1.1 ActionWindow

```
@interface ActionWindow : Window
{
  CommandList *commandBindings;
}

- init;
- free;
- bindCommand: (ULONG) command withObject: anObject
     selector: (SEL) aSel;
- findCommandBinding: (ULONG) command;
- (MRESULT) execCommand: (ULONG) command;

@end
```

1.2 AutoCheckBox

```
@interface AutoCheckBox : Button
{
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;

@end
```

1.3 AutoRadioButton

```
@interface AutoRadioButton : Button
{
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;

@end
```

1.4 AutoTriStateButton

```
@interface AutoTriStateButton : Button
{
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;

@end
```

```
- init: (ULONG) aKey data: (void *) aData;
- free;

- insert: (CommandList *) element;
- (int) compare: (CommandList *) elem1
          with: (CommandList *) elem2;

- find: (ULONG) aKey;

- setKey: (ULONG) aKey;
- setData: (void *) aData;
- setNext: (CommandList *) element;

- (ULONG)  key;
- (void *) data;
- next;

@end
```

1.9 Container

```
@interface Container : Window
{
  ULONG              createFlags;
  CONTAINER_MINIREC *recordBuffer;
  FIELDINFO         *columnBuffer;
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
        in: (Window *) parent;
- addColumn: (char *) aTitle;

- insertObject: anObject;
- insertObject: anObject withTitle: (const char *) aTitle;
- insertObject: anObject withTitle: (const char *) aTitle
      andIcon: (ULONG) anIcon;

- arrange;
- iconView: sender;
- nameView: sender;
- textView: sender;
- treeView: sender;
- detailView: sender;

- (ULONG) records;
- object;

- (CONTAINER_MINIREC *) firstRecord;
- (CONTAINER_MINIREC *) lastRecord;
- (CONTAINER_MINIREC *) nextRecord;
- (CONTAINER_MINIREC *) previousRecord;
- (CONTAINER_MINIREC *) firstSelected;
```

```
- copySelection;
- cutSelection;
- pasteSelection;

- (BOOL) changed;
- (BOOL) readOnly;

- setReadOnly;
- setReadWrite;
- setTextLimit: (SHORT) limit;

@end
```

1.11 FileDlg

```
@interface FileDlg : Object
{
  FILEDLG  fileDlg;
}

- init;
- initForOpen: (const char *) aTitle
   withFilter: (char *) aFilter;
- initForSaveAs: (const char *) aTitle
     withFilter: (char *) aFilter;

- setTitle: (char *) aTitle;
- setFilter: (char *) aFilter;
- setFlags: (ULONG) aFlags;
- setOKTitle: (const char *) aTitle;

- (ULONG) runModalFor: sender;

- (char *) fileName;

@end
```

1.12 Frame

```
@interface Frame : Window
{
}

@end
```

1.13 ListBox

```
@interface ListBox : Window
{
}
```

1.17 NoteBook

```
@interface NoteBook : Window
{
}

@end
```

1.18 PushButton

```
@interface PushButton : Button
{
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;

@end
```

1.19 RadioButton

```
@interface RadioButton : Button
{
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;

@end
```

1.20 ScrollBar

```
@interface ScrollBar : Window
{
}

- initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;

- (SHORT) position;
- (SHORT) lowerBound;
- (SHORT) upperBound;

- setPosition: (SHORT) position;
- setScrollbar: (SHORT) position
    withBounds: (SHORT) lower : (SHORT) upper;
- setThumbSizeForVisible: (SHORT) visible
                      of: (SHORT) all;

@end
```

1.25 StdDialog

```
@interface StdDialog : ActionWindow
{
  id      delegate;
  ULONG   result;
  BOOL    running;
}

- initWithId: (ULONG) anId;
- loadMenu;
- free;

- delegate;
- setDelegate: aDelegate;
- (ULONG) result;

- makeKeyAndOrderFront: sender;
- runModalFor: sender;
- dismiss: sender;

- (MRESULT) handleMessage: (ULONG) msg
            withParams: (MPARAM) mp1 and: (MPARAM) mp2;

@end
```

