Chapter 1

Basic Widgets

This chapter is about the basic window gadgets (aka widgets) used to compose graphical user interfaces in Pharo using Morphic. Since Morphic is based on a composite pattern, you can easily assemble different widgets to build complex interfaces. We will explain how the essential widgets work and how to assemble them together. To help you understand the mechanisms behind Morphic, a single example will be used through the whole chapter.

1.1 The window

Jannik ►*explain the basis of Widget ? What is a widget in pharo ? Which classes should I know ?* ◄

Opening a window

The basic class for managing a window is StandardWindow. Let's start with an empty one.

To create and open an empty window just try the following:

```
StandardWindow new openInWorld
```

You should see a window with a top bar Jannik ►*title bar ?*◄ that you can move with the mouse (see Figure **??**). The text in the top bar is referred to as the window label.

On one side, the topbar has three buttons for closing, collapsing and expanding the window. On the other side, the topbar has a menu button with a default set of items:

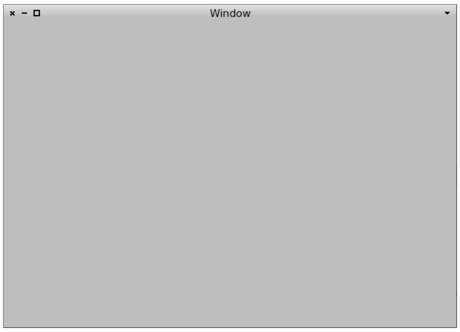


Figure 1.1: An empty window

- Close, which closes the window
- About, which displays the About window
- Change title, which allows the user to set their own title for this window
- Send to back, which puts this window behind all the others on the screen
- Make next-to-topmost, which puts this window behind the window immediately behind it
- Make closable/unclosable, which changes whether the window can be closed
- Make draggable/undraggable, which controls whether or not the window is fixed in place
- Maximize, which makes the window take up most of the screen
- and Window color, which controls the default background color

These default menu items are added by SystemWindow>>buildWindowMenu, and in a moment we will learn how to extend this menu.

A window and its model

One of the first problems one runs into in developing graphical applications is the coupling of user interface code with application code. Proper object-oriented design promotes a separation of concerns: parts of the application that deal with the business logic should not also worry too much about what color font their text is rendered in. Every program with a UI must address this problem.

The early days of Smalltalk produced a novel solution to the problem of separating UI code from application code in the form of the Model-View-Controller design pattern. The idea is very simple:

- *Models* represent the problem domain and the intelligence of the application
- *Views* handle the visual representation of the model
- *Controllers* manage the interaction between the view and the model

We've come a long way since then, and Morphic simplifies the relationship, mostly removing the need for the controller **Daniel** ►*I have no idea if this is really true*◄. In Morphic, we instead deal with *pluggable views*: views which manage themselves and delegate high-level responsibilities to a model that can be plugged into them. By specifying a model for a window, and implementing specific methods, the model will be able to control some of the window's behavior.

It's easy to set a model on a window:

```
StandardWindow new model: myModel
```

Example:

First, let's create the model class:

Class 1.1: *Defining a specific Model.*

```
Object subclass: #MyModel
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'PBE2−Examples'

MyModel>>#initialExtent

   ^ 200@200
```

Let's see the result:

```
StandardWindow new openInWorld.
StandardWindow new model: (MyModel new); openInWorld.
```

You see a window with the same size as the previous one, and a small window whose size is exactly what you specified in the method initialExtent (see Figure **??**).

Stéf ►*should we always specifies a model?*◄ **Benjamin** ►*If you want to specify the initial extent, either you define a model or you subclass (as far as I know)*◄ **Stéf** ►*what should be put in the model vs. the Morph itself?*◄ **Benjamin** ►*I think/hope it will be clear when the API section will be complete*◄ **Jannik** ►*if you are sure about your answers, just include them in the real text.*◄

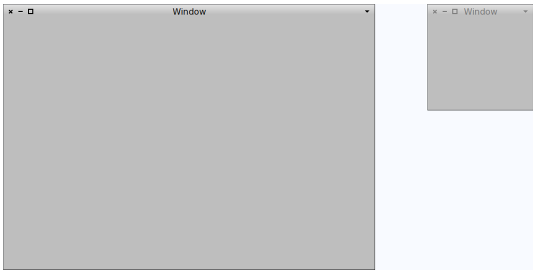


Figure 1.2: Two windows: one without a model and one with a model

Benjamin ►*move that into Model API part*◄

You can also control whether or not the window can be closed by implementing okToChange:

Method 1.2: *Define if the model is ok to change or not*

```
MyModel>>#okToChange

   ^ false
```

Now the window doesn't close when you try to close it, because the model has explicitly stated that the widget is not OK to be changed.

Your own topbar menu

You can easily add shortcuts to the window's menu. Simply override the method addModelItemsToWindowMenu: aMenu and fill up the provided menu. For example:

Alain ►*addModelItemsToWindowMenu: is in Object !!*◄

Method 1.3:

```
MyModel>>#addModelItemsToWindowMenu: aMenu
   "Add model−related items to the window menu"
```

```
    "First, we add a separator"
    aMenu addLine.

    "Then, we add our items"
    aMenu
        add: 'Label of the entry'
        target: receiverOfTheFollowingSelector
        action: #selectorWeWantToBeExecuted.
```

Example:

Method 1.4: *Define the extra entries of the menu*

```
MyModel>>#addModelItemsToWindowMenu: aMenu
    "Add model−related items to the window menu"
    super addModelItemsToWindowMenu: aMenu.
    aMenu addLine.
    aMenu
        add: 'Open an inspector on me'
        target: self
        action: #inspect.
```

When you click on the menu button, you see the new label at the end of the list (see Figure **??**) and when you click it, an inspector is opened (see Figure **??**.)

