Lesserphic Tutorial

Takashi Yamamiya

VPRI Memo M-2011-002

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Lesserphic Tutorial

Takashi Yamamiya

takashi@vpri.org

2011-07-11

- Introduction
- Our first Lesserphic program
- Anatomy of LBox
- LShape
- Event Handling
- Announcement
- LObject
- Layout
- Text Field
- LBox component design
- Serialization
- Acknowledgments
- References

Introduction

Lesserphic is a GUI framework for the STEPS project written by Yoshiki Ohshima. The goal of Lesserphic is to design a lightweight graphics and event model. Unlike Morphic in Squeak where a graphical object is described in the Morphic class hierarchy, Lesserphic has only one type of graphical object named LBox (there are a few exceptions such as LWindow for topmost box and LHand for the cursor), and its behavior is described by adding components which specify aspects of a particular object.

Current Lesserphic is implemented in Squeak, but Lesserphic itself is intended as platform agnostic, and we plan to make Lesserphic much less dependent on Squeak in the future.

This text is Takashi's note about how to make a Lesserphic application. As a Lesserphic beginner, I wrote various useful tips and caveats as well as basic usage.

Our first Lesserphic program

This is a simple program in Lesserphic. A Lesserphic program is developed in the Morphic environment in the Squeak Moshi image. If you run this code in a regular morphic workspace in Moshi, you will see a tiny box inside a gradient-colored bigger box.

```
| world box |
world := LesserphicMorph new. "Create a new LesserphicMorph"
world openInWorld.            "Show the morph"
box := LBox new.              "Create a new LBox"
world window add: box.        "Add the LBox to LesserphicMorph's window"
```

The outer box is a LesserphicMorph, which works as an interface between the Morphic world and the Lesserphic world. The inner box is an instance of LBox. If you want to show a new LBox, you need to add it to another container LBox which is already showing. In this case, `world window` is used as a container. `world` (a LesserphicMorph) has a topmost LBox object that you can access by the `window`

method.

Like Morphic, you can interact with an LBox via its halo. In Lesserphic a halo appears when you control-click on an object. This is a brief description of the halo's buttons.

Todo: Insert pictures of the icons.

- Red button: A menu with various box-related functions
- Orange button: A menu for debug
- Rotate button (bottom left)
- Resize button (bottom center and half-way up the right side)
- Scale button (bottom right): Unlike morphic, Resize and Scale are different operations and must have different buttons.

  Todo: Explain in detail later?

Warning: If you see a black rectangle instead of gradient gray, you have to move the pointer on the rectangle. Lesserphic doesn't draw anything until you move the mouse.

Warning: Unlike morphic, the first halo you see is on the innermost object. Shift-control-click to get the halo of the next enclosing object.

Tip: You can find a lot of examples in the "examples" category of LesserphicMorph class.

Anatomy of LBox

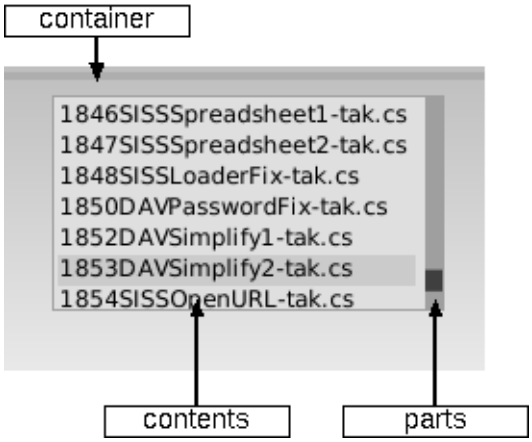
You can inspect inside an LBox by clicking the orange halo button and choosing "inspect box - morphic" from the menu. I will introduce some of an LBox's instance variables.

The first five variables are inherited from the super class LObject, and we will revisit them later in the LObject section. In particular, `updatedVector`, `whole` and `registry` are for event handling; in most cases you do not need to care about these variables. The only important variable inherited from LObject is the `properties` dictionary. You can store any instance-specific information in `properties`. Other LBox variables are:

- `container` : The parent box
- `contents` : Child boxes
- `parts` : Child boxes
- `backgroundParts` : Child boxes
- `transformation` : Position and rotation of the box
- `shape` : How the box is drawn.
- `pivotRatio` : Scale and rotation center expressed as fractions of the extent (e.g., 0.5@0.5 to mean the box's center).
- `fullDrawingBounds` : is used to optimize drawing.

A graphical object is a tree structure made of a set of LBox objects. Each LBox has one parent box in `container`, and it may have child objects in `contents`, `parts` or `backgroundParts`. `parts` and `backgroundParts` are used to store child boxes for special purpose. In a case of scroll pane, control widgets like the scroll bar are kept in the `parts`, and actual "contents" are stored in `contents`. An announcement is propagated in order of `parts`, `contents`, and `backgroundParts`. These are also drawn in the opposite order.

`wholeContents` is an array of `contents`, `parts` and `backgroundParts`, and it exists for optimization.



Unlike Morphic, LBox only specifies the structure but not visual shape. The shape is described as a LShape object in `shape` instance variable.

> Warning: Except for LSimpleLayout, the layout only works on `contents`, and not `parts` or `backgroundParts`.

LShape

While you customize the graphics on a Morph by implementing a `drawOn:` method, you can draw graphics in Lesserphic by constructing a LShape object. The next example shows how to make a LShape object and assign to an LBox

```
| world box shape |
world := LesserphicMorph new openInWorld.

shape := LGenericShape new.
shape elements: {
    (GeziraStrokedPath  polygon: {0@37. 100@37. 19@95. 50@0. 80@95. 0@37})
        fill: Color blue;
        stroke: (GeziraStroke round: 6)}.

box := LBox withShape: shape.

world window add: box.
```

In this case, I use LGenericShape to make a vector graphics. A LGenericShape object contains a series of path elements which specify shapes. Lesserphic uses Dan Amelang's Gezira as its subsystem. Gezira supports various kinds of stroke and fill styles.