1.26 StdWindow

```
@interface StdWindow : ActionWindow
{
  HWND       frame;
  id         delegate;
}

- initWithId: (ULONG) anId;
- initWithId: (ULONG) anId andFlags: (ULONG) flags;
- free;

- setSize: (LONG) x : (LONG) y : (LONG) w : (LONG) h;
- setRect: (LONG) w : (LONG) h;

- (LONG) framexoffset;
- (LONG) frameyoffset;
- (LONG) framewidth;
- (LONG) frameheight;

- (HWND) frame;
- delegate;
- setDelegate: aDelegate;

- setTitle: (char *) aTitle;
```

```
- createObjects;
- insertChild: aChild;
- insertSibling: aSibling;
- findFromID: (ULONG) anId;
- findFromHWND: (HWND) aHwnd;

- (char *) text: (char *) buffer;
- (int) textLength;
- setText: (char *) buffer;
- setSize: (LONG) x : (LONG) y : (LONG) w : (LONG) h;
- setRect: (LONG) w : (LONG) h;
- size: (PSWP) aSize;

- (LONG) width;
- (LONG) height;
- (LONG) xoffset;
- (LONG) yoffset;

- (HWND) window;
- (ULONG) pmId;

- enable;
- disable;
- activate;
- deactivate;

- invalidate;
- show;
- hide;

- (MRESULT) handleMessage: (ULONG) msg
            withParams: (MPARAM) mp1 and: (MPARAM) mp2;

@end
```

Instance Variables:

CommandList * commandBindings;

This variable stores a list of all command bindings set up for a certain instance of `ActionWindow` or one of its subclasses.

Methods:

- init;

The instance method `init` initializes the instance variable `commandBindings` to `nil`.

- free;

`free` frees the memory allocated for the list of command bindings.

- bindCommand: (ULONG) command withObject: anObject selector: (SEL) aSel;

`bindCommand: withObject: selector:` sets up a new command binding. `command` is the command identifier, which normally is the identifier of the sender of the command (Pushbutton, Menuitem, ...). `anObject` is the *Target*, `aSel` the selector[1] of the *Action*.

An Action must be of the form `nameOfMethod: sender`. Only these methods can be called by `execCommand`. Actions should return `nil` on successful execution, a non-`nil` value otherwise.

- findCommandBinding: (ULONG) command;

This method is used for checking, if a command binding for `command` has been set up previously. `findCommandBinding:` returns `nil`, if no command binding for `command` has been set up, a non-`nil` value otherwise.

- (MRESULT) execCommand: (ULONG) command;

`execCommand:` searches for the command binding for `command` and executes the corresponding *Action* in the set up *Target*, if one was found.

2.2 AutoCheckBox

Inherits from: Button : Window : Object

Class description:

The class `AutoCheckBox` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

For a short description of an instance of this class see table 2.1 on page 26. Figure 2.1 on page 27 shows an instance of this class. See the description of the class `Button` for access methods.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes a newly created instance of `AutoCheckBox`. Using this class and method is similar to creating a Button object while specifying the flag `BS_AUTOCHECKBOX`.

[1] The selector of a method can be queried via `@selector (...)`

Flag	Description
BS_PUSHBUTTON	The created Button will be a Pushbutton.
BS_CHECKBOX	The Button will be a Checkbox.
BS_AUTOCHECKBOX	The Button will be an AutoCheckbox, this one toggles it's state every time the user clicks on the Button.
BS_RADIOBUTTON	The Button will be a Radiobutton. In contrast to Checkboxes, a dot appears if the Button is checked.
BS_AUTORADIOBUTTON	In addition to a normal Radiobutton an AutoRadiobutton automatically unchecks all other Radiobuttons in the same group if it is checked.
BS_3STATE	A Tri-state Button has an additional check state, which is called *indeterminate*.
BS_AUTO3STATE	same as AutoCheckbox, but Tri-state Button.
BS_USERBUTTON	The button created will be an application-defined button. It has to be drawn by the application when a `BN_PAIN` message is received by the parent window.