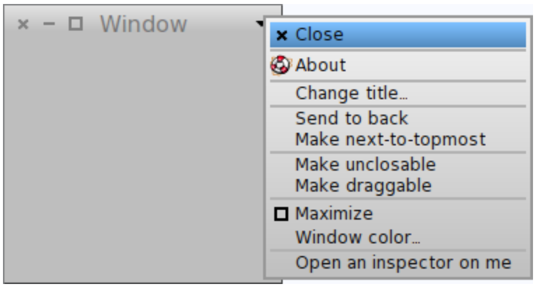


Figure 1.3: Menu with our extra item at the end

Main Window API

Jannik ►*some blahblah*◄ Here is the list of the main API for a window:

- Title

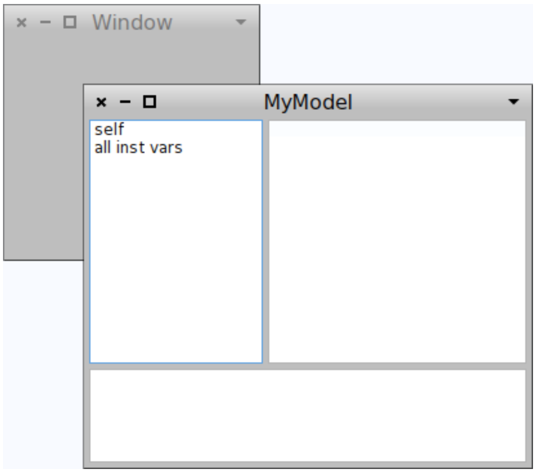


Figure 1.4: An inspector is open

- Color
- roundcorners?
- minimumExtent: **Daniel** ▶*these two methods exist but seem to do nothing*◀
- maximumExtent:

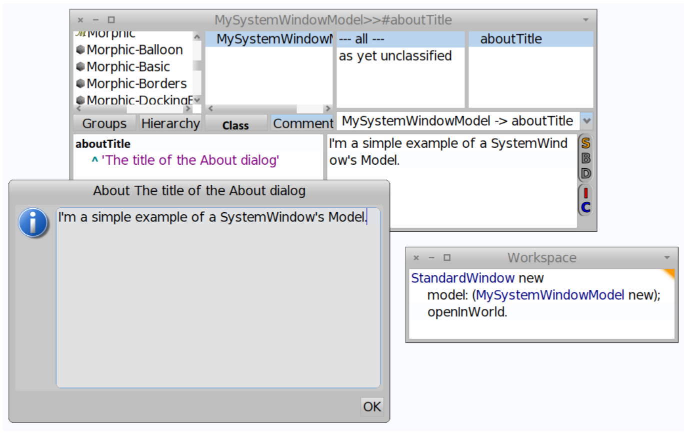
  ```
  SystemWindow new
     maximumExtent: 200@100; openInWorld

  pareil pour

  StandardWindow new
     unexpandedFrame: (0@0 extent: 200@100) ; openInWorld
  ```

  je ne comprends pas pourquoi je peux alors avoir une fenetre immense? **Jannik** ▶*what ?*◀
- unresizeable? **Jannik** ▶*what ?*◀
- Action on close? **Jannik** ▶*what ?*◀

About dialog. The SystemWindow will generate an About window for you. You can control the title by implementing aboutTitle and it will take the body of the class comment on your model as the text of the about dialog. For example:



```
aboutTitle
	^ 'The title of the About dialog'
```

I'm a simple example of a SystemWindow's Model.

```
StandardWindow new
	model: (MySystemWindowModel new);
	openInWorld.
```

Figure 1.5: Customizing the About dialog

Stepping. Graphical animations, simulations and other processes that need to be able to update the user interface continuously and frequently can make use of the stepping system on their model. To participate, your model must implement wantsStepIn: aSystemWindow to indicate that you want stepping and stepTimeIn: aMillisecondTimeStamp to indicate how frequently you want them. For example:

Method 1.5:

```
MyModel>>#wantsStepsIn: aSystemWindow
   ^true

MyModel>>#stepTimeIn: aSystemWindow
   ^100
```

This indicates I want 10 steps a second (every 100 milliseconds.) You can prove to yourself that it works with the Transcript:

Method 1.6:

```
MyModel>>#stepAt: aMillisecondTime in: aSystemWindow
   Transcript crShow: aMillisecondTime
```

Other SystemWindow model delegate messages:

- #initialExtent
- #modelWakeUpIn: aSystemWindow—sent when the window becomes active

- #modelSleep—sent when the window becomes inactive
- #windowIsClosing—sent during window deletion, before model is cleared
- #desiredWindowLabelHeightIn: aSystemWindow—sent during label creation
- #windowReqNewLabel: aString—sent by relabel
- #taskbarIcon
- #taskbarLabel

Additional StandardWindow model delegate messages:

- #okToClose

Title. title: The default title of the window is simply 'Window'. You can change this title by sending title: to a window with a String as argument:

```
w := StandardWindow new title: 'My first window'.
w openInWorld
```

Position and size. In order to change the position and the size of a window, you can send the messages extent: and position: to a newly opened window.

```
w := StandardWindow new title: 'My first window'.
w openInWorld.
w extent: 100@50.
w position: 0@0
```

Now try the following: first create a window but without opening it; then set its size and its position and then open it by sending openInWorld *to it.*

```
w := StandardWindow new title: 'My first window'.
w extent: 100@50.
w position: 0@0
w openInWorld.
```

You should observe that the size and the position are not set with your values. Don't worry, you did nothing wrong, its a normal behavior regarding the way window opening is implemented in Morphic. In fact you have to know that a window internal state is overwritten with default values when it is opened. This is one of the responsibilities of RealEstateAgent.