> Todo: Gezira reference

Event Handling

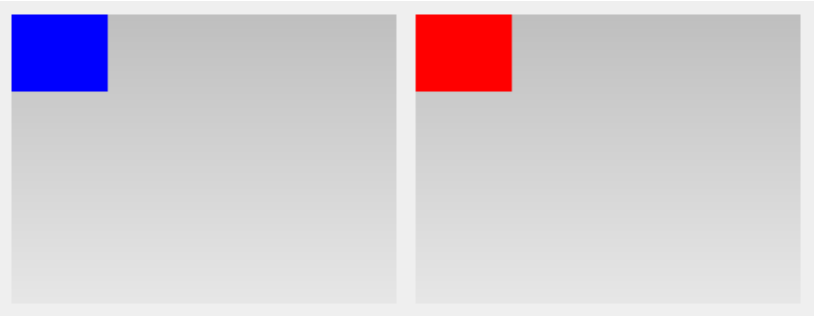
Installing a handler

So far, our examples didn't have any user interactions. Now we try to add a simple drag-and-drop feature.

```
| world box |
world := LesserphicMorph new openInWorld.
box := LBox new.
box install: LPickUpHandler new. "Install a LPickUpHandler"
world window add: box.
```

This example is almost the same as the first one but it installs LPickUpHandler in line 4. This allows you to drag the tiny rectangle. In a Morphic program, we implement an event handler as a method of the Morphic class, but in Lesserphic you can combine any number of event handlers by installing them in an object. The object responds to all of its handlers, each with its own trigger event(s).

Designing a handler



To understand the behavior of `install:`, we will implement a simple handler by ourselves. The goal is to handle a mouse event. When the pointer comes inside the LBox the box will color itself red, and when the pointer leaves the color will be set to blue.

An event handler must be a subclass of LObject.

```
LObject subclass: #LEnterLeaveHandler
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial'
```

Lesserphic's event handler can handle a set of announcements (announcement means event or notification in Lesserphic's terminology) in a handler. To define which events we want to handle, a `listensTo` method is defined. `LPointerEnter` or `LPointerLeave` is sent when a mouse pointer enters or leaves respectively. `LMotionQuery` is sent when you move the pointer, and is needed in order to handle enter and leave announcements.

The method `listensTo` must return an array of class objects. The message `asListensArray` can be used (for reasons of optimization) to convert an array of symbols to classes.

```
LEnterLeaveHandler >> listensTo
        ^ #(LPointerEnter LPointerLeave LMotionQuery) asListensArray
```

Once you specify `listensTo`, the body of the event handler follows. When an LObject receives an announcement specified in `listensTo`, the `receive:from:` method is called. Once the handler is installed into an LBox, the target LBox is accessible through the handler's `whole` variable. You can write anything you want in `receive:from:`. Typically, the incoming announcement is tested using the `isMemberOf:` method, and the appropriate code is run. We have three cases in this example.

```
LEnterLeaveHandler >> receive: ann from: obj
        ((ann isMemberOf: LMotionQuery)
                        and: [whole containsPoint: (ann localPointFor: whole)])
                ifTrue: [^ ann handled: whole].

        (ann isMemberOf: LPointerEnter)
                ifTrue: [^ whole fill: Color red].

        (ann isMemberOf: LPointerLeave)
                ifTrue: [^ whole fill: Color blue]
```

- `LMotionQuery`: sent when the user moves the pointer, wherever the pointer is. You typically need to test (using `containsPoint:`) whether the event occurred inside the object ("whole") to decide whether to handle it. To handle a subsequent `LPointerEnter` or `LPointerLeave`, you must set `whole` as the handler of the `LMotionQuery` announcement.
- `LPointerEnter`: sent when the pointer moves into the "whole" object. It makes the color red in this case.
- `LPointerLeave`: sent when the pointer leaves the "whole" object. It makes the color blue in this case.

Finally, a good Smalltalk programmer alway writes "example" class methods to show how to use the class. So let's write a method `LEnterLeaveHandler example1` that will let us test the event handling.

```
example1
        "LEnterLeaveHandler example1"
        | world box |
        world := LesserphicMorph new openInWorld.
        box := LBox new.
        box install: LEnterLeaveHandler new.
        world window add: box
```

You can download the complete source code from LEnterLeaveHandler.st.

Tip: Once a handler is installed to the target object, the handler accesses the target as `whole`.

Tip: We call a user event like LPointerEnter an "announcement".

Tip: A notification like #contents in an old MVC Browser would be considered an "announcement" in Lesserphic.

Tip: By convention, the name of any subclass of LObject is prefixed with the letter L.

Warning: You must specify all necessary announcements in both `listensTo` and `receive:from:`.

Warning: You must handle `LMotionQuery` when you want to use `LPointerEnter` and `LPointerLeave`.

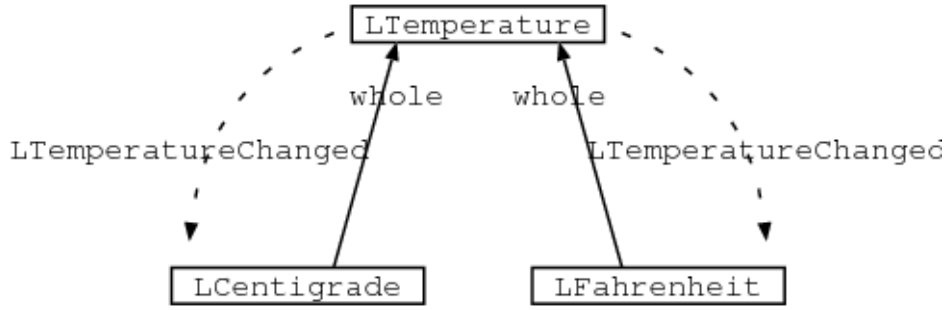
Warning: Once you specify `LMotionQuery` in `listensTo`, `LMotionQuery` is sent regardless of the pointer position.

Warning: Although, many `receive:from:` implementers return an Announcement, the return value doesn't actually matter.

Announcement

Simple Announcement

In the previous example, we made the LEnterLeaveHandler class which handles three kinds of announcement in a single object. But there is an altenative way: to handle announcements one by one. Also bear in mind that announcements are used for notification as well as interaction events. We now learn how to use announcement for notification in a fine-grained way.



In this example, we will make a simple Fahrenheit Centigrade converter using Smalltalk tools. We don't use any graphical objects. We make one model (`LTemperature`) and two views (`LCentigrade` and `LFahrenheit`).

- A model `LTemperature` has a variable `value` and its accessors. It represents temperature in Centigrade.
- A view `LCentigrade` has a variable `value` and its accessors. It is a copy of LTemperature.
- A view `LFahrenheit` has a variable `value` and its accessors. It represents temperature in Fahrenheit.
- Both a `LFahrenheit` and a `LCentigrade` depend on a `LTemperature`. When `LFahrenheit>>value:` or `LCentigrade>>value:` is sent, `LTemperature`'s value is updated, and also `LFahrenheit`'s value and LCentigrade's value are updated.
- A `LTemperatureChanged` announcement is used to notify the update.
- The variable `whole` is used by views to access the model.