Table 2.1: Main Button styles used to define the type of Button

Figure 2.1: This figure shows (from left to right) the following Buttons types: *Pushbutton*, *Radiobutton*, *Checkbox* and *Tri-state Button*.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

Using this Initializer the Programmer can create a new Button in an existing parent window. `anId` is the PM id of the button to be created, `flags` specify the creation flags for the Button control (`BS_xxxx` and `WS_xxxx` constants). `parent` is the parent window of the newly created Button, which normally is either an instance of `StdDialog` or `StdWindow`.

After creation of the Button the size can be set via `setSize::::` and the text to be displayed via `setText:`.

Association to an existing PM Button Window should be done by using `associate:`.

A newly created Button Object is not automatically inserted as a child window of it's parent. Use `[parent insertChild: button]` where `parent` is the parent window and `button` is the newly created Button Object.

Flag	Description
BS_NOCURSORSELECT	The Radiobutton is not selected when it is given the focus from keyboard actions.

Table 2.2: Button styles which can be combined with an AutoRadiobutton

Flag	Description
CBS_SIMPLE	If this flag is specified, the entry field *and* the listbox are visible at any time.
CBS_DROPDOWN	This flag causes the listbox only to be displayed when requested by the user.
CBS_DROPDOWNLIST	CBS_DROPDOWNLIST should be used, if the only valid entries in the entry field are items already shown in the listbox. Here the listbox is only displayed on user demand, and the entry field displays one of the listbox items. It is not editable.

Table 2.5: Combo box control styles

2.6 CheckBox

Inherits from: Button : Window : Object

Class description:

The class `CheckBox` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes a newly created instance of `CheckBox`. Using this class and method is similar to creating a Button object while specifying the flag `BS_CHECKBOX`.

2.7 ComboBox

Inherits from: ListBox : Window : Object

Class description:

`CombobBox` is a class designed to provide an interface to OS/2 PM windows of class `WC_COMBOBOX`.

The only method implemented specifically for this class is `initWithId: andFlags: in:`.

A `ComboBox` consists of a `EntryField` and a `ListBox`. Access to the text in the `EntryField` is provided via `setText:` and `text:`. The items in the `ListBox` can be accessed by using the inherited methods of the superclass `ListBox`.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

Using this method, a previously allocated instance of `ComboBox` is initialized. `anId` is the OS/2 window identifier of the object to be initialized, `flags` are the window flags to use (see table 2.5), which can be a combination of one of the flags special to *Combo boxes* or general window flags (e.g. `WS_VISIBLE`).

Flag	Description
CCS_AUTOPOSITION	If this flag is specified, whenever necessary, the container automatically repositions the items displayed in the container.
CCS_MINIRECORDCORE	This flag specifies that the information stored in the container should consist of items of the datatype `MINIRECORDCORE`. This flag *must* be specified at the moment.
CCS_READONLY	Specifying this flag causes all items in the container to be read-only. If you want only some of the information displayed to be read-only, use `setColumnTitleAttributes:` or `setColumnDataAttributes:`.
CCS_VERIFYPOINTERS	Setting this flag ensures that all application pointers used are members of a linked list stored internally in the container. This flag should not only be used for debugging purposes. Using this feature decreases response time of the container object.

Table 2.6: Global container creation styles

Flag	Description
CCS_SINGLESEL	Only selection of a single item is allowed.
CCS_EXTENDEDSEL	Enable extended selection of the container.
CCS_MULTIPLESEL	Enable multiple selection.

