

The Mjølner BETA System
Lidskjalv:
User Interface Framework
Tutorial

Mjølner Informatics Report

MIA 95-30(1.1)

August 1996

Copyright © 1995-96 Mjølner Informatics ApS.
All rights reserved.
No part of this document may be copied or distributed
without prior written permission from Mjølner Informatics

Contents

THE LIDSKJALV USER INTERFACE FRAMEWORK	1
1 STRUCTURE OF A LIDSKJALV APPLICATION	5
1.1 LIDSKJALV DECLARATIONS	5
1.2 LIDSKJALV INITIALIZATION	6
1.3 LIDSKJALV EVENT HANDLING.....	6
1.4 APPLICATION SUSPEND, RESUME AND TERMINATE.....	6
1.5 OVERVIEW OF LIDSKJALV.....	7
2 ACCESS TO GLOBAL STRUCTURES	7
2.1 ACCESS TO THE MOUSE.....	7
2.2 ACCESS TO THE CLIPBOARD	8
2.3 STANDARD INPUT AND OUTPUT	8
2.6 COORDINATE SYSTEM.....	8
3 WINDOWS	9
3.1 EVENT HANDLING IN WINDOWS.....	10
3.2 CONTENTS OF WINDOWS	11
3.4 WINDOW ITEMS	11
4 MENU HANDLING	12
4.1 BASIC MENU FACILITIES.....	12
4.1.1 <i>Menu Item Facilities</i>	13
4.1.2 <i>Static and Dynamic Menu Items</i>	14
5 THE MENU BAR.....	16
5.1 STANDARD MENUS	16
6 GRAPHICS	17
6.1 FIGURE ITEMS	18
6.2 INTERACTIVE GRAPHICS FACILITIES.....	20
6.2.1 <i>Selection of Graphics Objects</i>	20
6.2.2 <i>Dragging of Graphics Objects</i>	21
6.3 CANVAS.....	21
6.4 SCROLLER.....	22
7 CONTROLS AND DIALOG BOXES	23
7.1 CONTROL ITEMS.....	24
8 SCROLL LISTS	27
9 WINDOW FIELDS.....	28
10 STANDARD DIALOGS.....	30

The Lidskjalv User Interface Framework

Lidskjalv is a **platform independent** object-oriented user interface construction toolkit for:

- Macintosh
- X Window System (Motif Widgets)
- Windows 95
- Windows NT

Lidskjalv allows construction of **portable user interfaces** in such a way that the look-and-feel of the applications, will conform to the standardized **look-and-feel** of the specific platform.

The framework defines abstractions for all commonly used interface objects, such as **windows, menu bars, menus, buttons, text fields, figure items, scrolling lists**, etc.

The application programmer does not have to handle **user interaction** at the event level of the underlying platform, because each interface object takes care of the interactions related to itself. It is the responsibility of the entire framework to ensure that the user interactions (such as mouse button presses, key presses, etc.) are converted internally into invocations of **virtual procedures** of the appropriate interface object. The only thing the application programmer needs to do is to bind the virtual procedures. All layout properties of interface objects can be **manipulated** through the Lidskjalv framework.

Following, a realistic example of using Lidskjalv is presented. It is a small text editor with full support for loading, editing, and saving files.

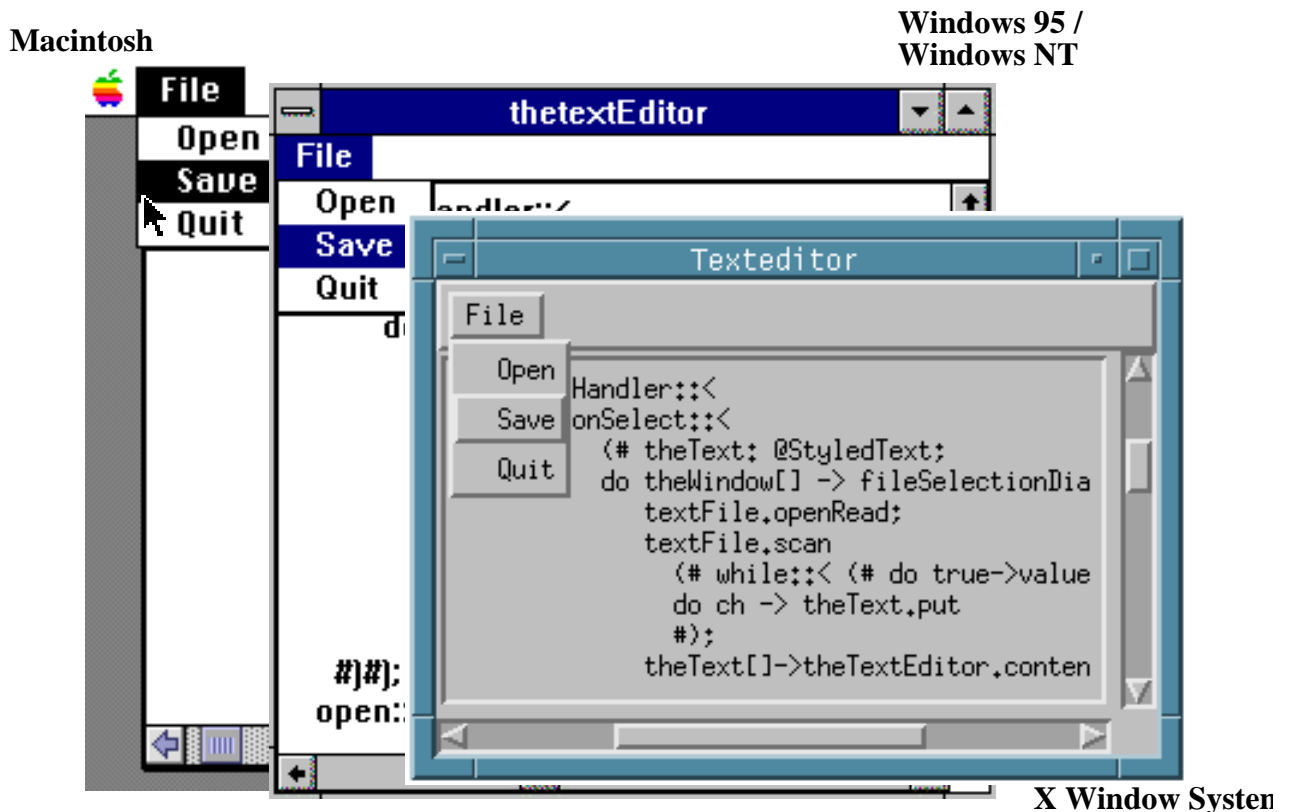
```
ORIGIN '~beta/guienv/v1.4/fields';
INCLUDE '~beta/guienv/v1.4/stddialogs';
INCLUDE '~beta/basiclib/v1.5/file';
-- program: descriptor --
guienv
(# theWindow: @window
  (# menubarType:
    (# fileMenu: @menu
      (# textFile: @file;
        openItem: @menuItem
          (# eventhandler:
            (# onSelect:
              (# theText: @StyledText;
                do theWindow[]->fileSelectionDialog->textFile.name;
                textFile.openRead;
                textFile.scan(# do ch->theText.put #);
                theText[]->theTextEditor.contents.contents;
                textFile.close;
              #)#);
            open: (# do 'Open' -> name #);
          #);
        saveItem: @menuItem
          (# eventhandler:
            (# onSelect:
              (# theText: @Text;
```

```

do textFile.openWrite;
    theTextEditor.contents.contents
    ->textFile.puttext;
    textFile.close;
    #)#);
    open::< (# do 'Save' -> name #);
#);
quitItem: @menuitem
    (# eventhandler::
        (# onSelect:: (# do Terminate #) #);
        open::< (# do 'Quit' -> name #);
    #);
open::<
    (#
    do 'File' -> name;
        openItem.open; openItem[] -> append;
        saveItem.open; saveItem[] -> append;
        quitItem.open; quitItem[] -> append;
    #)#);
    open:: (# do fileMenu.open; fileMenu[] -> append #);
#);
thetextEditor: @textEditor
    (# open::
        (#
        do theWindow.size->Size;
            True->bindBottom->bindRight
        #)#);
    open:: (# do thetextEditor.open #);
#);
do theWindow.open;
#)

```

The following three screen snapshots show how this application appear on the screen after the program has loaded its own source text for editing, and with the menu opened.



This document contains a tutorial on the use of version 1.4 of the Lidskjalv user interface framework.

These libraries are collectively referred to as the Lidskjalv user interface framework. Lidskjalv consists of a number of libraries, of which `guienv`, `controls`, `fields`, `scrolllists` and `figureitems` will be described in this tutorial. Most Lidskjalv applications will only be using some of these libraries.