`LTemperatureChanged` is just a subclass of `LAnnouncement`.

```
LAnnouncement subclass: #LTemperatureChanged
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-Announcement'
```

The model `LTemperature` is also simple. It has only a variable `value`. The interesting part is `value:` method. After a new value is set, `LTemperatureChanged` is announced by `announce:` method.

```
LObject subclass: #LTemperature
        instanceVariableNames: 'value'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-Announcement'

LTemperature >> value
        ^ value

LTemperature >> value: aNumber
        value := aNumber.
        self announce: LTemperatureChanged new.
```

One of the views, `LCentigrade`, just copies the same value from the model. Variable `whole` is defined in the superclass LObject, so you don't have to define it here. The tricky part is that `value:` doesn't assign the number to the instance variable directly. Instead, it updates the model (`whole` in Lesserphic's convention), and once the model is updated, the `LCentigrade` receives the `LTemperatureChanged` by `receive:from:` and updates itself. This prevents circular dependency. Note that there are various way to prevent circular dependency, but readers familiar with Smalltalk may recognize the approach taken here as being from traditional MVC.

```
LObject subclass: #LCentigrade
        instanceVariableNames: 'value'
        classVariableNames: ''
```

```
        poolDictionaries: ''
        category: 'LObjects-Tutorial-Announcement'

LCentigrade >> value
        ^ value

LCentigrade >> value: aNumber
        whole value: aNumber

LCentigrade >> receive: anAnnouncement from: anObject
        (anAnnouncement isMemberOf: LTemperatureChanged)
                ifTrue: [value := whole value]
```

Another view `LFahrenheit` is more interesting. While `LCentigrade` was just a copy of `LTemperature`, `LFahrenheit` actually converts numbers between Centigrade and Fahrenheit.

```
LObject subclass: #LFahrenheit
        instanceVariableNames: 'value'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-Announcement'

LFahrenheit >> value
        ^ value

LFahrenheit >> value: aNumber
        whole value: (aNumber - 32) / 1.8

LFahrenheit >> receive: anAnnouncement from: anObject
        (anAnnouncement isMemberOf: LTemperatureChanged)
                ifTrue: [value := whole value * 1.8 + 32]
```

Finally, we can try our test code. In this example, `on:to:` is used to connect the model and the views. Also, the views use `whole:` to remember the model. Once the relationship is established, the `LCentigrade` and the `LFahrenheit` affect each other.

```
| c f model |
c := LCentigrade new.
f := LFahrenheit new.
model := LTemperature new.
model on: LTemperatureChanged to: c.
model on: LTemperatureChanged to: f.
c whole: model.
f whole: model.
f value: 86.
Transcript cr; show: f value; show: 'F = '; show: c value; show: 'C'.
c value: 20.
Transcript cr; show: f value; show: 'F = '; show: c value; show: 'C'.
```

If you run the code on the workspace, the output is shown in the Transcript

```
86.0F = 30.0C
68.0F = 20C
```

Tip: You can make a simple model/view relationship by `on:to:`, `receive:from:` and `announce:`.

Relationship between `install:` and `on:to:`

I introduced `install:` to add a behavior in an LBox at first, then I constructed a dependency by `on:to:` to show a low-level update mechanism. Now we are going to rewrite the Fahrenheit/Centigrade converter in terms of `install:` to show how `install:` works.

As we saw earlier in LEnterLeaveHandler, `install:` requires the `listensTo` method to specify which events need to be handled.

```
LCentigrade >> listensTo
        ^ #(LTemperatureChanged) asListensArray

LFahrenheit >> listensTo
        ^ #(LTemperatureChanged) asListensArray
```

This is pretty much all we need. And the example script can be written with `install:`. The output is identical to the previous case.

```
| model |
model := LTemperature new.
model install: LCentigrade new.
model install: LFahrenheit new.
(model \ #Fahrenheit) value: 86.
Transcript cr; show: '86F = '; show: (model \ #Centigrade) value; show: 'C'.
(model \ #Centigrade) value: 20.
Transcript cr; show: '20C = '; show: (model \ #Fahrenheit) value; show: 'F'.
```

Basically, `install:`does three things.

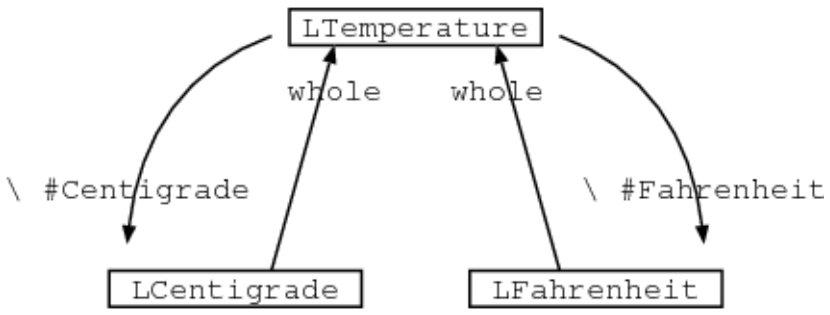
- The announcement relationship between the receiver and the argument is established.
- The argument is set in the argument's `components`.
- The receiver is set as the argument's `whole`.

One noticeable thing here is that the view is stored as the model's component, and can be accessed by name using the `\` (backslash) operator. The component's name is determined by the component's class name. By default, the name is chosen from the class name by getting rid of the "L" prefix. Unlike the one-way dependency made by `on:to`, `install:` makes an explicit double link between a whole and its component. This is Ted Kaehler's basic idea of Membrane [1].

You can download the complete source code from LObjects-Tutorial-Announcement.st.

Tip: A component's default name is the class name without the "L" prefix.

Membrane



Not only LBox, but any LObject can be built from components. Unlike the ordinary relationship between Smalltalk's objects and dependants where a dependant doesn't necessarily know about the model object, a component always has access to the owner through the `whole` variable.

Todo: Short description of Membrane's goal.

Implicit announcement with `LValueChanged`

There are many cases where we want to receive an announcement when some instance variable is updated. In our previous example, `announce:` is used to dispatch an announcement from the setter method. But by using an `LValueChanged` (or its subclasses), you can avoid having to write `announce:` explicitly.

Typically, LValueChanged is used in a GUI program. In each rendering step, Lesserphic scans all modified variables where an LValueChanged is needed as specified by `listensTo`. If a modified variable is found, Lesserphic makes an announcement for it. This happens only once in a rendering cycle in `LWorldState >> processActions`, even if some setter has been called many times since the previous cycle.