Table 2.7: Container creation styles concerning Selection

ULONG createFlags;

This instance variable is used to store the creation flags specified at initialization using `initWithId: andFlags: in:`. This will be used in later releases of this library to correctly store the state of the object in a stream to be able to load it afterwards.

CONTAINER_MINIREC * recordBuffer;

`recordBuffer` is used as an internal buffer variable. Many of the methods described later store temporary data in this variable.

FIELDINFO * columnBuffer;

Again, `columnBuffer` is an internal buffer variable used by some of the methods for querying *column information* in detail's view.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

Using this method, a newly created instance of `Container` gets initialized. The parameter `anId` is the PM identifier of the window, which is created. `parent` is a pointer to the parent window (normally an instance of `Window` or one of it's subclasses) of this instance. `flags` is used to pass creation flags to the container window. The flags which can be used in addition to the standard creation flags (e.g. `WS_VISIBLE`) can be logically grouped. Tables 2.6 and 2.7 shortly describe the flags specific to container controls.

This method causes the items currently stored in the container to be rearranged. This can be necessary after inserting some new items into the container.

If you plan to insert many items, don't call this method after inserting each of them, only after inserting all of them.

- iconView: sender;

Using this method sets the display mode of the container window to icon view. The parameter `sender` is ignored. It can be specified as `nil` or `self`.

- nameView: sender;

Using this method sets the display mode of the container window to name view. The parameter `sender` is ignored. It can be specified as `nil` or `self`.

- textView: sender;

Using this method sets the display mode of the container window to text view. The parameter `sender` is ignored. It can be specified as `nil` or `self`.

- treeView: sender;

Using this method sets the display mode of the container window to tree view. The parameter `sender` is ignored. It can be specified as `nil` or `self`.

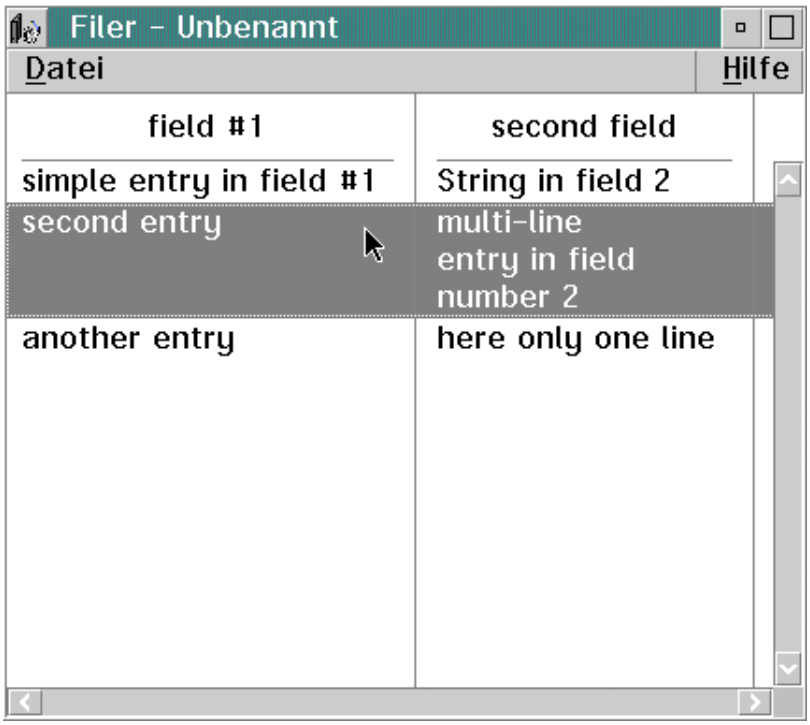

Figure 2.4: Container window in details view. The second record is selected.

- detailView: sender;

Using this method sets the display mode of the container window to detail view. The parameter `sender` is ignored. It can be specified as `nil` or `self`.