The `controls` library offers the capabilities of interface controls, such as buttons, scrollbars, etc.

The `fields` library offers the capabilities for displaying bitmaps, rasters and advanced text editing.

The `figureitems` library offers fairly advanced graphics capabilities, including maintenance of graphical objects on the screen, which can be selected and dragged.

The `scrolllists` library offers facilities for making scrolling lists as used in e.g. the file dialog.

Besides the basic user interface libraries, as described above, the Lidskjalv framework contains a series of utility libraries (not described in this manual). These utility libraries can be found in the `utils` subdirectory of the Lidskjalv directory tree.

This tutorial will contain screen dumps mainly from the Windows 95 and Windows NT platforms. All demos can be recompiled on the other supported platforms, resulting in similar windows, just with the look-and-feel of that platform.

Further Readings

This tutorial is accompanied with a reference manual for the Lidskjalv framework: *Mjølner BETA System: Lidskjalv: User Interface Framework - Reference Manual*, MIA 94-27. User's manuals for the Mjølner BETA System on the different platforms are also available. Furthermore, a reference manual for the BETA is available: *Mjølner BETA System: Compiler Reference*, MIA 90-2. Finally, a reference manual is available on the basic libraries: *Mjølner BETA System: Basic Libraries*, MIA 91-8. The reader is advised to consult these documents (along with the other Mjølner BETA Manuals) as well as this tutorial.

1 Structure of a Lidskjalv application

A Lidskjalv application is usually structured along the following lines:

```
ORIGIN '~beta/guienv/v1.4/guienv'  
--- program: descriptor ---  
guienv  
(# ... declarations ...  
do ... initializations ..  
#)
```

The `ORIGIN` specification informs the compiler, that this program is utilizing the `guienv` library. The `'~beta/guienv/v1.4/guienv'` specifies that the BETA compiler is expected to find the fragment `guienv` on the disk in the subdirectory of the `guienv` directory, which is supposed to be located in the directory where the Mjølnir BETA System is located. `v1.4` is the version number of Lidskjalv and it must be consistent with the version of Lidskjalv installed on your system. Please also note that the `guienv` fragment must be located in the specified directory. If this is not the case, change the above directory specification.

If you want selectively to use some of the other Lidskjalv libraries (`fields`, `control`, `scrollingList` or `figureItem`), the libraries must be specified in `INCLUDE` clauses. E.g. to utilize both the `fields` and `control` libraries, the program must look like:

```
ORIGIN '~beta/guienv/v1.4/guienv';  
INCLUDE '~beta/guienv/v1.4/fields';  
INCLUDE '~beta/guienv/v1.4/controls';  
--- program: descriptor ---  
guienv  
(# ... declarations ...  
do ... initializations ...  
#)
```

Note that the main part of a Lidskjalv application contains an inserted instance of the `Guienv` pattern (not to be confused with the `Guienv` library). This `Guienv` pattern is taking care of all event handling of events originating from the underlying window system (e.g. mouse button events, window refresh events, keyboard events, etc.) such that Lidskjalv application programmers does not have to be concerned with managing the global event loop. Each user interaction (e.g. menu selections) result in execution of some specific actions of some BETA objects (details later).

1.1 Lidskjalv Declarations

The *declarations* part of a Lidskjalv application contains declaration of patterns, objects, and declaration of specializations of user interface objects such as menus, windows, buttons, etc. Most of the functionality of Lidskjalv applications will in fact

be specified in these specializations, since activation of most of the functionality will originate from the user manipulating items in the user interface.

1.2 Lidskjalv Initialization

The *initializations* part of a Lidskjalv application is primarily concerned with initialization of objects and with the creation and initialization of the various menus, windows, buttons, etc. The structure of Lidskjalv applications is such that the main part of the application is normally not concerned with invoking the functionality of the application, since that is usually the result of the user manipulating the user interface.

1.3 Lidskjalv Event Handling

Events (e.g. window refresh events, mouse button events, keyboard events) must be taken care of by the Lidskjalv application some way or another. The approach taken is to handle the global event loop for the programmer. When specifying interface objects¹ in a Lidskjalv program, the underlying implementation takes care of propagating events to the appropriate interface object. Events, (e.g. mouse button) will be converted into invocation of virtual procedure patterns of interface objects. These virtual procedure patterns (e.g. `onMouseDown` and `onRefresh`), must be extended by the Lidskjalv application programmer to contain the proper response to the specific event. That is, interface objects define various virtual procedure patterns that specify the types of events that are relevant for this type of interface object. In Lidskjalv programs, the programmer creates specializations of interface objects with further bindings for the virtual procedure patterns with the proper response to those events. During the discussion of the various types of interface objects, we will be discussing more details of this event handling.

1.4 Application Suspend, Resume and Terminate

Lidskjalv applications will continue to be executing until the application explicitly specifies that it may be terminated. Termination of a Lidskjalv application is done by executing the `Guienv` attribute `terminate` when termination is wanted. The result hereof is that the global event handling is immediately terminated, resulting in the termination of the execution of the entire application.

One final element of event handling needs to be presented here, namely the way in which background computation can be specified in Lidskjalv applications. Background computation is handled by the underlying window system by sending the application so-called idle events if no other events are waiting to be processed. The application can then be able to make proper background computation by responding to these idle events. Lidskjalv handles this by offering the application programmer an `onIdle` virtual event pattern which will be executed each time a idle event reaches

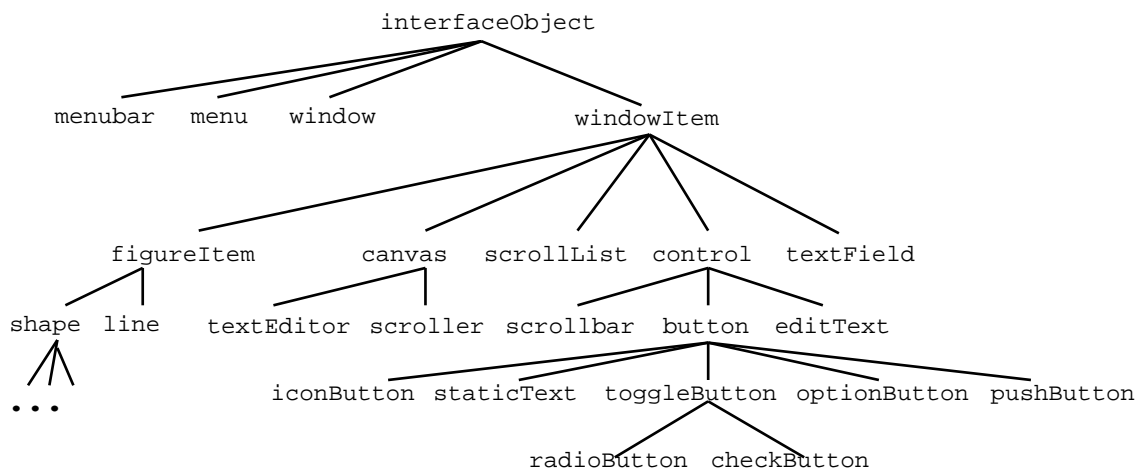
¹Interface objects are BETA objects that represent elements on the graphical user interface (e.g. a menu item, a button, etc.). Interface objects will be discussed in detail later.

the application (please note, that onIdle is not implemented in v1.4 on the Windows 95 and Windows NT platforms).

1.5 Overview of Lidskjalv

Lidskjalv contain many different patterns for implementing advanced (and simple) applications utilizing the graphical user interface system. It is impossible in this tutorial to give all details of these patterns and the presentation here will therefore only stress the most important patterns and the most important attributes of each pattern, along with illustrative examples.

Since there are many patterns and a somewhat elaborate pattern hierarchy, the following figure will show the most important classes and their super/subpattern relations.



2 Access to Global Structures

Lidskjalv offers access to several global objects of the window system, such as the mouse, the clipboard, the menubar, etc.

2.1 Access to the Mouse

The Mouse attribute in Guienv provides access to the physical mouse connected to the window system. Mouse.globalPosition returns the current position of the mouse (in screen coordinates). Mouse.buttonState returns the status of the mouse buttons and returns 1, 2, or 3 if the corresponding mouse button is currently pressed, and 0 otherwise.

2.2 Access to the Clipboard

The `clipboard` attribute in `Guienv` gives direct access to the underlying window system clipboard.

`clipboard.hasText` returns true, if the clipboard containing textual information. If `txt[]` is a reference to text object, then `txt[]->clipboard.textContents` places the text in `txt[]` onto the clipboard, and `clipboard.textContents->txt[]` copies the contents of the clipboard as text into `txt[]`. To clear the clipboard, you can invoke `clipboard.clearContents`.