To learn about LValueChanged, let us rewrite our program. The model `LTemperature` is almost identical but we don't need to use `announce:` any more. So `value:` becomes simpler.

```
LTemperature value: aNumber
        value := aNumber
```

Instead of `LTemperatureChanged`, we use predefined LValueChanged. So we rewrite `listensTo:` and `receive:from:`.

```
LCentigrade >> listensTo
        ^ #(LValueChanged) asListensArray

LCentigrade >> receive: anAnnouncement from: anObject
        (anAnnouncement isMemberOf: LValueChanged)
                ifTrue: [value := whole value]
```

```
LFahrenheit >> listensTo
        ^ #(LValueChanged) asListensArray

LFahrenheit >> receive: anAnnouncement from: anObject
        (anAnnouncement isMemberOf: LValueChanged)
                ifTrue: [value := whole value * 1.8 + 32]
```

Finally, the example script must be rewritten. Note that `LValueChanged` is announced only in a GUI process, so we need to create a Lesserphic window. And each time the value is modified, the `step` method must be invoked to send announcements. This is done by the system automatically in normal cases, but we did it by hand because we want to show the result in the same process. In other words, `LValueChanged` is processed asynchronously.

```
| model world |

world := LesserphicMorph new openInWorld.

model := LTemperature new.
model install: LCentigrade new.
model install: LFahrenheit new.
(model \ #Fahrenheit) value: 86.
world step.
Transcript cr; show: '86F = '; show: (model \ #Centigrade) value; show: 'C'.

(model \ #Centigrade) value: 20.
world step.
Transcript cr; show: '20C = '; show: (model \ #Fahrenheit) value; show: 'F'.

world delete.
```

You can download the complete source code from .

Tip: `LValueChanged` is announced only once in a rendering cycle.

Warning: `LValueChanged` is processed asynchronously.

LObject

Now we are ready to examine all the instance variables in an LObject.

- `updatedVector` : Used by LValueChanged mechanism to record all instance variables updated between two rendering cycles.
- `properties` : A dictionary to hold instance-specific data.
- `whole` : Announcements come from the whole object.
- `registry` : A dictonary to map from an announcement class to the receiver objects
- `components` : A dictonary to keep components. As a convention, the default name is the class name without the "L" prefix.

These instance variables are managed by the Lesserphic framework, so you must not touch them directly except `whole`. There are plenty of accessors to handle an LObject's internal state. From a programmer's point of view, there are three categories of memory slot in an LObject: instance variables, properties, and components. Each category has its own character, so it is important to choose the most suitable one.

Components

Components are used if

- the member is optional
- the member has a reference to the owner (whole)
- the member needs to receive announcements from the owner.

LObject has several methods to manage components.

- `LObject >> install:` Add a component to the receiver.
- `LObject >> install:as:` Add a component as the specified name.
- `LObject >> componentAt:` Get the component at the name.
- `LObject >> \` A synonym of `componentAt:`.
- `LObject >> removeComponentAt:` Remove a component at the name.
- `LObject >> uninstallComponentAt:` Same as `removeComponentAt:`.
- `LObject >> removeComponent:` Remove the component.

Properties

Properties are used if the member is optional. LObject has several methods to manage properties.

- `LObject >> valueOfProperty:` Get the property at the name.
- `LObject >> valueOfProperty:ifAbsent:` Get the property at the name. If it doesn't exist, the second argument is evaluated.
- `LObject >> valueOfProperty:ifAbsentPut:` Get the property at the name. If it doesn't exist, the second argument is evaluated and written as the property value.
- `LObject >> setProperty:toValue:` Add a property value under the given name.
- `LObject >> removeProperty:` Remove the property.
- `LObject >> removeKey:ifAbsent:` Remove the property. Evaluate the second argument if the property does not exist.
- `LObject >> hasProperty:` Return true if the property exists, otherwise false.
- `LObject >> keyAtValue:ifAbsent:` Return the key (i.e., property name) that has the argument as its value. Evaluate the second argument if no such key exists.

Instance variables

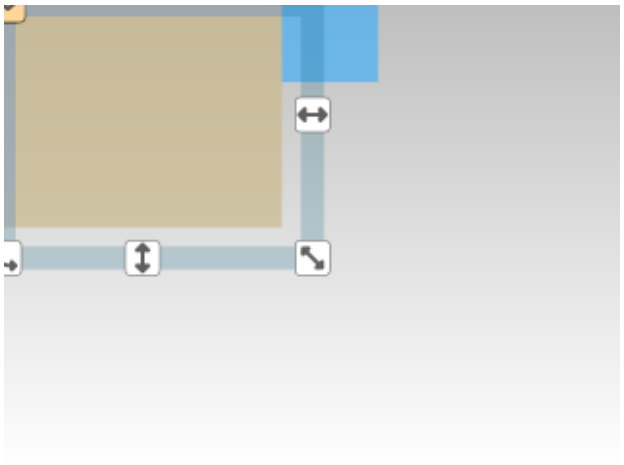
Instance variables are used if all instances of the class has the member.

Layout

Lesserphic has a layout management system to arrange LBoxes according to a set of rules. This mechanism is powerful enough that even the characters in a text editor are laid out using this system. We are going to see a few examples to learn how layout works. The first example shows the simplest way to arrange LBoxes horizontally.

Basic layout

```
| world box1 box2 |
world := LesserphicMorph new openInWorld.
world window layout: LHorizontalLayout new.

box1 := LBox new.
box2 := LBox new.
box1 pivotRatio: 0 @ 0.
box2 pivotRatio: 0 @ 0.

world window add: box1; add: box2.
```