- (CONTAINER_MINIREC *) nextRecord;

This method queries the next record. The search *must* have been initialized using `firstRecord`.

As `firstRecord`, this method returns a pointer to the appropriate `CONTAINER_MINIREC` structure for the next record or `NULL`, if none exists.

- (CONTAINER_MINIREC *) previousRecord;

This method queries the previous record. The search *must* have been initialized using `lastRecord`.

As `lastRecord`, this method returns a pointer to the appropriate `CONTAINER_MINIREC` structure for the previous record or `NULL`, if none exists.

- (CONTAINER_MINIREC *) firstSelected;

Using the methods `firstSelected` and `nextSelected` all selected records can be queried one by one. Use this piece of code to visit all selected records:

```
.
.
  if ([container firstSelected]) {
    do {
      /* specific manipulations  */
      /* for each record go here */
    } while ([container nextSelected]);
  }
.
.
```

If no record is currently selected, `NULL` is returned, otherwise a pointer to the `CONTAINER_MINIREC` structure of the first selected record.

- (CONTAINER_MINIREC *) nextSelected;

`nextSelected` is the counterpart to `nextRecord`. It returns the `CONTAINER_MINIREC` structure of the next selected record. Before, the search must have been initialized using `firstSelected`.

If no more record is selected, `NULL` is returned.

- (BOOL) recordIsSelected;

This method returns `YES`, if the current record specified in `recordBuffer` is selected. Otherwise `NO` is returned.

- invalidateRecord;

After changing the data of an item, which could affect display, you should call this method. This causes the current record to be redisplayed if necessary.

This method *must* be called, if you decide to change some parameters in the `CONTAINER_MINIREC` structure of the current record other than the data in the object stored.

- invalidateSelectedRecords;

`invalidateSelectedRecords` works the same as `invalidateRecord`, but it extends invalidation to all selected records.

- hideRecord: sender;

Flag	Description
CFA_BITMAPORICON	If this flag is set, the title string should be really a handle of a bitmap or an icon. This bitmap is displayed instead of a title string. Because at the moment, only text strings are supported by this class, you should not specify this flag.
CFA_FITITLEREADONLY	Specifying this flag causes the title text to be read-only.

Table 2.8: Flags specified for Container column titles

Flag	Description
CFA_LEFT	This causes either the data or the title string to be aligned to the left.
CFA_CENTER	If specified, data or title string are horizontally centered.
CFA_RIGHT	Align data or title string to the right.
CFA_TOP	Top-align the data or title string.
CFA_VCENTER	Cause the data or title string to be vertically centered.
CFA_BOTTOM	Align the data or title string to the bottom.

Table 2.9: Container flags specifying alignment of data and title text

```
.
.
  if ([container firstColumn]) {
    do {
      /* specific manipulations  */
      /* for each column go here */
    } while ([container nextColumn]);
  }
.
.
```

- (FIELDINFO *) previousColumn;

This method queries information about the previous column. The search must have been initialized using `lastColumn`.

This part of code can be used to query and modify information for all existing columns, starting at the last one:

```
.
.
  if ([container lastColumn]) {
    do {
      /* specific manipulations  */
      /* for each column go here */
    } while ([container previousColumn]);
  }
.
.
```

- (char *) columnTitle;

This method restores the visibility state of a previously hidden column.

- showAllColumns: sender;

`showAllColumns:` shows all columns, including those previously hidden.

- (BOOL) columnIsHidden;

If the current column is hidden, `YES` is returned, otherwise `NO`.

- invalidateColumns;

After changing either the title attributes or the data attributes of a column, you should call `invalidateColumns` to cause the modifications to be redisplayed.

- setColumnTitleAttributes: (ULONG) attr;

`setColumnTitleAttributes:` is used to change the title attributes stored for the current column. After changing the attributes, don't forget to use `invalidateColumns` to cause the columns to be redisplayed correctly.