2.3 Standard Input and Output

In `betaenv`, the standard input and output from the user is obtained through the `screen` object (or through the `putText`, `getText`, etc. operations of `betaenv`).

Obtaining input and output through `screen` should however be used sparsely in `Lidskjalv` since the facilities for input and output through `Lidskjalv` will conform to the user interface guidelines of the underlying window system and result in more elegant and powerful user interfaces.

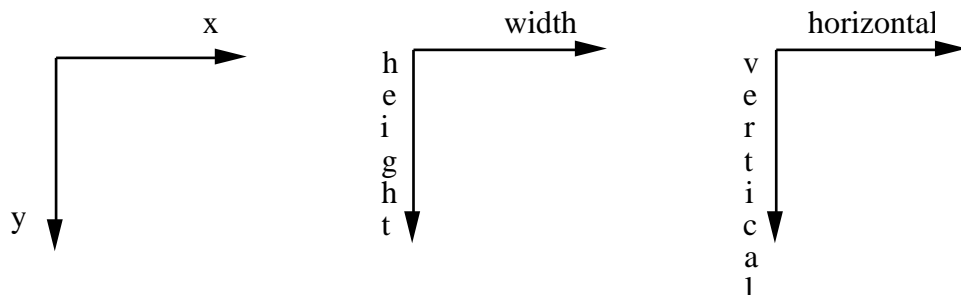
In window-based environments there usually are two ways to invoke applications: either directly from some sort of console window (e.g. `Xterm` on UNIX platforms, `MS-DOS Box` on Windows 95 and Windows NT, and `MPW Shell` on Macintosh platforms), or by double-clicking on some graphical icon in the graphical user interface. `Lidskjalv` behaves different in these two cases with respect to handling standard input and output.

In `Lidskjalv` applications invoked through console windows, standard input and output will be obtained from the console window from which the application is invoked. In `Lidskjalv` applications invoked through the graphical user interface, a console window will be created by the application, and standard input and output will be obtained from this console window.

2.6 Coordinate System

Many aspects of the programming in `Lidskjalv` involves specifying positions on the desk-top of the underlying window system (i.e. the position and size of a window). `Lidskjalv` defines a `point` and `rectangle` pattern for representing such properties.

The coordinate system used in the specification of these positions etc. are having a horizontal X-axis with X increasing to the right, and the Y-axis is vertical with Y increasing downwards. In defining e.g. the size of a window, the terms `width` and `height` are used along the X-axis, respectively the Y-axis.



The screen of the underlying window system has the (0,0) positioned at the upper left corner of the screen. Windows on the desk-top also has the (0,0) positioned in the upper left corner of the window. In general,, the (0,0) position is located in the upper left corner of all interface objects.

3 Windows

The window pattern describe properties of underlying window system windows. All Lidskjalv windows has a position on the screen, a height and width and a title (shown in the title bar). Windows may be visible on the screen or hidden. Finally, the window may specify the particular event handling to be associated with that window.

The `open` attribute initializes the window according to the window attributes. During the lifetime of a window, it may shift between being visible on the screen or not. This is controlled by the `show` and `hide` attributes. A window may take the control of the entire underlying window system (i.e. act as a modal window) if it is shown using `showModal` instead of `show`.

The following creates an ordinary window. The window will be initially visible:

```
ORIGIN '~beta/guienv/v1.4/guienv'
--- program: descriptor ---
guienv
(# simpleWindow: @window
  (# open::
    (#
      do (100, 100)->position;
        (300, 100)->size;
          'simpleWindow'->title
        #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #) #);
  #)
do simpleWindow.open
#)
```

**simple-
Window.bet**



**screendump
(Windows NT)**

Please note the `onAboutToClose:: (# do terminate #)` part of this small demo program. You will see this in all following demo programs. This small piece of code is included in these demo programs with the intent to make the demo applications terminate, when the window is closed. In realistic applications, this form of termination is very seldom used, since such applications often use a number of windows, and a Quit menu item to actually terminate the application..

3.1 Event Handling in Windows

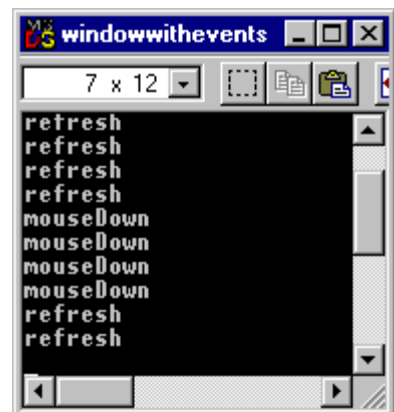
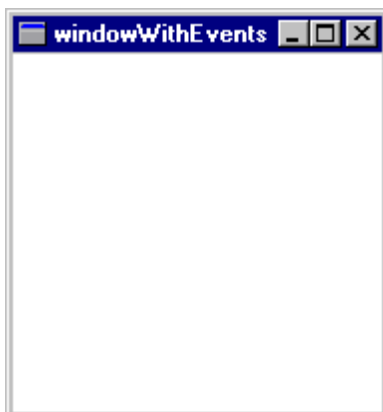
All user activities with the mouse, keyboard, etc. are turned into events by the underlying window system. Lidskjalv turns the window relevant events into invocations of virtual procedure patterns, defined in the eventhandler of the window. Eventhandler defines the following virtual patterns: `onIdle`, `onActivate`, `onDeactivate`, `onRefresh`, `onKeyDown`, `onMouseDown`, `onMouseUp`, and `onAboutToClose`.

These patterns are invoked when the corresponding event occurs and further bindings of these patterns may specify actions to be executed then. `onIdle` is invoked in the active window when nothing else happens in the underlying window system and can be used to specify some background computations, relative to this window (e.g. repagination). `onActivate` is invoked when this window is becoming the active window and `onDeactivate` is invoked when another window becomes the active window. `onRefresh` is invoked when the window has been corrupted (e.g. when the window is opened or when another window, which is obscuring parts of this window, is moved). `onKeyDown` is invoked when the user presses a key on the keyboard. `onKeyDown` takes one parameter which is the character associated with that key. `onMouseDown` is invoked when the user presses the mouse button and `onMouseUp` is invoked, when the user releases the mouse button. `onAboutToClose` is invoked when the mouse button is pressed when the cursor is located in the Close box. E.g.:

**window-
WithEvents.bet**

```
ORIGIN '~beta/guienv/v1.4/guienv'
--- program: descriptor ---
guienv
(# eventWindow: @window
  (# open::
    (#
      do (100,100)->position;
      (300,300)->size;
      'windowWithEvents'->title
    #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #);
        onIdle:: (# do 'idle'->putline #);
        onRefresh:: (# do 'refresh'->putline #);
        onMouseDown:: (# do 'mouseDown'->putline #)
      #)
  #)
do eventWindow.open;
#)
```

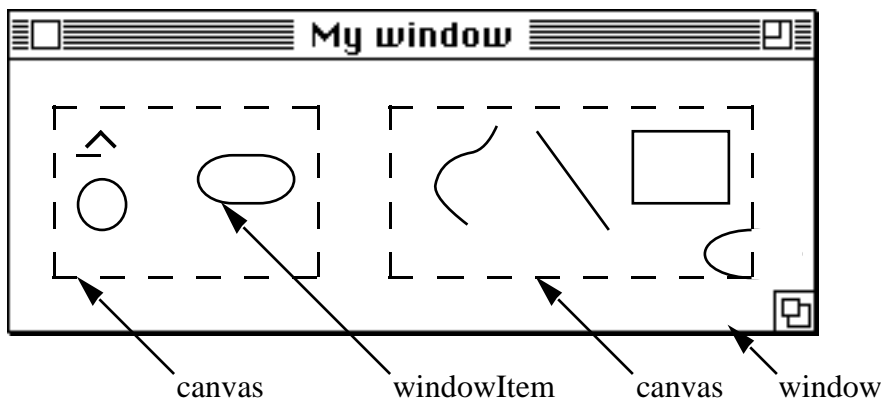
**screendumps
(Windows 95)**



The `refresh` and `mouseDown` printouts originates from user interaction with the window.

3.2 Contents of Windows

The window pattern in Lidskjalv offers several advanced facilities for specifying the contents of the window. These facilities are e.g. the `windowItem`, `Canvas` and `Scroller` patterns. `WindowItem` is the basic pattern for describing items to be displayed in a window, `canvas` (a subpattern of `windowItem`) is used to group `windowItems` of a window, and `scroller` enables scrolling of the window items. Several other subpatterns of `windowItem` exists, and will be discussed in later sections.