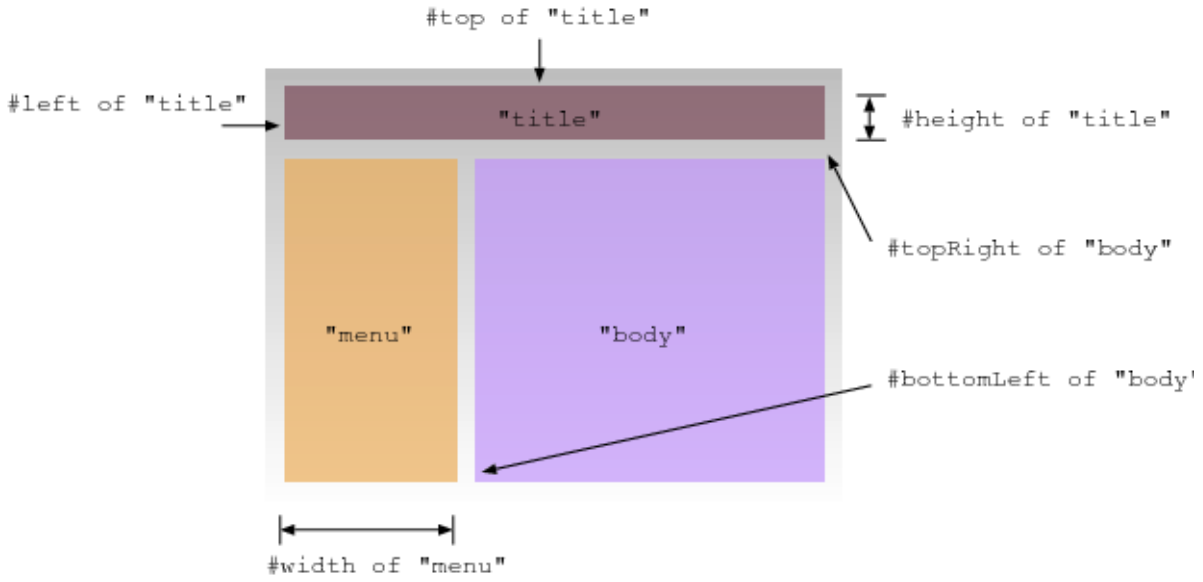
This program shows two boxes in the world. When you modify the extent using the halo, these two boxes maintain their horizontal relationship. `LHorizontalLayout` is a key element of the program. Once a layout object is installed by `layout:` method into the container, the child objects are arranged by the layout object. In an `LHorizontalLayout`, new coordinates are calculated based on `pivotRatio` in a box. In this case, each child box is given a `pivotRatio` of 0@0 (the top left corner). If you replace `LHorizontalLayout` with `LVerticalLayout`, the child boxes are arranged vertically.

Simple layout

LSimpleLayout provides a more flexible way to arrange boxes. Using LSimpleLayout, you can specify constraints between the positions of boxes in terms of pixels or of ratios relative to the container's size. To use LSimpleLayout, you need to name each box. In the next example, we make three boxes named "title", "menu", and "body".

```
| world layout |
world := LesserphicMorph new openInWorld.

"LSimpleLayout requires that each box has unique name."
world window add: (LBox new name: 'title').
world window add: (LBox new name: 'menu').
world window add: (LBox new name: 'body').
```

```
layout := LSimpleLayout new.
world window layout: layout.

layout keep: #top of: 'title' to: 10.
layout keep: #left of: 'title' to: 10.
layout keep: #height of: 'title' to: 30.
layout keep: #right of: 'title' to: #right offset: -10.

layout keep: #topLeft of: 'menu' to: #bottomLeft of: 'title' offset: 0@10.
layout keep: #width of: 'menu' to: #width scale: 0.3.
layout keep: #bottom of: 'menu' to: #bottom offset: -10.

layout keep: #topRight of: 'body' to: #bottomRight of: 'title' offset: 0@10.
layout keep: #bottomLeft of: 'body' to: #bottomRight of: 'menu' offset: 10@0.
```



When layout is necessary, a LSimpleLayout recalculates each bounding box to satisfy all constraints. In the basic case, each constraint describes the subject box's attribute in terms of a reference box's attribute with distance parameters specified either in pixels (offset) or as a ratio (scale). Constraints are constructed by the `LSimpleLayout >> keep:` method family. The most generic form is `LSimpleLayout >> keep:of:to:of:scale:offset:`.

```
layout
  keep: subjectAttr
  of: subject
  to: referenceAttr
  of: reference
  scale: refScale
  offset: refOffset
```

- *`subjectAttr`* : The attribute name used by the constraint. Possible values are `#top, #bottom, #left, #right, #height, #width, #extent, #hcenter, #vcenter,` and these combinations (see decomposing category in `LSimpleLayout`).
- *`subject`* : The name of the subject box.
- *`referenceAttr`* : The attribute name of the reference. Possible values are same as *`subjectAttr`*. *`subjectAttr`* is used as the default value.
- *`reference`* : The name of the reference box. The container box is used as the default value.
- *`refScale`* : A point or a number. Relative scale to the reference box. The default value is 1.
- *`refOffset`* : A point or a number. Relative offset to the reference box. The default value is 0.

But you can use other variations if you don't need some parameters: for example, `LSimpleLayout >> keep:of:to:offset:` is used when you don't need *`reference`* nor *`refScale`*. For example:

```
layout keep: #right of: 'title' to: #right offset: -10.
```

is equivalent to:

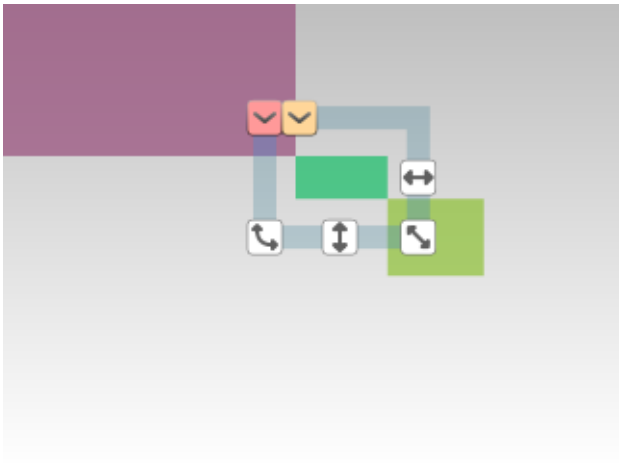
```
layout keep: #right of: 'title' to: #right of: nil scale: 1 offset: -10.
```

It makes the right side of the "title" box be 10 pixels left of the right side of the container.

Make a layout by yourself

It is not too difficult to implement your own layout manager. The essential method required in a layout class is `layOut:`. It is called with the container object as the argument when layout is neccesary. The most easy way to implement a layout class is to make a subclass of LLayout.

```
LLayout subclass: #LDiagonalLayout
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-Layout'

LDiagonalLayout >> layOut: container
        container contents
                inject: 0 @ 0
                into: [:topLeft :anLBox |
                        anLBox topLeft: topLeft.
                        anLBox bottomRight]
```



This layout arranges child boxes so that each box's bottom right corner touches next box's top left corner. You can download the complete source code from LDiagonalLayout.st.