1.2 Buttons

Push button

Button is the simplest common GUI element. Button shows information what it will do by showing texts and images. Button sends message when clicked by the user. The simplest button is a button that shows nothing nor does anything.

```
b := PluggableButtonMorph new openInWorld.
b bounds: (100@100 extent: 40@30).
```

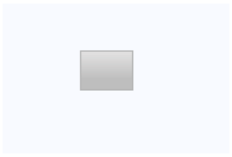


Figure 1.6: simple button

If you click this button, you will notice the button does nothing. That's becuase we haven't told it what to do. We can set up the button to send a message to an object when it is clicked. The object who will receive a message is called "target" sometimes "model". For now it is called "model". The selector of the message is called "action". Let's set up the model and action, so the button itself be deleted when it is clicked.

```
b model: b.
b action: #delete.
```

click the button

OK. The button is deleted. Since, block is an object and block is a chunk of operation that can be evaluated later. We can make a button do more complicated operations by setting model with a block and action with #value .

```
b := PluggableButtonMorph new openInWorld.
b bounds: (100@100 extent: 40@30).
b model: [ self inform: 'you pressed me' ].
b action: #value.
```

To give our user some hint what the button will do, it will be nice put a label on the button.

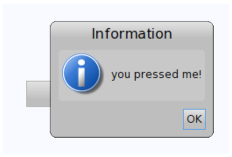


Figure 1.7: simple button with block

Figure 1.8: simple button with label

```
b label: 'press me'.
```

Let's delete the button and move on to the next subject.

```
b delete.
```

Last button we played with had a label. But maybe our model for the button changed its state by pressing the button or by other events. User must be informed with correct information. Button has a getLabelSelector which is a selector used for asking the model what the new label should it be. To use getLabelSelector, we must send a message #on:getState:action:label:menu:, which is a very typical form of method used in Morphic. It sets a bunch of requried selectors with its model. Put your model for keyword on: and your button action for keyword action: and getLabelSelector for keyword #label:.

Evaluate the code below in your workspace. You will see a button with label showing an OrderedCollection which is the model for the button.

```
m := OrderedCollection withAll: {1. 2. 3. 4. 5}.

b := PluggableButtonMorph new openInWorld.
b addDependent: m.
b bounds: (100@100 extent: 40@30).
b on: m getState: nil action: #myButtonAction label: #printString menu: nil.
```

Since we assigned the getLabelSelector with #printString, label of the button will be the result of #printString on m.

If you press the button, you will get a #dontUnderstandMessage: error. That is because OrderedCollection does not have method myButtonAction. Define one as code below.

```
myButtonAction

   self removeFirst.
   self changed: #printString.
```

Now if you press the button you will see the the m is being #removeFirst and its label is updated. Button does not update the label of itself even though the getLabelSelector is set. It need to be asked the update itself. To do that #changed: in the #myButtonAction method and #addDependent: is used. To learn more about dependency system read section .... **HwaJong** ►*is there one?*◄

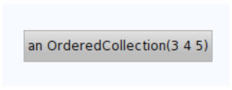

an OrderedCollection(3 4 5)

Figure 1.9: self updating button

Since the size of the collection is 5, more than five clicks, it will raise error. To avoid this, we need make it unpressable when collection gets empty. To do this, it is time to use #getEnabledSelector. Set a selector that will be used for asking the model whether the button may be enabled.

```
b getEnabledSelector: #notEmpty.
```

To trigger the enabling update, add another #changed: send inside the #myButtonAction method.

```
myButtonAction

   self removeFirst.
   self changed: #printString.
   self changed: #notEmpty
```

click the button more than 5 time

Checkbox

Checkbox is a button that can show on and off state, and toggles it when clicked. Filled box means on state, empty box means off state. In Pharo, checkbox can be made with ThreePhaseButtonMorph or CheckboxButtonMorph.

```
c := CheckboxButtonMorph new.
c openInWorld.
```

Send #selected: message with boolean object to change its state.

```
c selected: true.  "turn on"
c selected: false. "turn off"
```

If target, and actionSelector is set(setting arguments is an option), checkbox can send message to its target.

```
m := Morph new.
m openInWorld.

c :=  CheckboxButtonMorph new.
c openInWorld.
c target: m.
c actionSelector: #toggleCornerRounding.
```

Radio button

Radio button in Pharo is sharing same classes with checkbox. Only difference is that it draws a circle than square box if it is sent a message beRadioButton. It does not implement any behavior(logic) of radio button, so it is our responsibility to write a good radio button model object that makes sure only one button is turned on, and at least one button is turned on.

Pharo has a model class that is for demonstrating radio buttons, named ExampleRadioButtonModel.

```
radioModel := ExampleRadioButtonModel new.
r1 := (CheckboxMorph on: radioModel selected: #isLeft changeSelected: #beLeft)
       label: 'left'.
r2 := (CheckboxMorph on: radioModel selected: #isCenter changeSelected: #beCenter)
       label: 'center'.
r3 := (CheckboxMorph on: radioModel selected: #isRight changeSelected: #beRight)
       label: 'right'.
r1 openInWorld.
r2 openInWorld.
r3 openInWorld.
r1 bounds: (440@10 extent: 80@30).
r2 bounds: (440@40 extent: 80@30).
r3 bounds: (440@70 extent: 80@30).
```

As explained earlier, we are using the same class CheckboxMorph, but it works perfectly as radio buttons. Radio buttons are asking the model whether they should turn themselves on with selectors #isLeft, #isCenter,

#isRight. And when the user press the radio button it send message like #beLeft, #beCenter, beRight to the model. Then the model sends message back to the buttons so the they turned on and off property.