`attr` specifies, which flags should be set. The current value of the title attributes can be queried via `columnTitleAttributes`.

Table 2.8 on page 39 and table 2.9 on page 39 show all possible flags which can be specified.

See [3] for more information concerning title attributes settings.

- setColumnDataAttributes: (ULONG) attr;

`setColumnDataAttributes:` is the counterpart to `columnDataAttributes` and causes the data attributes of the current column to be set to `attr`. Possible flags which can be specified are shown in table 2.9 on page 39, table 2.10 on the facing page and table 2.11 on the preceding page.

After modifying any data, don't forget to call `invalidateColumns` to redisplay the modifications.

See [3] for more information concerning data attributes settings.

- select;

Calling this method selects the current record. The selection is only displayed after calling `invalidateRecord` or `invalidateSelectedRecords`.

- deselect;

Calling this method deselects the current record. The change in the selection state is only displayed after calling `invalidateRecord` or `invalidateSelectedRecords`.

- selectAll: sender;

`selectAll:` selects all records in the container. This also affects temporary hidden records, but does not display them. So be careful using this method when planning to modify all selected records afterwards.

- deselectAll: sender;

`deselectAll:` deselects all records in the container. This also affects temporary hidden records, but does not display them. So be careful using this ethod when planning to modify all selected records afterwards.

- sort: (ULONG) column;

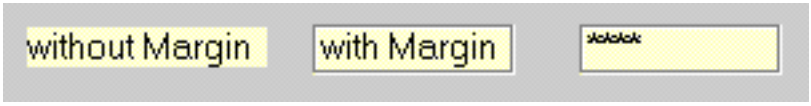

Figure 2.5: In this figure you can see (from left to right) an EntryField without a margin, one with a margin and an EntryField with margin and the style option BS_UNREADABLE

After creating the Entryfield the size can be set via `setSize::::` and the text to be displayed via `setText:`. Clearing the text of an Entryfield can be achieved calling `[entryfield setText: ""]`.

Association to an existing PM Entryfield Window should be done by using `associate:`.

A newly created EntryField Object is not automatically inserted as a child window of it's parent. Use `[parent insertChild: entryfield]` where `parent` is the parent window and `entryfield` is the newly created EntryField Object.

Table 2.12 (page 42) shows most of the available `ES_`xxxx flags used at creation of the EntryField.

In addition to these flags there's also another group of flags defining the encoding scheme for the text in the EntryField. These flags are only used when a double-byte encoding scheme is used for text.

Figure 2.5 on page 43 shows three possible forms of how an EntryField can look.

- (BOOL) changed;

`changed` returns `TRUE` if the text displayed in the EntryField has changed since the last call to this method, `FALSE` otherwise.

- (BOOL) readOnly;

By using this method the programmer can query if the EntryField is in *read-only* or in *read-write* mode. When *read-only* no characters can be typed into the EntryField.

This method returns `TRUE` if the EntryField is in *read-only* mode, `FALSE` otherwise (*read-write*).

- setReadOnly;

Calling this method activates the *read-only* mode of the EntryField.

- setReadWrite;

`setReadWrite` switches the EntryField to *read-write* mode.

- setTextLimit: (SHORT) limit;

By calling `setTextLimit:` the programmer can set the maximum number of characters which can be entered into the EntryField. `limit` is this maximum number of characters.

When querying the contents of the EntryField via `text:` the maximum number of characters returned is `limit + 1`, including the concluding `'\0x0'` at the end of the string.

2.11 FileDlg

Inherits from: OBJECT

Class description:

Flag	Description
FDS_APPLYBUTTON	Specifying this flag causes an additional *Apply* Button to be displayed. This is useful when the dialog is not run modal.
FDS_CENTER	This flag causes the dialog to be centered in its parent window.
FDS_ENABLEFILELB	When a Save As dialog is used and this flag is specified, the file listbox is enabled for selection. Otherwise, the user is not allowed to select an existing file.
FDS_HELPBUTTON	Display a *Help* Button. The button has the PM identifier `DID_HELP_PB`.
FDS_MULTIPLESEL	Allow multiple selection of file names.
FSD_OPEN_DIALOG	Create an open dialog.
FDS_PRELOAD_VOLINFO	Preload the volume info (volume name,...).
FDS_SAVEAS_DIALOG	Create a save as dialog.