The window pattern defines one attribute, related to handling the window contents: `contents`. `Contents` is an operation, returning a `canvas` as exit parameter. `Window` displays all `windowItems` having `window.contents` as their father (explained later) and handles everything associated with refreshing the window contents in response to `windowItems` being associated with the window, and related to exposure of previously hidden parts of the window.

3.4 Window Items

As mentioned above, instances of `windowItem` are the elements displayed in a window. The `windowItem` pattern defines attributes common to all the different types of window items, defined in the different Lidskjalv libraries.

Window items are organized in a father-child hierarchy with respect to some `canvas` (or the `contents canvas` of a window) and all items have a father. The father defines the coordinate system for the children (e.g. the position of each child is relative to the position of the father, such that moving the father also moves the children). The `father` attribute of a window item refers to the `canvas` that this window item is a child of. `WindowItem` defines attributes for accessing and changing the position and size of the item and for controlling the visibility of the window item.

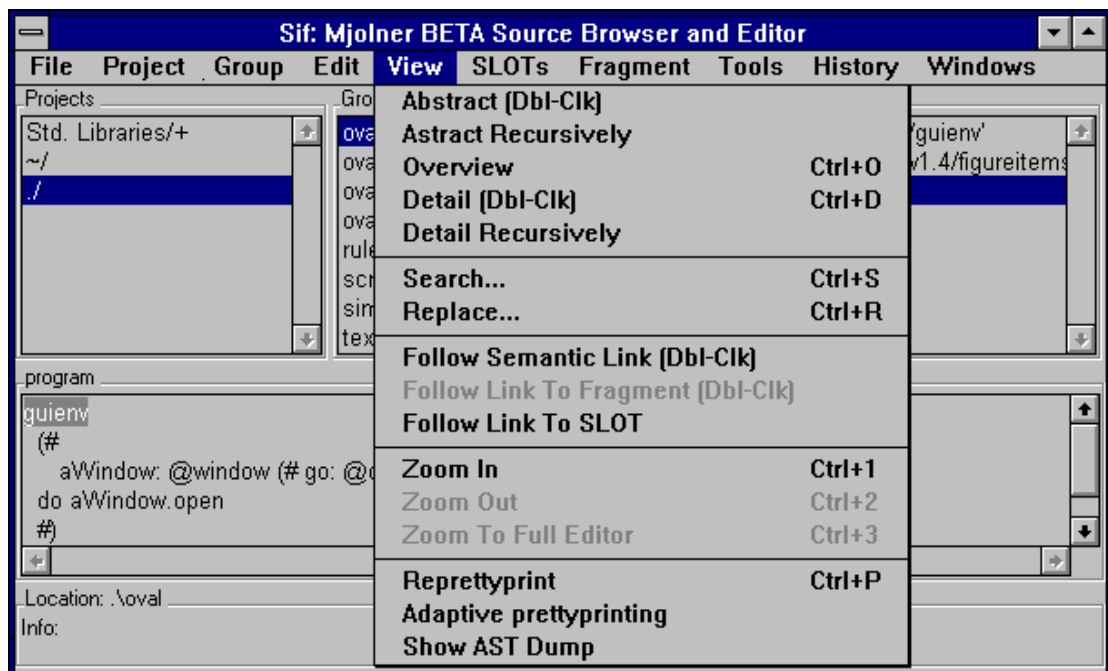
All window items are able to receive events from the user, and defines an event handler (similar to windows) to take care of these events. The `enable` and `disable` attributes are used to control whether the item will react to these events or not (i.e. a disabled window item will ignore e.g. `onMouseDown` events even though an event pattern is defined for `onMouseDown` event in that window item). The event

patterns defined in the event handler of window items are: OnFatherChanged, ChangedFrame, OnFatherChanged, onMouseDown, onMouseUp, onKeyDown, onEnableTarget, onDisableTarget, onRefresh, onActivate and onDeactivate. Specific behavior for these events can be specified by further binding the appropriate event pattern.

4 Menu Handling

Lidskjalv offers several facilities for dealing with menus. In Lidskjalv applications, menus can be of four types, namely pulldown and pop-up menus (and both menu types can be linear or hierarchical), where the linear and hierarchical pulldown menus are the most often used menu types. Pulldown menus are associated with the menubar that is located in the top of the window (or on the top of the screen on the Macintosh platform). Each pulldown menu has a title which is shown in the menubar. When that title is selected with the mouse, the pulldown menu is shown.

Screendump
(Windows NT)



Pop-up menus are under the control of the application programmer, who at any point in the program may specify that a particular menu must be popped up at a specific position (e.g. inside a window).

4.1 Basic Menu Facilities

The Menu pattern defined in Lidskjalv describes the facilities of any type of menu in Lidskjalv. Menu is a subpattern of interfaceObject. The menu handling is fully supported by Lidskjalv in the sense that the application programmer specifies the title and format of the menu (including layout of individual menu items, submenus, etc.) and specifies the actions, associated with the individual menu items.

The menu is inserted in the menubar and `Lidskjalv` handles all events associated with menus (e.g. when some menu item is selected, the proper actions are executed).

Menus may be enabled or disabled. A disabled menu is visible in the menu bar but its title is dimmed and it is impossible to pull the menu down from the menu bar.

If `theMenu` be an instance of `Menu`, initialization of `theMenu` may be done by further binding the `open` virtual procedure pattern attribute of `Menu`. In this further binding, the individual items in the menu are defined. The individual items are instances of the `menuItem` pattern (described below).

The menu may be used as a pull-down menu by inserting it in the menubar, e.g. by `THIS(Menu)[]->MenuBar.append`, or `theMenu[]->MenuBar.append`. The menu may also be used as a pop-up menu by invoking `(i,p,wi[])->THIS(menu).popUp` or `(i, p, wi[])->theMenu.popUp`. This will show the menu at position `p` with the menu element number `i` selected. `wi` is a reference to the window item in which the menu should pop-up. Finally, the menu may be used as a hierarchical menu. The menu can be inserted as a submenu of an item of another menu by `THIS(Menu)[]->anotherMenuItem.subMenu` or `theMenu[]->anotherMenuItem.subMenu`. The menu will then be a submenu of `anotherMenuItem`.

4.1.1 Menu Item Facilities

Each individual item in a menu is described by the `menuItem` pattern, defined locally to the `Menu` pattern in `Guienv`. Most facilities of `menuItem` deals with describing the format of the menu item. `Name` makes it possible to specify the name of the item. `Key` makes it possible to control the keyboard equivalent, and finally, `SubMenu` makes it possible to control the submenu of this item. `Checked` is used to control whether or not this menu item should be checked (a toggle menu item). The position of the item in the menu is examined by `position`.

Items in menus may be enabled (e.g. it is possible to select this menu item) or disable (e.g. the menu item cannot be selected and the menu item is dimmed in the menu). Enabling and disabling of menu items are controlled by the virtual procedure pattern `onStatus`. The application programmer must further bind `onStatus` in menu items in order to specify dynamic changes in the selectability of menu items.

When a menu item is selected, some actions must be executed. This is specified using the virtual event pattern `onSelect`. The application programmer must further bind `onSelect` in order to specify the actions to be executed as the result of this menu item being selected.

Menu items are initialized in two steps. The title of the menu item is first specified by giving a text string as the name operation of `MenuItem`. This is usually done in the `open` virtual in the menu item. Then the menu item is appended to the menu.

The menu items are numbered from the top of the menu, starting with 1, and menu separators are numbered too. Menu separators are specified by using the separator pattern, e.g.

```
&separator[]->sep[];          sep.open;          sep[]->
>animalMenu.append
```

Let us look at a small example:

```
ORIGIN '~beta/guienv/v1.4/guienv'
--- program: descriptor ---
guienv
(# menuWindow: @window
  (# menubarType:
    (# animalMenu: @menu
      (# iCat: @menuItem
```

menus.bet

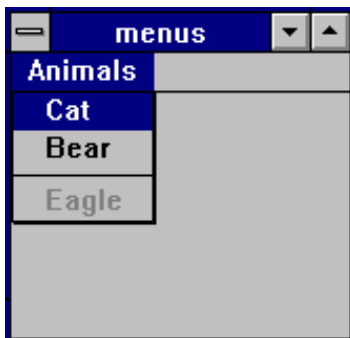
```

        (# eventhandler::
          (# onSelect:: (# do 'Cat chosen'->putline #) #);
          open:: (# do 'Cat'->name #)
        #);
iBear: @menuItem
  (# eventhandler::
    (# onSelect:: (# do 'Bear chosen'->putline #)#);
    open:: (# do 'Bear'->name #)
  #);
iEagle: @menuItem
  (# eventhandler::
    (# onSelect:: (# do 'Eagle chosen'->putline #);
    onStatus:: (# do false->value #)
    #);
    open:: (# do 'Eagle'->name #)
  #);
open::
  (# sep: ^menuItem
  do 'Animals'->name;
  iCat.open; iBear.open; iEagle.open;
  iCat[]->animalMenu.append;
  iBear[]->animalMenu.append;
  &separator[]->sep[]; sep.open; sep[]->animalMenu.append
  iEagle[]->animalMenu.append
  #)
  #);
  open:: (# do animalMenu.open; animalMenu[]->append #)
  #);
eventhandler::
  (# onAboutToClose:: (# do terminate #) #);
  open:: (# do 'menus'->title #)
  #)
do menuWindow.open
#)

```

animalMenu is a menu with three items and one separator. The title of animalMenu is Animals and the three menu items have the titles Cat, Bear, and Eagle:

screen dumps
(Windows NT)



When either Cat or Bear is selected, the title of the item will be printed. The Eagle item is disabled (shown dimmed) and cannot be selected.