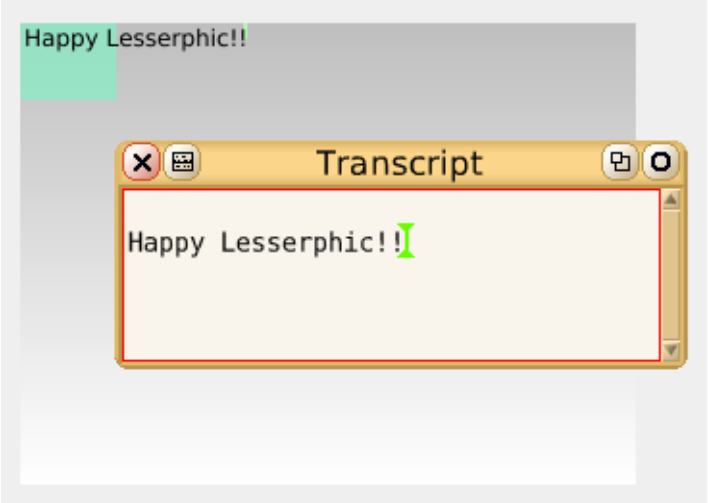
Text Field

A text editor is also one of the layouts in Lesserphic. A simple way to display a text is with the `LWordWrapLayoutF` layout and `textContents:` method. This example shows "Hello, World!" in a box on the window.

```
| world text |
world := LesserphicMorph new openInWorld.

text := LBox new.
text layout: LWordWrapLayoutF new.
text textContents: 'Hello, World!'.

world window add: text.
```

Different text layout classes allow for different types of text field. Using `LLineEditor`, you can make a one-line text field which triggers an action. The next example responds with the text that you type into such a text field.

```
| world text |
world := LesserphicMorph new openInWorld.

text := LBox new.
text layout: LLineEditor new.
text textContents: '<INPUT HERE>'.
text on: LContentAccepted
       send: #value:value:
       to: [:ann :box | Transcript cr; show: box textContents].

world window add: text.
```

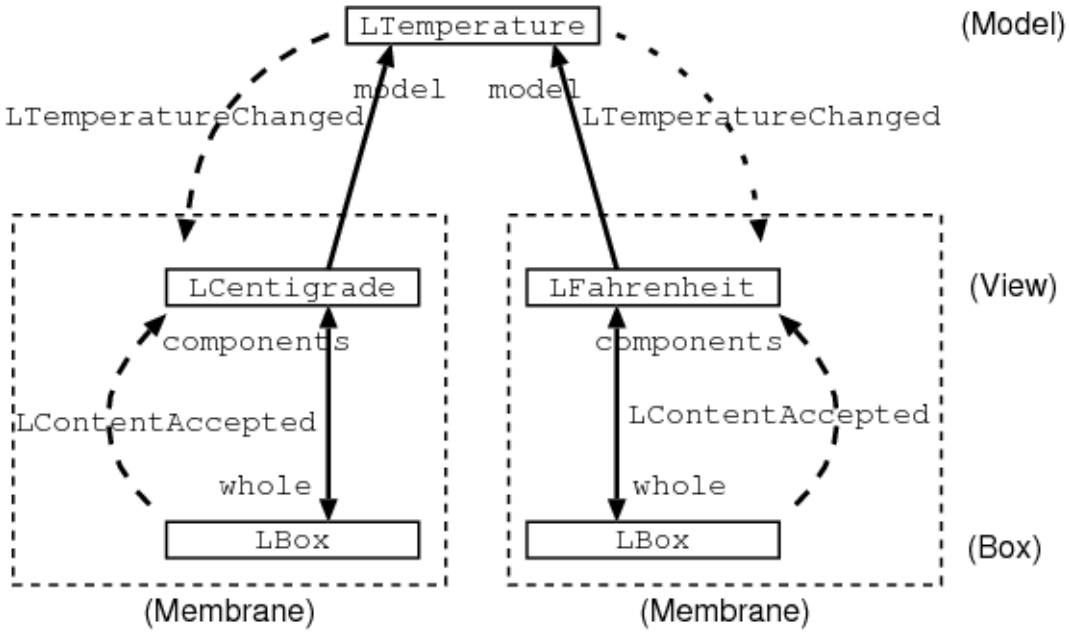


In this example, `LObject >> on:send:to:` configures the event handler to receive the text input. The method `on:send:to:` is a variation of `on:to:`, and sends a message specified by the second argument to the third argument. The message's first argument is the announcement itself. In this case, we send `#value:value` to a block given as the third argument. It turns out that the text box object is passed to the block through the temporary variable `box` when you type something, and the Transcript shows the content.

Warning: Sometime text layout becomes broken. Try `LFamily initialize` to fix it.

LBox component design

We have covered pretty much all techniques needed to make a Lesserphic application. As a summary, we shall implement yet another temperature converter with a graphical user interface. Before getting into the implementation, take a moment to examine a typical Lesserphic application design.



This diagram shows the relationships between the objects we need. It has three layers of object: model, view, and box. A solid line shows an explicit reference (an instance variable), and a dotted line shows an announcement. An `LTemperature` is a model which keeps the actual state: in this case, a temperature in Centigrade. `LCentigrade` and `LFahrenheit` are views of different perspectives of the model. Each view has its own LBox.

When you hit the Enter key within the text box, an `LContentAccepted` is dispatched from the `LBox` to the components, and one of views receives the announcement and sets the value to the model. When the value is changed on the `LTemperature` model, it dispatches a `LTemperatureChanged`, and then, these two views receive the announcement and interpret the model's value according to the type of the view. Finally, the boxes are updated with the new value.

Model

The implementation is quite straightforward. The model is the same as in the former example.

```
LObject subclass: #LTemperature
        instanceVariableNames: 'value'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-GUI'

LTemperature >> value
        ^ value

LTemperature >> value: aNumber
        value := aNumber.
        self announce: LTemperatureChanged new.
```

View

The implementations of `LCentigrade` and `LFahrenheit` share many common methods, so we implement a common superclass named `LTemperatureView`. It has an instance variable `model` to point to the model.

```
LObject subclass: #LTemperatureView
        instanceVariableNames: 'model'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-GUI'
```

The actual behavior of the view is defined by the accessors `value` and `value:`. They must be implemented by its subclasses.

```
LTemperatureView >> value
        self subclassResponsibility

LTemperatureView >> value: aNumber
        self subclassResponsibility
```

To establish a relationship to the model, the `model:` method is configured to accept an `LTemperatureChanged` using `on:to:`.

```
LTemperatureView >> model: anLTemperature
        anLTemperature on: LTemperatureChanged to: self.
        model := anLTemperature
```

