the width aNumber.