4.1.2 Static and Dynamic Menu Items

Menu items are either constantly associated with the same actions during the entire execution of the program as described above (i.e. *static menu items*), or they may be associated with different actions during the execution of the program (i.e. *dynamic menu items*). For that reason, Lidskjalv contains two different menu item patterns: menuItem and dynamicMenuItem. MenuItem (described above) describes the static menu items and dynamicMenuItem (a subpattern of menuItem) describes the dynamic menu items. Since static menu items were the subject of the previous

section, we will here concentrate on the additional properties of dynamicMenuItems.

DynamicMenuItem is a subpattern of menuItem, and the dynamics of dynamic menu items is controlled by attaching and detaching so-called menuActions to the menu item during the execution of the program. MenuAction is a pattern defined in the menu pattern and defines two attributes: noStatus and onSelect with the same purpose as the onStatus and onSelect attributes of a static menu item. That is, by specializing the onSelect attribute, the actions of the menuAction are specified, and the noStatus attribute controls whether the menuAction is enabled or not.

DynamicMenuItem defines only two new attributes: attach and detach. Attach takes a menuAction as enter parameter and attaches it to the menu item. The result hereof is that then the noStatus attribute of the menu item is invoked, the onStatus attribute of the attached action is invoked instead, and invocation of the onSelect attribute of the menu item will result in invocation of the onSelect attribute of the attached action. The attached action is in this way becoming the behavior of the dynamic menu item. By changing the action associated with a dynamic menu item during the execution of the program, different behaviors may be associated with one particular dynamic menu item. If the dynamic menu item executes a detach, the action is detached and the menu item becomes disabled.

E.g.:

```

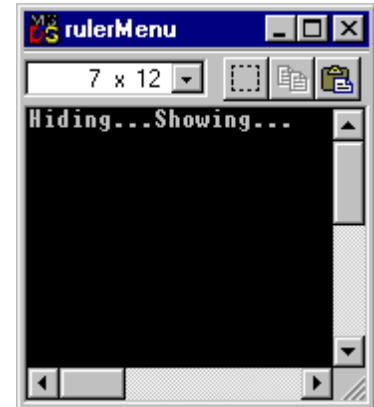
ORIGIN '~beta/guienv/v1.4/guienv'
--- program: descriptor ---
guienv
(# rulerWindow: @window
  (# menubarType::
    (# rulerMenu: @menu
      (# iHideRuler: @dynamicMenuItem
        (# open:: (# do 'Hide ruler'->name #) #);
      iShowRuler: @dynamicMenuItem
        (# open:: (# do 'Show ruler'->name #) #);
      hideRuler: @menuAction
        (# onSelect::
          (#
            do 'Hiding...'->puttext;
            iHideRuler.detach;
            showRuler[]->iShowRuler.attach
          #)
        #);
      showRuler: @menuAction
        (# onSelect::
          (# do 'Showing...'->puttext;
            iShowRuler.detach;
            hideRuler[]->iHideRuler.attach
          #)
        #);
      open::
        (#
          do 'Rulers'->name;
          iHideRuler.open; iHideRuler[]->rulerMenu.append;
          hideRuler[]->iHideRuler.attach;
          iShowRuler.open; iShowRuler[]->rulerMenu.append;
        #)
      #);
      open:: (# do rulerMenu.open; rulerMenu[]->append #)
    #);
  eventhandler::
    (# onAboutToClose:: (# do terminate #) #);
  open:: (# do 'rulerMenu'->title #)
#)

```

rulerMenu.bet

```
do rulerWindow.open
#)
```

screen.dumps
(Windows 95)



5 The Menu Bar

The `menuBar` attribute of `Guienv` is the interface to the underlying window system `menubar`. `menubar.Clear` removes all menus from the `menubar`. If `theMenu` is a menu (discussed earlier), `theMenu[]->menubar.append` inserts `theMenu` as the last menu in the `menubar` and menus are removed from the `menubar` by `theMenu[]->menubar.delete`.

5.1 Standard Menus

Most window systems have user interface guidelines that defines that the two first menus of any applications must be: **File** and **Edit**. It is also often the case that the **File** menu contains at least **New**, **Open**, **Close**, **Save**, **Save As**, **Revert**, **Print**, **Page Setup** and **Quit**, and that the **Edit** menu contains at least **Undo**, **Cut**, **Copy**, **Paste** and **Clear**.

To make it easy to create such menus, Lidskjalv contains a pattern `standardMenuBar`, containing two menu definitions: `standardFileMenu` and `standardEditMenu`, with exactly those menu item described above (as dynamic meny items).

These menu items are realized by instances of `dynamicMenuItem` with related names (e.g. `saveMenuItem` for the **Save** item). The actions to be associated with the individual items are specified by attaching an `menuItemAction` to the menu item in question, e.g.

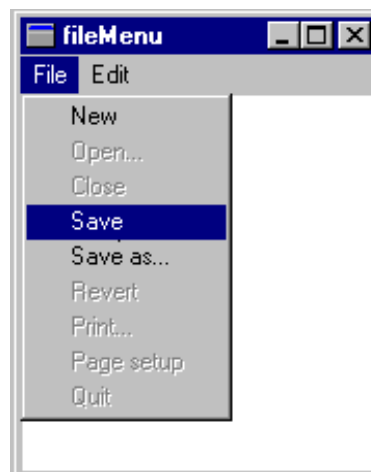
```
anMenuItem[]->theFileMenu.saveMenuItem.attach
```

E.g.:

```
ORIGIN '~beta/guienv/v1.4/guienv'
--- program: descriptor ---
guienv
(# fileMenuWindow: @window
  (# menubarType:: standardMenubar
    (# fileMenu:: standardFileMenu
      (# newMenuItem: @menuItem
        (# onSelect:: (# do 'New...'->putline #) #);
        saveMenuItem: @menuItem
        (# onSelect:: (# do 'Saving...'->putline #) #);
        saveAsMenuItem: @menuItem
        (# onSelect:: (# do 'Saving As...'->putline #) #);
        open::
          (#
            do newMenuItem[]->newMenuItem.attach;
            saveMenuItem[]->saveMenuItem.attach;
            saveAsMenuItem[]->saveAsMenuItem.attach
          #)
      #);
    editMenu:: standardEditMenu
  #);
  eventhandler::
    (# onAboutToClose:: (# do terminate #) #);
    open:: (# do 'fileMenu'->title #)
  #)
do fileMenuWindow.open
#)
```

fileMenu.bet

Note that the open attribute, further bound in fileMenu is automatically invoked on the theFileMenu instance during the initialization of Guienv.



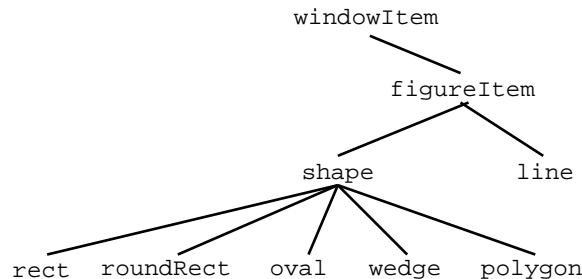
**Screendumps
(Windows 95)**

6 Graphics

As described above, the contents of windows (including ordinary graphics) are controlled by attaching instances of (subpatterns of) windowItems to the window (or to some canvas attached to the window). This section will describe the simple figure items.

6.1 Figure Items

As described above, the responsibility for the contents of an instance of window relies on the programmer. To aid the programmer in making graphics, Guienv defines a number of patterns for drawing lines, rectangles (with sharp or round corners), ovals, wedges and polygons.