Following codes will change their appearance, too.

```
r1 beRadioButton.
r2 beRadioButton.
r3 beRadioButton.
2 timesRepeat: [
   r1 toggleEnabled.
   r2 toggleEnabled.
   r3 toggleEnabled.
].
\hjo{Hmm... Hope there is easier why to do this...}
```

Delete morphs after you are finished using them.

```
r1 delete.
r2 delete.
r3 delete.
```

1.3 Text fields

get text

set text

selection

1.4 Text editor

1.5 Panes and layout managing

newRow:

newColumn:

newGroup:

1.6 List widgets

getting list

getting selections

setting selections

HwaJong ►*outlining with subsections.*◄

1.7 Tree widgets

getting list

getting selections

setting selections

collapsing and expanding

1.8 Layout management

Benjamin ►*Maybe we should write a section about layouts to describe their API ?*◄ **Stéf** ►*yes excellent idea. Do you know that luarent and hilaire wrote some text* ◄ **Benjamin** ►*in fact, only TableLayout have a specific API*◄

A morph is basically a composite. As such it can be composed of sub-morphs. As it is explained below, adding a morph into a parent is very simple: just use the addMorph: message which is sent to a parent morph with a sub-morph passed as argument. The problem is to understand how a morph can manage its own layout according to its size and the morphs that it contains. There are two main possibilities: (1) a morph can be added to another morph with a fixed position or (2) the placement of a morph is automatically managed with regards to its parent morph.

Of course, the second solution is predominantly chosen. In this case, a layout manager is to be used to manage the space of a morph. But one have to decide if the placement is managed from the parent-morph or from the sub-morph point of view:

top-down placement: the placement is managed from the point of view of the parent morph which makes use of a layout manager that decides how to place the sub-morphs;

bottom-up placement: the placement is managed locally from the point of view of each sub-morph, the layout manager decides how to place a sub-morph according the parent extent.

Historically, these two possibilities are respectively provided by the TableLayout and by the ProportionalLayout classes. As it is explained in this chapter, each of these layout manager has its own variants and now a lot of possibilities are offered by the morphic framework.

Composing a morph

To add a morph into a parent morph, just make use of the addMorph: message. Together with the use of the extent: and of the position: messages, it is straightforward to compose a morph.

Declare a morph with a small extent and its parent with a bigger extent, then assign a color to the parent (different from blue), set the position of the parent, add the first morph to the parent and open the parent morph

As an example, you can try with the following code:

```
sub := Morph new extent: 30 @ 30. "the sub−morph with a small extent"
parent := Morph new extent: 100 @ 80. "the parent morph with a bigger extent"
parent position: 10 @ 10; color: Color orange. "set the parent position and color"
parent addMorph: sub. "add the sub−morph"
parent openInWorld "open the parent morph"
```

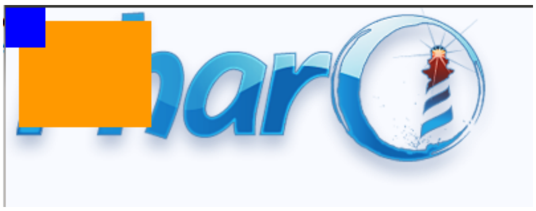


Figure 1.10: Composing a morph: a first try

The result is shown in Figure **??**. You may be surprised because the sub-morph is drawn outside it's parent. There are two reasons for this:

- first, you must recall that a particularity of the morph system is to consider a morph's position as an absolute one. Here, the sub-morph position is 0@0 because it has not been explicitly changed. It implies that the sub-morph is drawn starting at the top left of the world.
- second, the parent position has been set **before** the sub-morph has been added to it.

To be convinced, try the same experiment but this time, set the parent morph before adding the sub-morph

```
sub := Morph new extent: 30 @ 30.
parent := Morph new extent: 100 @ 80.
parent addMorph: sub. "add the sub-morph"
parent position: 10 @ 10; color: Color orange. "set the parent position after"
parent openInWorld
```

The result of this second try is shown in Figure **??**.

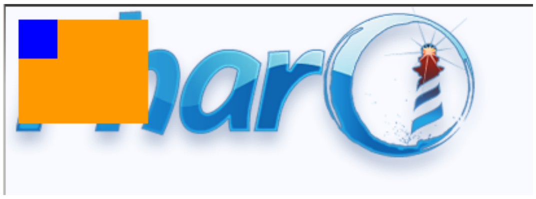


Figure 1.11: Composing a morph: a second try

For this second solution, the sub-morph is well physically inside its parent.

Move the parent morph with the mouse or by sending to it the message position: *with a point as argument. Do this for the two solutions*

You can observe that the sub-morph is also moved whatever the solution you consider. If you inspect the sub-morph after having moved its parent you can see that its position has changed accordingly. So we can say that the sub-morph is logically in its parent but this doesn't mean that it is physically inside its parent. The reason why the second solution sub-morph is drawn inside its parent is only because the parent position has been changed after the sub-morph adding.

The consequence is that if you want a submorph to be placed inside the area occupied by its parent then you have to explicitly calculate its position according to the parent position.

Open the morph halo on the parent morph, then change the parent extent by clicking on the bottom right yellow button and moving the mouse around with the mouse button down. Do the same but with the sub-morph

You observe that, if you change the parent morph extent, then the sub-morph extent stays unchanged and that you can make the parent morph smaller that its sub-morph. The same thing occurs if you change the sub-morph extent. It means that sub-morph and parent morph extents are managed independently.

So, if you want a morph extent to be changed according to a particular rule, for example according to a parent or a sub-morph extent change, then you have to implement it.

To exercise yourself, try to implement the following: a red morph containing one row with three adjacent white sub-morphs, each containing itself a StringMorph *showing its position. see Figure* **??**

.

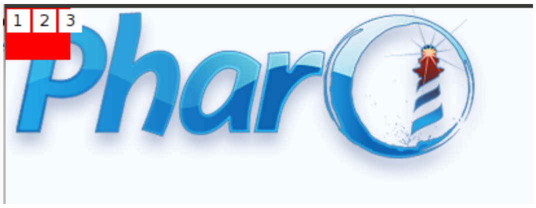


Figure 1.12: Composing a morph with a row of sub-morphs

Here is a solution:

```
parent := Morph new color: Color red.
1 to: 3
 do: [:i |  | sub |
   sub := Morph new color: Color white; borderWidth: 1; extent: 20 @ 20; yourself.
   sub addMorphCentered: (StringMorph contents: i asString).
   parent addMorph: sub.
   sub position: (Point x: i − 1 * sub width y: 0)].
parent openInWorld
```

Now, imagine that you want the parent morph extent to exactly fit its row extent, that you want to be able to add a sub-morph and have the parent morph extent updated accordingly. It is clear that one can't implement it like that. But don't be despaired of this, don't give up!. Of course, there are many way do automatically manage morph positions and area extents as it is explained in next sections.