Also, to establish a relationship to the box, `LContentAccepted` is accepted. In this case, `listensTo` is used to make a whole-component (membrane) relationship.

```
LTemperatureView >> listensTo
        ^ #(LContentAccepted) asListensArray
```

A `LTemperatureView >> receive:from:` accepts two kinds of announcement. When an `LContentAccepted` is dispatched from the box, the value of the model is set by `LTemperatureView >> value:` which is implemented by the subclass

When an `LTemperatureChanged` is dispatched, the text content of the box is set using `LTemperatureView >> value` which is also implemented by the subclass.

```
LTemperatureView >> receive: anAnnouncement from: anObject
        (anAnnouncement isMemberOf: LContentAccepted)
                ifTrue: [self value: anAnnouncement content asNumber].
        ((anAnnouncement isMemberOf: LTemperatureChanged)
                and: [whole notNil])
                ifTrue: [whole textContents: self value printString]
```

The concrete classes are simple. `LCentigrade` uses just the same value as the model because the model keeps its temperature value in Centigrade.

```
LTemperatureView subclass: #LCentigrade
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-GUI'

LCentigrade >> value
        ^ model value

LCentigrade >> value: aNumber
        model value: aNumber
```

And `LFahrenheit`'s accessors are implemented as conversion functions.

```
LTemperatureView subclass: #LFahrenheit
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-GUI'

LFahrenheit >> value
```

```
        ^ model value * 1.8 + 32

LFahrenheit >> value: aNumber
        model value: (aNumber - 32) / 1.8
```

Announcement

`LTemperatureChanged` is identical as former examples.

```
LAnnouncement subclass: #LTemperatureChanged
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'LObjects-Tutorial-GUI'
```

Box

Finally, we implement GUI elements in `LTemperature`'s class side.

`LTemperature class >> newTextFieldAt:` builds a text field at the specified place. It uses `LLineEditor` as the layout so it can handle the enter key.

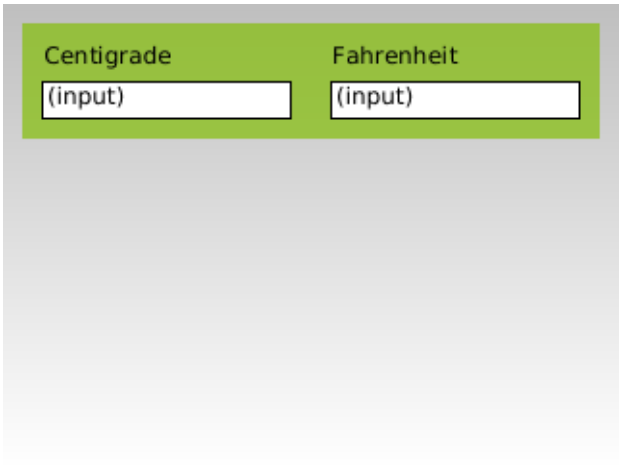
```
LTemperature class >> newTextFieldAt: aPoint
        | text |
        text := LBox new.
        text layout: LLineEditor new.
        text textContents: '(input)'.
        text fill: Color white.
        text extent: 140 @ 20.
        text borderWidth: 1.
        text topLeft: aPoint.
        ^ text
```

`LTemperature class >> example3` is the test script to show two text fields for both Centigrade and Fahrenheit. `LBox >> newLabel:` is used to show a non-editable text label.

```
LTemperature class >> example3
        "A temperature converter GUI version"
        "LTemperature example3"
        | world cLabel fLabel cField fField model cView fView box |
        world := LesserphicMorph new openInWorld.
        box := LBox new name: 'FCconverter'.

        model := LTemperature new.
        cView := LCentigrade new model: model.
        fView := LFahrenheit new model: model.

        cLabel := LBox newLabel: 'Centigrade'.
        cLabel topLeft: 10 @ 10.
        fLabel := LBox newLabel: 'Fahrenheit'.
        fLabel topLeft: 160 @ 10.

        cField := self newTextFieldAt: 10 @ 30.
        cField install: cView.

        fField := self newTextFieldAt: 160 @ 30.
        fField install: fView.

        box add: cLabel.
        box add: fLabel.
        box add: cField.
        box add: fField.
```

```
        box bounds: (10@10 extent: 300 @ 60).
        world window add: box.
```

This is the screen shot of the program. You can download the complete source code from LObjects-Tutorial-GUI.st.



Serialization

Simple usage

Lesserphic has a serialization mechanism named SISS to save a snapshot of an object on a storage device. The format is based on S-expressions. Let me show you a simple example. If you run `Time now sissScanObjects prettyString` in a workspace, you will see the result:

```
(objects :root "1"
        (Time :idref "1"
                (slot :class "SmallInteger" :value "34167" :name "seconds")
                (slot :class "SmallInteger" :value "0" :name "nanos")))
```

The format is simple. The top, or **first level** node `objects` shows where the root object is. This value is always "1". The second level node `Time` represents a Time object which has two instance variables named `seconds` and `nanos`. As you see, the default behavior of SISS is to write all of the instance variables.

Although an S-expression can represent any data in general, SISS uses it in a more restricted way. The generic form of a SISS element is:

```
(Tag :Key1 Value1 :Key2 Value2 :Key3 Value3 ...
        Child1
        Child2
        Child3 ...)
```

where *Values* are always `String`. If you see the form carefully, you will notice that SISS is just another form of XML.

You can restore the original value by `asSExpElement sissReadObjects` :

```
Time now sissScanObjects prettyString asSExpElement sissReadObjects
=> 9:37:52 am
```

Now we have used four methods to serialize and de-serialize by SISS. These methods are used for debugging purposes; I will explain a real use case later.

- `Object >> sissScanObjects` : Serialize the receiver and create a SExpElement.
- `SExpElement >> prettyString` : Write easy to read representation.
- `String >> asSExpElement` : Create a SExpElement from the string.
- `SExpElement >> sissReadObjects` : Restore the original value.