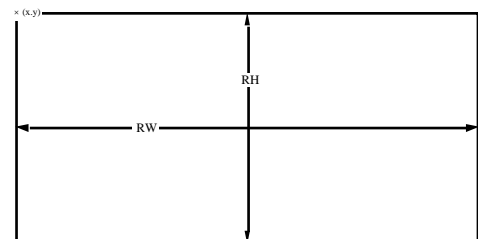


FigureItem is a subpattern of windowItem and inherits as such all its functionality (including the event handling possibilities) and defines the basic properties that are shared by all figure items. A figure item have a pen to be used for drawing the item. The pen defines attributes for defining the drawing pattern, the foreground and background colors, and the size of the pen (a rectangle).

Line is a straight line and defines the attributes `start` and `end` for accessing and changing the end-points of the line.

Shape is used as superpattern to all figure items that are defined by means of a rectangle and can be filled. Shape contains a `fill` attribute which defines the facilities for filling the shape (i.e. the tile pattern and the foreground and background colors to be used for the fill).

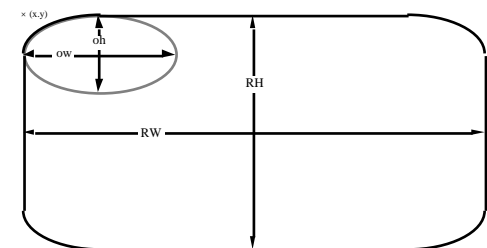
Rect is a real figure item in the sense that it is extending all figureItem procedures such that an instance of `rect` can be properly drawn in a window.



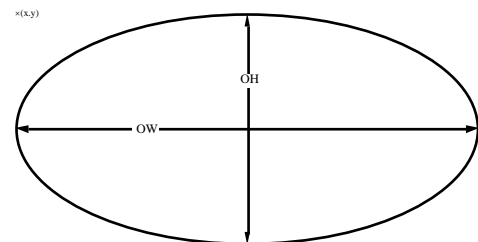
RoundRect is like `rect`, except that it also defines `roundness` to takes two integers, defining the round corners by defining the width and height of the oval in the corners. If R is an instance of `roundRect`, then

`(OW, OH) -> R.roundness`

defines the rectangle seen here.



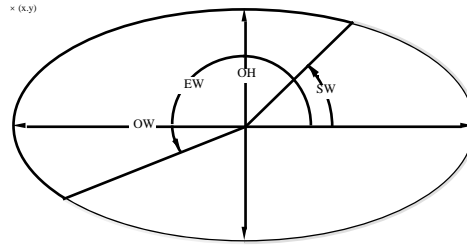
Oval is also like `rect` except that an oval is drawn.



Wedge is like oval, except that it also defines `startAngle` and `endAngle` which takes one integers, defining the start and end angles of the arc. If `A` is an instance of wedge, then

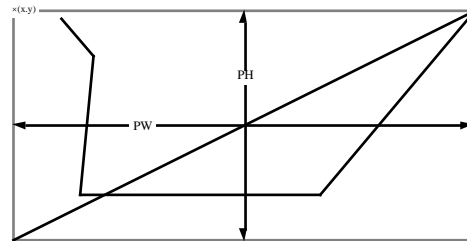
```
SW -> A.startAngle
EW -> A.endAngle
```

defines the wedge seen here.



Polygon is a figure item that consists of a collection of connected line segments. The points defining the polygon is specified by:

```
p: [6] ^point
do &point[]->p[1][]; (3,3) ->p[1];
&point[]->p[2][]; (5,7) ->p[2];
&point[]->p[3][]; (4,45) ->p[3];
&point[]->p[4][]; (30,45) ->p[4];
&point[]->p[5][]; (44,3) ->p[5];
&point[]->p[6][]; (1,55) ->p[6];
go.open;
p[]->go.points
```



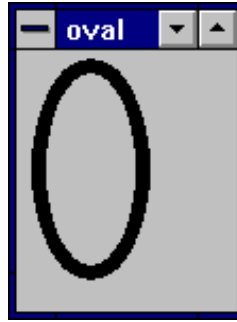
Drawing using figure items is as simple as:

```
ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/figureitems'
--- program: descriptor ---
guienv
(# aWindow: @window
  (# go: @oval
    (# open::
      (#
        do (100, 100)->position;
          (50, 100)->size;
          7->pen.size
        #)
      #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #) #);
    open::
      (#
        do (100, 100)->position;
          (300, 300)->size;
          'oval'->title;
          go.open
        #)
      #)
  do aWindow.open
  #)
```

oval.bet

This creates a window with an oval, positioned in (100, 100) and with the oval drawn using a 7x7 sized pen.

**Screendump
(Windows NT)**



6.2 Interactive Graphics Facilities

Interactive graphics in window is handled through the definition of event handlers of the `windowItems` attached with the window. The actual event handling of the window items (realizing that they are clicked, the mouse is entering them, etc) is handled entirely by Lidskjalv. The refresh of window items are also handled entirely by Lidskjalv. Please note, that these interactive facilities applies for all subpatterns of `windowItem` (i.e. not only for subpatterns of `figureItem`). Figure items are merely used here for demonstrative purposes.

6.2.1 Selection of Graphics Objects

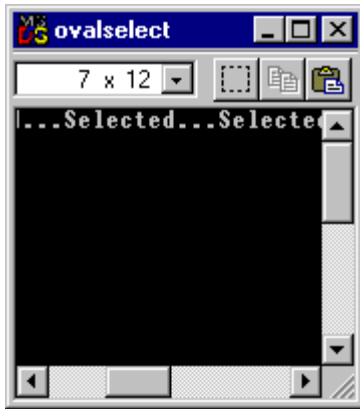
Selection of a `windowItem` is realized by specializing the `onMouseDown` event pattern in the appropriate `windowItem`. E.g.

ovalSelect.bet

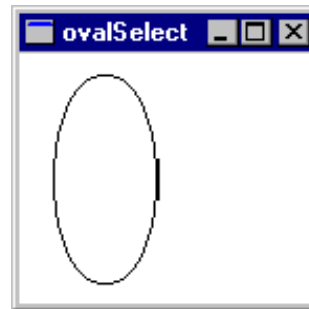
```

ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/figureitems'
--- program: descriptor ---
guienv
(# selectWindow: @window
  (# go: @oval
    (# open::
      (#
        do (100, 100)->position;
          (50, 100)->size;
        #);
      eventhandler::
        (# onMouseDown:: (# do 'Selected...'->puttext #) #)
      #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #) #);
    open::
      (#
        do (100, 100)->position;
          (300, 300)->size;
          'ovalSelect'->title;
          go.open
        #)
      #)
  do selectWindow.open
  #)

```

screen.dumps
(Windows 95)



6.2.2 Dragging of Graphics Objects

Dragging is specified using the drag pattern of windowItem. Dragging of e.g. a oval can be specified as follows:

```

ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/figureitems'
--- program: descriptor ---
guienv
(# dragWindow: @window
  (# go: @oval
    (# open::
      (#
        do (100, 100)->position;
        (50, 100)->size;
        7->pen.size
      #);
      eventhandler::
        (# onMouseDown:: (# do drag #) #)
    #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #) #);
    open::
      (#
        do (100, 100)->position;
        (300, 300)->size;
        'ovalDrag'->title;
        go.open
      #)
    #)
do dragWindow.open
#)

```

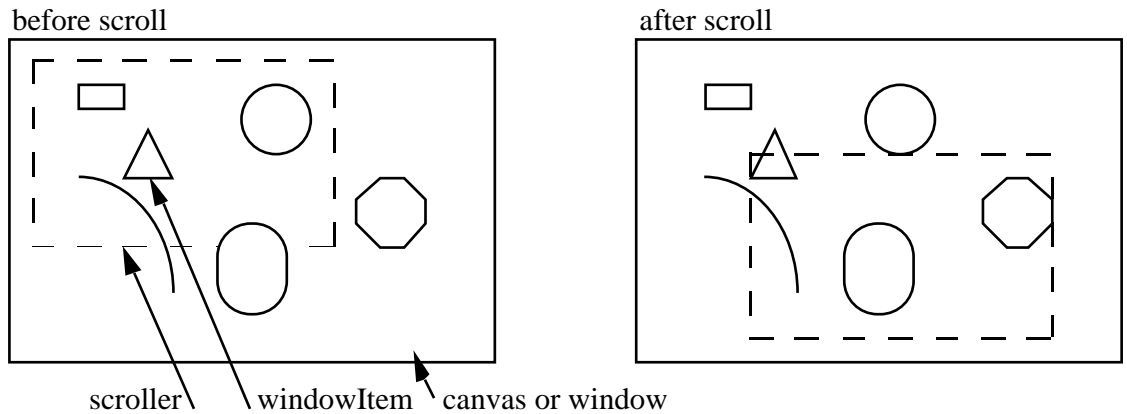
ovalDrag.bet

6.3 Canvas

Canvases are used for grouping other window items together to form a single unit such that e.g. moving the canvas inside the window moves all the window items attached to the canvas. Canvas is a subpattern to windowItem.