With the default layout into a window

This section explains how to add morph into a window with the default layout. Alain ►*Se serait mieux d'utiliser un parent Morph plutot qu'un Window*◄ The default layout is ProportionalLayout. To use it, you have to use the method addMorph: aMorph frame: aFrame

```
aWindow
   addMorph: morphToAdd
   frame: (x0@y0 corner: x1@y1)
```

where x0, y0, x1 and y1 are defined as shown in Figure **??**. Note that their values are floats between 0 and 1.

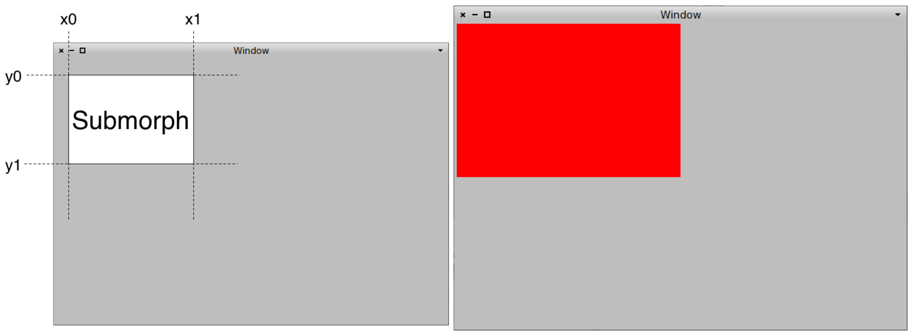


Figure 1.13: An example with a frame 0@0 corner: 0.5@0.5.

Some Morphs

First, let's define some morphs to illustrate and experiment with.

```
| container redMorph blueMorph greenMorph |
redMorph := Morph new color: Color red; yourself.
blueMorph := Morph new color: Color blue; yourself.
greenMorph := Morph new  color: Color green; yourself.
container := PanelMorph new.
```

We will not repeat their definition in the future except in the first code snippet so that you can get its full definition.

A first configuration. The following snippet of code asks the red morph to occupy all the space of its container.

```
| window |
window := SystemWindow new.
redMorph := Morph new color: Color red; yourself.
window
   addMorph: redMorph
   frame: (0@0 corner: 1@1).

redMorph color: Color red.
window openInWorld
```

Note: you have to reset the color of the morph after having added it because the window set the default color. Here it seems strange but for more complicated morphs (like buttons, list ...) it sets the background color for a better integration.

As a result, you can see that the red morph is stretched the fill the space both vertically and horizontally (see Figure **??**).

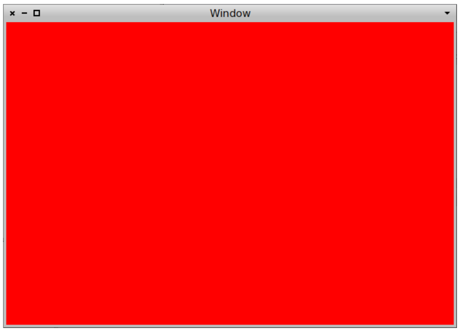


Figure 1.14: The red morph fill the whole space.

A little more complicated configuration. Now we add three morphs of different colors.

```
window
   addMorph: redMorph
   frame: (0@0 corner: 0.33@1).
window
   addMorph: blueMorph
   frame: (0.33@0 corner: 0.66@1).
window
   addMorph: greenMorph
   frame: (0.66@0 corner: 1@1).
```

As a result, you can see three stripes of color where each is horizontally a third of the window size and fills the space vertically (see Figure **??**).

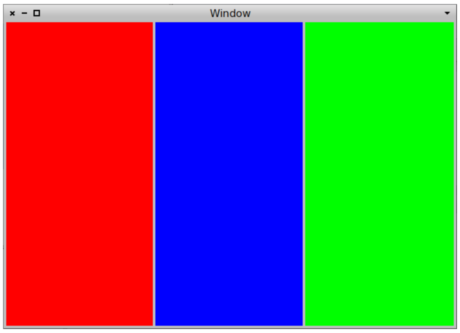


Figure 1.15: Three color stripes

Note that if you resize the window, proportions are kept. Also note that each stripe can be resized horizontally.

A last example. Now we change the configuration of

```
window
   addMorph: redMorph
   frame: (0@0 corner: 0.5@0.5).

window
   addMorph: blueMorph
   frame: (0.5@0 corner: 1@0.5).

window
   addMorph: greenMorph
   frame: (0@0.5 corner: 1@1).
```

As you may guess, the result is composed by two squares above a green rectangle (see Figure **??**).

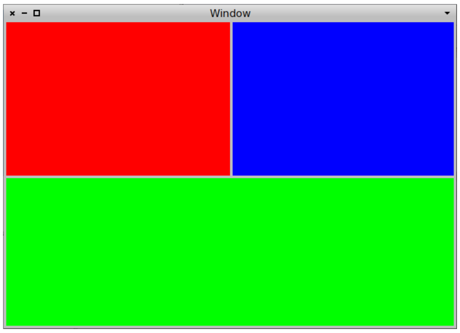


Figure 1.16: Two squares above and a rectangle below

Note that like for the previous example, you can resize each part. So basically, you now know everything about the default layout.

More complicated layouts

This section explains how to use different layouts and to use them to add morphs into another morph, which is a window or not.

Now that we have seen the default layout, we introduce you quickly the other layouts:

- LayoutFrame
- RowLayout
- StackLayout
- TableLayout

LayoutFrame

This layout is used when you have to specified both a fix part [1] and a proportional part[2].