Various SISS forms

An object is serialized to various forms by SISS. Try this expression:

```
(Array  with: 1
        with: 'hello'
        with: Time now) sissScanObjects prettyStringWithSorting
```

The output shows various kinds of SISS representations. The first `Array` object has indexed slots, and the second `ByteString` has no slot but the value is represented by the `:value` property, and the third `Time` object has key-value slots.

```
 (objects :root "1"
        (Array :idref "1" :basicSize "3"
                (slot :class "SmallInteger" :value "1" :name "1")
                (slot :idref "2" :name "2")
                (slot :idref "3" :name "3"))
        (ByteString :idref "2" :value "hello" :basicSize "5")
        (Time :idref "3"
                (slot :class "SmallInteger" :value "43195" :name "seconds")
                (slot :class "SmallInteger" :value "0" :name "nanos")))
```

The output has three second level nodes. The form of the **second level** is:

```
(Class :idref idref properties ...
        slots ...)
```

where *idref* is a serial identifier which is refered by other objects, and *slots* shows contents of the object. The **third level** is slot descriptions:

```
(slot properties ... :name name)
```

where *name* is either a slot name or an index number. The *property* of slot description is different depending on the type. A slot is either **literal** or **non-literal**. If it is literal, the value is described as inline in the slot e.g. the first element of the array `(slot :class "SmallInteger" :value "1" :name "1")`, but if it is non-literal, *idref* is used to point to other second level element e.g. `(slot :idref "2" :name "2")`.

A literal object must be immutable and it can be described inline. But a non-literal object is mutable and can be shared by multiple objects, and *idref* is necessary to point the object.

Warning: Class name is described as a tag in the second level e.g. `(Array ...)`, but as a value in the third level e.g. `(slot :class "SmallInteger" ...)`.

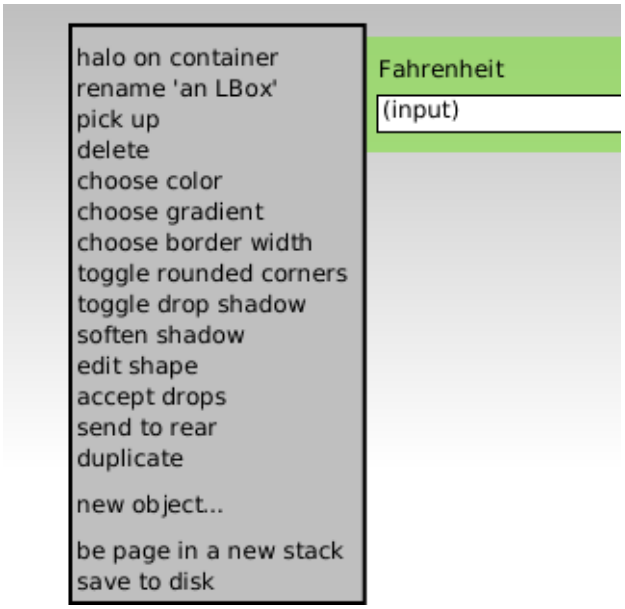
Warning: *Idref* in the second level is used to define the identifier, but *idref* in the third level is used to refer to the identifier.

Warning: A second-level element is always a non-literal object.

Save a Lesserphic object by SISS

Lesserphic provides a way to save your Lesserphic object using SISS. To save a Lesserphic object, open

the halo (control+click on the object), click the red button, and select the "save to disk" item on the menu. You will be asked to enter the file name. The file name must end with "`.lbox`"



To show a saved object on the screen, evaluate the following expression:

```
LesserphicMorph openURL: 'name.lbox'
```

Note that our temperature converter doesn't work in this way yet. To work with SISS mechanism properly, we need to customize the serializing methods.

Customize SISS

Although the default SISS serializer does its best to save your object, what SISS does might not be what you want. SISS tries to save all of the instance variables in the object, but there are various cases where this behavior is not desirable:

- Some instance variables act as cache, and are not supposed to be serialized.
- Announcement handlers must be installed.
- The object has a reference to some global object e.g. world state.

Above all, saving all instance variables breaks encapsulation, so you are encouraged to implement your own serializing methods. There are three major methods that allow you to customize the behavior of SISS.

- `Object >> sissContentsInto:context:` Customize serializing object.
- `Behavior >> sissCreateInstanceFromSexp:idref:from:to:` Customize de-serializing object.
- `Object >> sissComeFullyUpOnReloadFrom:to:` A hook called after all objects are de-serialized.

To see how those methods work, we implement them for the model class `LTemperature`. It is simple because `LTemperature` has only one number.

```
LTemperature >> sissContentsInto: sexp context: ctx
        sexp attributeAt: #value put: value printString

LTemperature class >> sissCreateInstanceFromSexp: sexp idref: idref from: from to: to
        | model |
        model := self new.
        to at: idref put: model.
        model value: (sexp attributeAt: #value) asNumber.
        ^ model
```

`sissContentsInto:context:` serializes a LTemperature object and the member is stored as a property named `value`. A created form looks like this:

```
        (LTemperature :idref "76" :value "10")
```

And the class side method `sissCreateInstanceFromSexp:idref:from:to:` is for the de-serializer. The method makes a new instance of LTemperature and registers *idref* to *to* object which is a dictionary mapping *idref* and the value.

The next example is for LTemperatureView. It shows more complicated object which includes non-literal object.

```
LTemperatureView >> sissContentsInto: sexp context: ctx
        sexp
                addElementKeyValues: {
                        #whole -> self whole.
                        #model -> self model}
                context: ctx

LTemperatureView class >> sissCreateInstanceFromSexp: sexp idref: idref from: from to: to
        | view child |
        view := self new.
        to at: idref put: view.
        sexp elementsDo: [:each |
                child := self fromSexp: each from: from to: to.
                (each attributeAt: #name) = 'whole' ifTrue: [view whole: child].
                (each attributeAt: #name) = 'model' ifTrue: [view model: child]].
        ^ view
```

In this example, `sissContentsInto:context:` uses `SExpElement >> addElementKeyValues:context:` to make slots for members both `whole` and `model`. Because it uses public accessors, the file format is not changed even when you modify the internal representation of `LTemperatureView`. A created form looks like this:

```
        (LCentigrade :idref "75"
                (slot :idref "73" :name "whole")
                (slot :idref "76" :name "model"))
```

The class side `sissCreateInstanceFromSexp:idref:from:to:` emulates all the slots by `SExpElement >> elementsDo:` and store by `whole:` and `model:`. Again, because it uses the public setters, the announcement handler is properly installed in the setter.

Now we have a Fahrenheit Centigrade converter with GUI, and it can be saved in a document by SISS. You can download the complete source code from LObjects-Tutorial-GUI.st.

Warning: You can use arbitrary slot names in SISS, but you can't customize tag (class) names in the current framework.

Acknowledgments

I would like to thank Aran Lunzer and Ted Kaehler to correct many technical and grammatical mistakes, and Yoshiki Ohshima to give useful feedback!

References

1. A Membrane with Parts: A new object model, Ted Kaehler