6.4 Scroller

A scroller is a special kind of windowItem that has scrollbars associated with it. The purpose of the scroller is to act as a viewport, restricting the visibility of the window items attached to the scroller. The purpose of the scrollbars is to enable this viewport to be scrolled to another position:



Scrolling is realized through three patterns: `abstractScroller`, `textEditor` and `scroller`. `AbstractScroller` implements the general scrolling facilities with scrollbars etc. It defines a virtual pattern, `contentsType`, which defines the type of `windowItem` to be controlled by the `abstractScroller`. The two other scrollers then further bind this virtual to `text` and `canvas`, respectively.

The following is the previous example with an associated scroller:

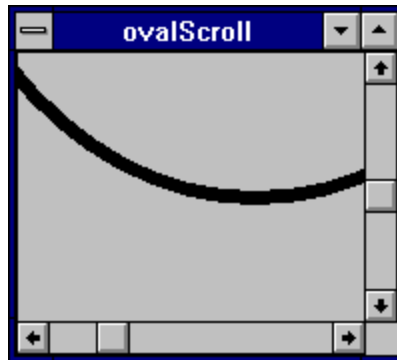
ovalScroll.bet

```

ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/figureitems';
INCLUDE '~beta/guienv/v1.4/fields'
--- program: descriptor ---
guienv
(# scrollWindow: @window
  (# scroll: @scroller
    (# contentsType:
      (# go: @oval
        (# open::
          (#
            do (100, 100)->position;
            (350, 450)->size;
            7->pen.size
          #);
          eventhandler::
            (# onMouseDown:: (# do drag #) #)
          #);
          open:: (# do go.open #)
        #);
        open::
          (# do true->bindBottom->bindRight #)
        #);
        eventhandler::
          (# onAboutToClose:: (# do terminate #) #);
        open::
          (#
            do (100, 100)->position;
            (300, 300)->size;
            'ovalScroll'->title;
            scroll.open;
            size->scroll.size
          #)
        #)
      #)
    #)
  #)
#)

```

```
do scrollWindow.open
#)
```



*screendump
(Windows NT)*

Please note the use of

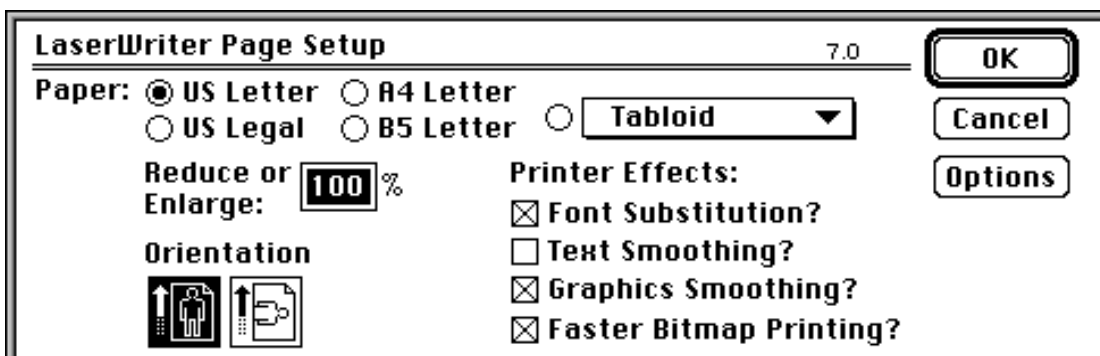
```
true -> bindBottom -> bindRight
```

in the definition of `scroll`. This informs Lidskjalv that `scroll` should extend towards the bottom and right if its enclosing window is resized. This facility is available for all `windowItems`.

7 Controls and Dialog Boxes

One of the most efficient ways to obtain structured information from the user is by presenting him with a dialog box in which he may enter text, select items from a list, check choices, etc. Lidskjalv enables the construction of such dialog boxes through the window pattern. Dialogs may be either modal or modeless. A modal dialog box will take over the entire control of the underlying window system, restricting the user only to interact with the dialog box until it is removed from the screen. A modeless dialog box, on the other hand, allows the user to choose also to interact with the other windows and menus on the screen while the dialog box is on the screen. Dialogs are constructed by a window, and either shown using either `showModal` or `show`.

Dialog boxes consists of the dialog box window and a number of control items in that window. Control items are either static text, editable text, buttons, check boxes, radio boxes or icons, along possibly with other graphics (e.g. figure items). The control items are used to specify the various options, that the user has to choose among in order to fill-in the requested information.

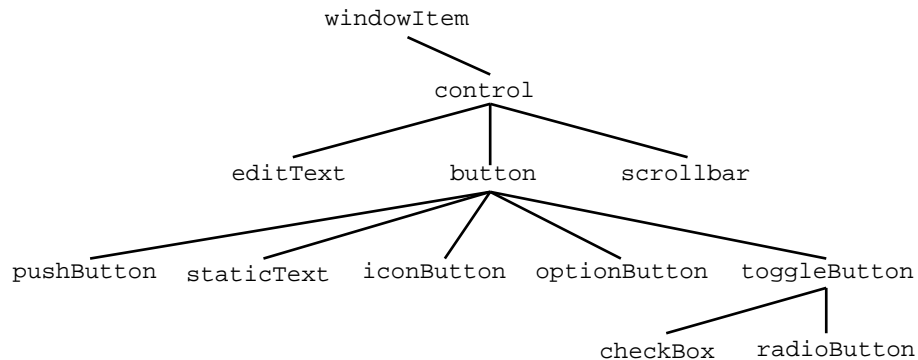





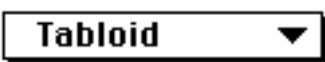

*screendump
(Macintosh)*

7.1 Control Items

Control items are the bread and butter of dialogs. Control items exist in seven different forms (all subpatterns of `windowItem`), namely text labels, text fields (editable), buttons, check boxes, radio boxes, icons and pictures. These control items are defined as subpatterns of the control pattern.

The hierarchy of control items are:



Control pattern name	Image (Macintosh)	Description
scrollbar		Used for various scrolling purposes.
staticText	Paper:	Used to specify permanent text in the dialog (usually explanatory text).
editText		Used to allow the user to enter some text.
pushButton		A button is used to specify some actions to be taken.
optionButton		Used to specify a button with associated pop-up menu.
checkBox	<input checked="" type="checkbox"/> Font Substitution? <input type="checkbox"/> Text Smoothing?	A check box is usually used together with other check boxes to present the user with a group of non-exclusive options.
radioButton	<input checked="" type="radio"/> US Letter <input type="radio"/> US Legal	A radio box is usually used together with other radio boxes to present the user with a group of exclusive options.
iconButton		A icon is used to show a minor picture in the dialog.

The control pattern is a subpattern of `windowItem` and inherits as such all its facilities (size, position, event handling, etc.). The actions to be associated with a control must be specified in further bindings of e.g. the virtual event pattern `onMouseDown`. Control defines some other facilities that can be ignored for most Lidskjalv applications.

Scrollbar defines various attributes for controlling the scrollbar (scrollAmount, maxValue and value). Besides the eventhandler, a new event patterns are defined: onThumbMoved. OnThumbMoved is invoked when the user moved the scroll thumb. The orientation (vertical or horizontal) is controlled through the vertical attribute, such that true->vertical specifies the scrollbar to be vertical (horizontal for false). Finally, the length of the scrollbar is controlled by the length attribute.

Button is also a subpattern of control and is the superpattern for the rest of the controls. The attributes of button are controlling the label (text and text style), associated with all buttons.

PushButton and staticText are simple subpatterns of button. IconButton is another simple subpattern of button, only defining one new attribute showLabel for controlling whether the label should be shown or not.

EditText is implementing a one-line text editor to be used for simple text specifications (file names etc.) The text style of the text is controlled by the style attribute, and the contents of the editText can be manipulated through the contents attribute. That is, if T is a text, and ET is a editText, then T[]->ET.contents sets the text shown in the editText control to the contents of T (i.e. setting the initial contents), and ET.contents->T[] copies the contents of the editText control into T (i.e. reading the user input).

OptionButton defines a field which will pop-up a menu in response to the user pressing the mouse button on top of the field. The attributes of optionButton controls the associated menu and the currently selected menu item (shown inside the field).