This layout is used by example to add a toolbar.

To use this layout, you will have to use the method addMorph: aMorph fullFrame: aLayout as follows:

```
toolBarHeight := 100.
window
   addMorph: redMorph
   fullFrame: (LayoutFrame
            fractions: (0@0 corner: 1@0) "proportional part"
            offsets: (0@0 corner: 0@toolBarHeight)). "fix part"

            "Here, fractions: (0@0 corner: 1@0) means that the morph will fit the while
      width of the morph, but that the height is not dynamic (both y values are 0)"
            "and offsets: (0@0 corner: 0@toolBarHeight) means that the height is static
      and values toolBarHeight"

window
   addMorph: blueMorph
   fullFrame: (LayoutFrame
            fractions: (0@0 corner: 1@1) "proportional part"
            offsets: (0@toolBarHeight corner: 0@0)). "fix part"

   "Here fractions: (0@0 corner: 1@1) means that the moprh will    fit the whole
      container"
   "but offsets: (0@toolBarHeight corner: 0@0) precise that the top of the morph will
      always be at toolBarHeight from the top of the container"
```

As a result, you can see the red static part and the blue part which fill the space. When you resize the window, the red part will always stay the same (see Figure **??**).

Here is another example showing the fixed sized bar at the bottom of the window.

```
toolBarHeight := 100.
window
   addMorph: redMorph
   fullFrame: (LayoutFrame
            fractions: (0@0 corner: 1@1) "proportional part"
            offsets: (0@0 corner: 0@(toolBarHeight negated))). "fix part"
window
   addMorph: blueMorph
```

[1]independent of the size of the window

[2]like the ProportionalLayout

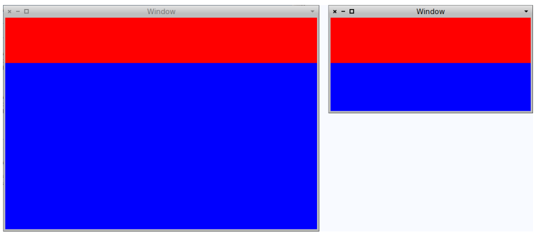


Figure 1.17: Fix red part and dynamic blue part

```
   fullFrame: (LayoutFrame
            fractions: (0@1 corner: 1@1) "proportional part"
            offsets: (0@(toolBarHeight negated) corner: 0@0)). "fix part"
```

Here you can see that the fix part (the blue one) is below (see Figure **??**).

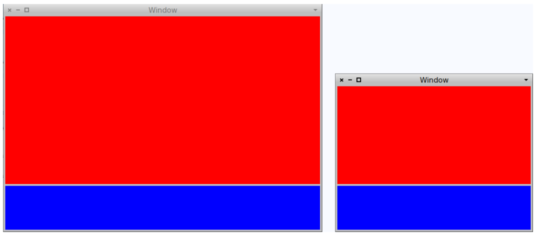


Figure 1.18: Fix blue part and red blue part

Let's try a bit more complicated example:

```
toolBarHeight := 100.
window
   addMorph: redMorph
   fullFrame: (LayoutFrame
            fractions: (0@0 corner: 1@0.4) "proportional part"
            offsets: (0@0 corner: 0@0)). "fix part"
window
   addMorph: greenMorph
   fullFrame: (LayoutFrame
            fractions: (0@0.4 corner: 1@0.4)
            offsets: (0@0 corner: 0@toolBarHeight)).
window
   addMorph: blueMorph
   fullFrame: (LayoutFrame
            fractions: (0@0.4 corner: 1@1) "proportional part"
            offsets: (0@toolBarHeight corner: 0@0)). "fix part"
window openInWorld.
```

So now, the fix part is the green morph which stick in the middle of the window (see Figure **??**).

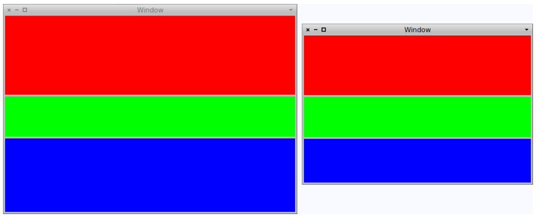


Figure 1.19: Dynamic red and blue parts and green fix part

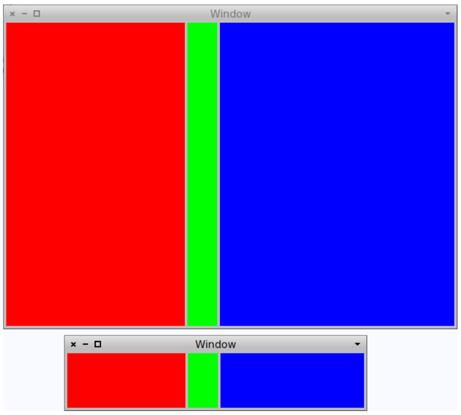


Figure 1.20: Dynamic red and blue parts and green fix part

Of course, you can do the same vertically:

```
toolBarWidth := 50.
window
   addMorph: redMorph
   fullFrame: (LayoutFrame
            fractions: (0@0 corner: 0.4@1) "proportional part"
            offsets: (0@0 corner: 0@0)). "fix part"
window
   addMorph: greenMorph
   fullFrame: (LayoutFrame
            fractions: (0.4@0 corner: 0.4@1)
            offsets: (0@0 corner: toolBarWidth@0)).
window
   addMorph: blueMorph
   fullFrame: (LayoutFrame
            fractions: (0.4@0 corner: 1@1) "proportional part"
            offsets: (toolBarWidth@0 corner: 0@0)). "fix part"
```

So the result is the same but vertically (see Figure **??**).

RowLayout

The row layout is used to add submorphs in a single row where each submorph will be equally dispatched.

So let's try to use it. The message layoutPolicy: specifies the new layout.

```
window layoutPolicy: RowLayout new.

window
   addMorph: redMorph;
   addMorph: blueMorph;
   addMorph: greenMorph.

window openInWorld.
```

As you can see, the window title bar is misplaced because it is added by the system using the same method once the layout has been changed (see Figure **??**).

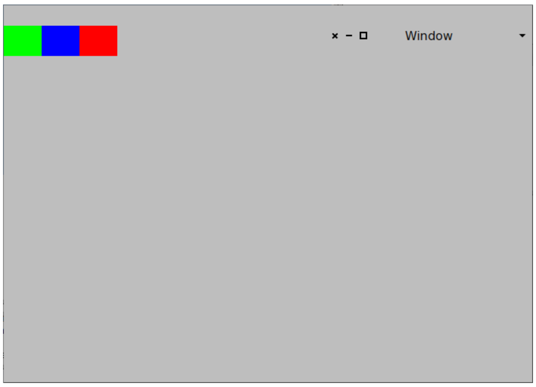


Figure 1.21: The title bar is misplaced

To fix that, we will use a container instance of PanelMorph.

```
container := PanelMorph new.
container layoutPolicy: RowLayout new.
container
   addMorph: redMorph;
   addMorph: blueMorph;
   addMorph: greenMorph.
window
   addMorph: container
```

```
    frame: (0@0 corner: 1@1).
window openInWorld.
```

As you can see, it's a bit better, but the morphs do not fill the whole space (see FIG **??**).

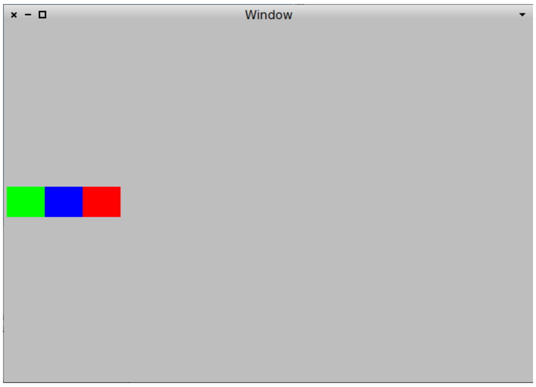


Figure 1.22: The title bar is well placed, but morphs do not fill the whole space

So let's fix that using hResizing: and vResizing:.

```
container := PanelMorph new.
container layoutPolicy: RowLayout new.
redMorph
    hResizing: #spaceFill;
    vResizing: #spaceFill.
blueMorph
    hResizing: #spaceFill;
    vResizing: #spaceFill.
greenMorph
    hResizing: #spaceFill;
    vResizing: #spaceFill.
container
    addMorph: redMorph;
    addMorph: blueMorph;
    addMorph: greenMorph.
window
    addMorph: container
    frame: (0@0 corner: 1@1).
window openInWorld.
```

So for each submorph we have specified that both vertically and horizontally it should fill the space. The result is what we have expected (see Figure **??**).

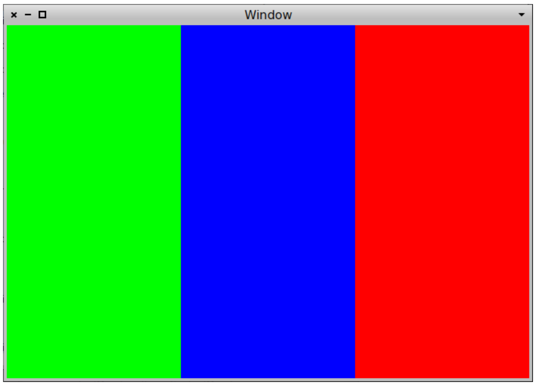


Figure 1.23: Finally what we expected

Note that contrary to the ProportionalLayout, here you can't resize any part.

StackLayout

Benjamin ►*I do not know how it works ... Should ask Igor*◄ **Stéf** ►*or may be we should drop it. Ask igor*◄

TableLayout

This layout is used to align submorphs following a row or a column and taking in account a direction. So by default, the layout builds a column directed from top to bottom.

```
container := PanelMorph new.
container
   layoutPolicy: TableLayout new;
   listDirection: #topToBottom.
{ redMorph. blueMorph. greenMorph } do: [:each |
   each
      hResizing: #spaceFill;
      vResizing: #spaceFill ].
container
   addMorph: redMorph;
   addMorph: blueMorph;
   addMorph: greenMorph.
window
   addMorph: container
   frame: (0@0 corner: 1@1).
window openInWorld.
```

So we obtain a column where any morph is dispatched equally (see Figure **??**).

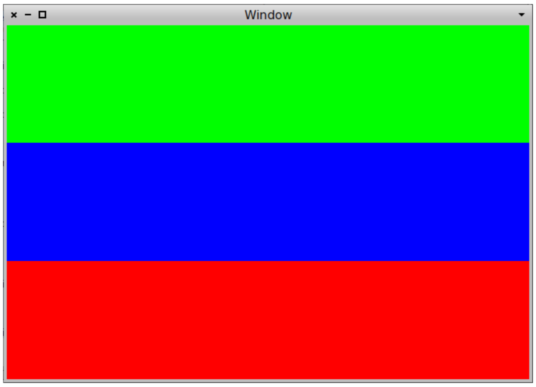


Figure 1.24: TableLayout by default

Note that you can't resize any submorph and that you have add in order: red, blue and green and you get green, blue and red.

The explanation is that when you add a morph following the direction, it's done like in Tetris, you make them fall following the direction. But the direction is kept when there is space to keep.

The following example show a situation where we do not force each morph to expand vertically

```
container := PanelMorph new.
container
   layoutPolicy: TableLayout new;
   listDirection: #topToBottom.
{ redMorph. blueMorph. greenMorph } do: [:each |
   each hResizing: #spaceFill ].
container
   addMorph: redMorph;
   addMorph: blueMorph;
   addMorph: greenMorph.
window
   addMorph: container
   frame: (0@0 corner: 1@1).
window openInWorld.
```

Here you see that the space is kept below submorphs (see Figure **??**).