ToggleButton is the common superpattern for the RadioButton and CheckBox controls. ToggleButton controls a binary state. A series of RadioButtons are used for specifying a set of exclusive options, and one or more checkBoxes are used for specifying a set of non-exclusive options.

Finally, one of the buttons in the window can be specified to function as a default button (i.e. be activated by pressing carriage return) by entering a reference to it to defaultButton.

The following example illustrates the use of controls:

```

ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/controls'
--- program: descriptor ---
guienv
(# authorName: @text;
 isReport: @boolean;
 theDialog: @window
  (# ctitleLabel: @staticText
   (# open::
    (#
     do (10, 10)->position; (95,25)->size;
     'Title: '->label
    #) #);
  cTitle: @editText
  (# open::
   (# do (115, 10)->position; (150,25)->size #)
  #);
  cAuthorLabel: @staticText
  (# open::
   (#
    do (10, 40)->position; (95,25)->size;
    'Author: '->label
   #) #);
  cAuthor: @editText

```

dialog.bet

```

        (# open::
          (# do (115, 40)->position; (150,25)->size #)
        #);
cReport: @checkBox
  (# open::
    (#
      do (10, 70)->position; (100,25)->size;
      'Report'->label
    #);
    eventhandler::
      (# onMouseUp:: (# do not state->state #) #);
  #);
cCancel: @pushButton
  (# open::
    (#
      do (115, 70)->position; (80,25)->size;
      'Cancel'->label
    #);
    eventhandler::
      (# onMouseUp:: (# do theDialog.close #) #)
  #);
cOk: @pushButton
  (# open::
    (#
      do (115+150-30, 70)->position; (30,25)->size;
      'OK'->label
    #);
    eventhandler::
      (# onMouseUp::
        (#
          do (* store values, then *)
            theDialog.close
        #) #)
  #);
eventhandler::
  (# onAboutToClose:: (# do terminate #) #);
open::
  (#
    do (40,40)->position; (275,100)->size;
    'dialog'->title;
    cOk.open; cCancel.open;
    ctitleLabel.open; cTitle.open;
    cAuthorLabel.open; cAuthor.open;
    cReport.open;
    cOk[]->defaultButton
  #)
  #)
do theDialog.open;
  theDialog.showModal
#)

```

screendump
(Windows NT)



This defines a dialog with two buttons, two editable text fields, two static text fields, and one check box (all enabled).

8 Scroll Lists

Scroll lists are used to display an interface object in which the user is able to select elements from a list of elements (e.g. file names).

A `scrollList` maintains the list of elements and the user is allowed to scroll in the list or to select elements in the list by clicking on them. `ScrollList` has operations for inserting, deleting and scanning the elements in the list. Furthermore, `scrollList` maintains a list of the currently selected elements in the list.

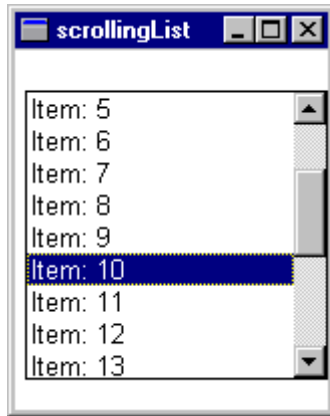
`TextScrollList` is a subpattern of `scrollList` for maintaining a list of text strings. `TextScrollList` defines additional operations for manipulating the text strings and for manipulation the text style of the elements in the list.

The following is an example of a `textScrollList` for selecting in a list of items:

```
ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/scrolllists'
--- program: descriptor ---
guienv
(# scrollListWindow: @window
  (# scrollList: @textScrollList
    (# open::
      (# tmpText: @text;
        windowSize: @point
        do (5,20)->position;
          20->append;
          (for inx: 20 repeat
            'Item: '->tmpText;
            inx->tmpText.putint;
            (inx,tmpText[])->setText
          for);
          this(scrollListWindow).size->>windowSize;
          position->>windowSize.subtract;
          (0,15)->>windowSize.subtract;
          windowSize->size;
          true->bindRight->bindBottom
        #);
      eventhandler::
        (# onSelect::
          (# do item->gettext->putline #)
        #)
      #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #) #);
    open::
      (#
        do 'scrollingList'->title;
        scrollList.open
      #)
    #)
do scrollListWindow.open
#)
```

scrollinglist.bet

screendumps
(Windows 95)



9 Window Fields

Lidskjalv offers facilities for defining more advanced fields than the above mentioned controls. These facilities include window items as two different text editing fields (`textField` and `textEditor`). These patterns are subpatterns of `windowItem` and inherits as such all its facilities.

`TextField` and `textEditor` are both advanced text editors offering the usual text editing facilities, such as fonts, cut/copy/paste, selections, etc. along with simple text manipulation functions. All event handling is automatically taken care of by the patterns. `TextEditor` is only special by offering scrolling facilities.

`TextField` handles text selection through the `selection` attribute. `selection.start` contains the character position of the first character in the selection and `selection.end` contains the character position of the last character in the selection. If `selection.start = selection.end`, then nothing is selected, and `selection.start` identifies the position of the text cursor. `selection.contents` returns the text in the selection. `ScrollIntoView` will make sure that the current selection is visible.

The text editing facilities are `cut`, `copy` and `paste`, that implements the usual cut/copy/paste functionality. `insert` takes a text as parameter, and inserts it immediately before the current selection, and `delete` deletes the text of the current selection.

To enable scanning the entire text in the text field, the `scanText` operation is defined. `ScanText` is a control pattern that takes two positions as parameters, and iterates over the characters in the text editor between the two positions. During the scan, `ch` will contain the current character in the text.

The text contents of the text field is accessed through the `contents` attribute that can be used for getting a copy of the current text in the text field.

The simplest possible 'Hello World' `textField` can be specified as follows:

`textField.bet`

```
ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/fields'
--- program: descriptor ---
guienv
(# textWindow: @window
  (# txtField: @textField
    (# open::
```

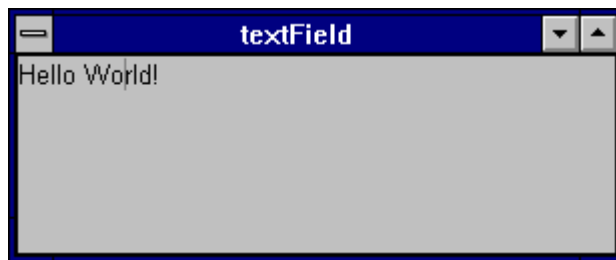


```

        (# t: @styledText;
        do (0, 0)->position;
          (300, 100)->size;
          'Hello World!'->t;
          t[]->contents
        #)
      #);
    eventhandler::
      (# onAboutToClose:: (# do terminate #) #);
    open::
      (#
      do (20, 100)->position;
        (300, 100)->size;
        'textField'->title;
        txtField.open
      #)
    #);
do textWindow.open
#)

```

This will create a `textField` at position (20, 100) and with size (300, 100). The initial contents of the `textField` is 'Hello World!'. All usual text editing facilities will be available in the editor:



*screendump
(Windows NT)*

By replacing `textField` by `textEditor` and subtracting (15,15) from the size of the `myTextField` will result in a window with a text editor with scrolling facilities. The reason for subtracting (15, 15) from the size of the editor field is to make room for the scrollbars at the right and bottom of the window.

```

ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/fields'
--- program: descriptor ---
guienv
(# textWindow: @window
  (# txtEdit: @textEditor
    (# open::
      (# t: @styledText;
      do (-1, -1)->position;
        (287, 87)->size;
        'Hello World!'->t;
        t[]->contents.contents;
        true -> bindRight;
        true -> bindBottom;
      #)
    #);
  eventhandler::
    (# onAboutToClose:: (# do terminate #) #);
  open::
    (#
    do (20, 100)->position;
      (285, 85)->size;
      'textEditor'->title;
      txtEdit.open
    #)
  #);

```

textEditor.bet

```

        #)
    #);
do textWindow.open
#)

```

*screendump
(Windows NT)*



Note, that the only visible difference, compared with the previous textField example is that a text editor automatically has both vertical and horizontal scroll bars.

10 Standard Dialogs

Lidskjalv contains a number of standard dialogs, including file selection dialogs. The attribute `fileSelectionDialog` will activate the standard file dialog and return the path name of the selected file:

fileDialog.bet

```

ORIGIN '~beta/guienv/v1.4/guienv';
INCLUDE '~beta/guienv/v1.4/stddialogs';
--- program: descriptor ---
guienv
(# name: ^text;
do fileSelectionDialog(# do 'fileDialog'->Title[] #)->name[];
  (if name[]=NONE then
    'Selected Cancel' -> putline;
  else
    name[] -> putline;
  if);
terminate;
#)

```

*screendump
(Windows NT)*