You can also experiment the other directions

TableLayout is also good to get a FlowLayout using its wrapDirection. By flow layout we mean which works the same way usually text is handled: All

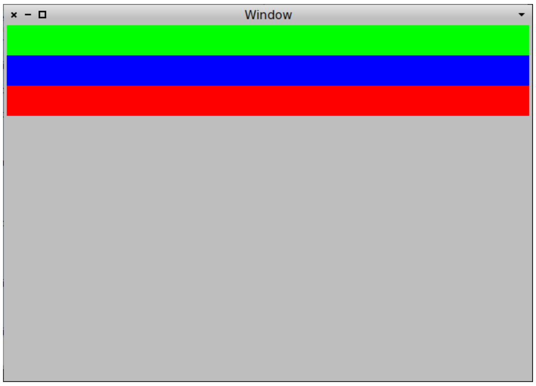


Figure 1.25: The space is left below submorphs

submorphs are put on a single line and wrapped when the border is hit.

HwaJong ►*Does reader know about UITheme here?*◄

```
|builder|
builder := UITheme builder.
(builder newRow: {
  Morph new color: Color red.
  Morph new color: Color yellow.
  Morph new color: Color green.
  Morph new color: Color blue.
  Morph new color: Color orange})
    cellInset: 10;
    wrapDirection: #topToBottom;
    openInWindow
```

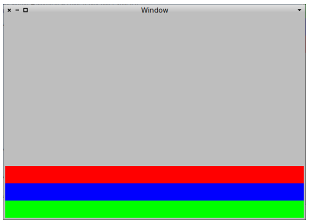


Figure 1.26: Bottom To Top

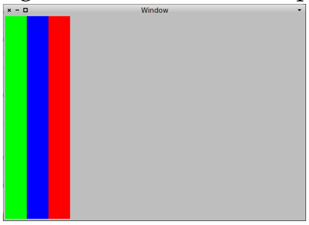


Figure 1.27: Left To Righ

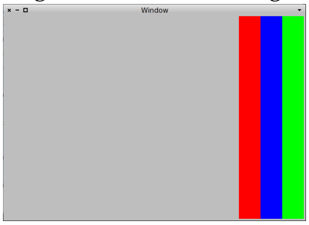


Figure 1.28: Right To Left

1.9 Some Examples

I would like to create a Text Morph that wraps the text horizontally, and expands the height to fit the text. Thus, this morph would never offer scrolling. Figure **??**.

```
| textMorph |
textMorph := UITheme builder newText: ''.
textMorph
   hResizing: #spaceFill;
   borderWidth: 1.
(UITheme builder newColumn: {textMorph}) openInWindow.
textMorph contentsWrapped: 'Some text here

Get a halo and inspect the text morph
then use #contentsWrapped: to change text'
```

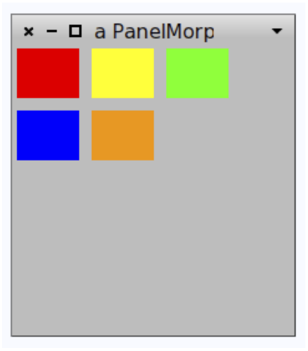
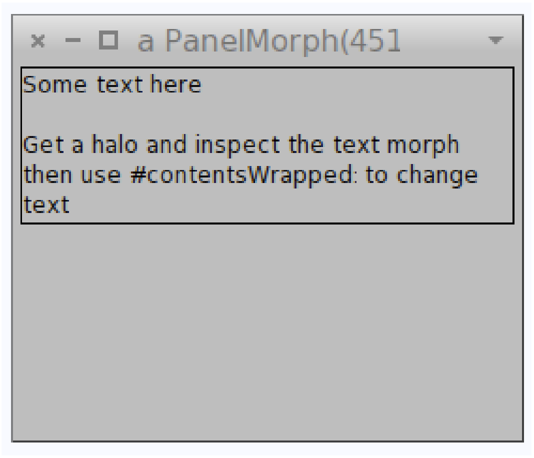


Figure 1.29: TableLayout can be used to implement FlowLayout To Left



Some text here

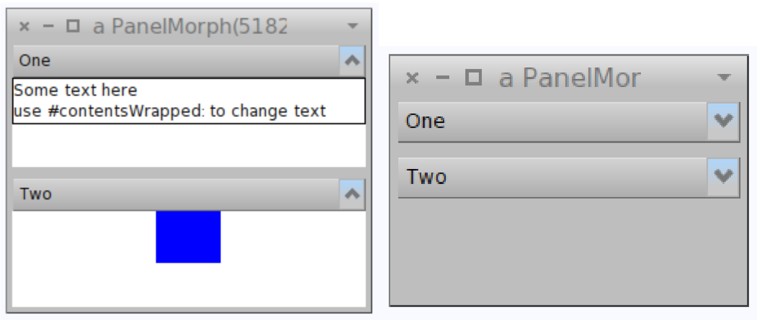
Get a halo and inspect the text morph then use #contentsWrapped: to change text

Figure 1.30: PaneMorph

If I place this one in a surrounding pane / expander / tab, how do I get the surrounding morph to resize when the text changes? See Figure **??**.

```
|textMorph|
textMorph := UITheme builder newText: ''.
textMorph
   hResizing: #spaceFill;
   borderWidth: 1.
(UITheme builder newColumn: {
   UITheme builder newExpander: 'One' for: textMorph.
   UITheme builder newExpander: 'Two' for: Morph new}) openInWindow.

textMorph contentsWrapped: 'Some text here
use #contentsWrapped: to change text'
```



Some text here
use #contentsWrapped: to change text

Figure 1.31: Two Expanders

In general, if the surrounding morph has #shrinkWrap constraints then changes to the (minimum) dimensions of its submorphs will propagate resulting in a change of size for the surrounding morph.

1.10 Conclusion