Table 2.13: Window flags which can be specified for file dialogs

The only way to change the appearance of the dialog is to use `setFlags:`.

`aTitle` is the title text of the dialog, which can also be specified at a later time using `setTitle:`.

The parameter `aFilter` specifies a filter by which the files in the displayed directory are selected for display. This string can hold some of the special characters `*` and `?`, which here have the same meaning as using then as a filename argument to an OS/2 command.

So specifying `*.dat` as filter string would cause only filenames to be displayed which are ending in `.dat`.

- initForSaveAs: (const char *) aTitle withFilter: (char *) aFilter;

This method initializes a dialog for saving a file. The dialog is always centered in it's parent window, which is specified as a parameter to

`aTitle` is the title text of the dialog, which can also be specified at a later time using `setTitle:`.

The parameter `aFilter` specifies a filter by which the files in the displayed directory are selected for display. This string can hold some of the special characters `*` and `?`, which here have the same meaning as using then as a filename argument to an OS/2 command. `runModalFor:`.

The only way to change the appearance of the dialog is to use `setFlags:`.

So specifying `*.dat` as filter string would cause only filenames to be displayed which are ending in `.dat`.

- setTitle: (char *) aTitle;

Using this method, you can change the title string displayed in the dialog after initializing the object. `aTitle` is this title string.

- setFilter: (char *) aFilter;

This method is used to set the filter string `aFilter` after initializing the dialog.

- setFlags: (ULONG) aFlags;

`setFlags:` is the only possibility to change the appearance of the file dialog to other settings than the default (either `(FDS_OPEN_DIALOG | FDS_CENTER)` or `(FDS_SAVEAS_DIALOG | FDS_CENTER)`, depending on the initializing method used).

Flag	Description
LS_HORZSCROLL	This flags adds a horizontal Scrollbar to the Listbox window, if it is specified at creation.
LS_MULTIPLESEL	Normally only one item in the Listbox can be selected once. If this flag is set, multiple selection is enabled. Currently querying the multiple selection is not supported by methods of this class.
LS_EXTENDEDSEL	Specifying this flag enables the extended selection user interface of the Listbox window.
LS_OWNERDRAW	This flag tells the Listbox not to draw the items itself. Appropriate messages are sent to the owner of the listbox, which has to draw them.
LS_NOADJUSTPOS	This flag tells the listbox not to adjust the size and position of the window. If this flag is set, maybe only part of the first or last item shown is drawn.

Table 2.14: `LS_xxxx` styles used at creation of a Listbox window

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

`initWithId: andFlags: in:` can be used to create a Listbox window at runtime. The parameters are the same as those used in the appropriate method of the class `Button`.

Figure 2.7 on page 48 shows two forms of Listbox windows. The left is a standard Listbox with only one Scrollbar – a vertical one. The right Listbox also has a horizontal Scrollbar.

How a Listbox window appears depends on what control flags you specify in the parameter `flags`. Table 2.14 shows which control flags are possible and what effect is caused by specifiying them. One ore more of the flags can be specified. These flags must be binary or-ed using the | operator. If none of them should be used, `0L` should be given as `flags` parameter.

- insertItem: (SHORT) pos text: (char *) buffer;

Using this method you can insert a new item into the Listbox. `pos` is the position in the Listbox where the item shall be inserted. If `pos` is `LIT_END`, the item will be inserted as the last item in the Listbox.

`buffer` is the title of the item to be inserted. This string is shown afterwards in the Listbox at the specified position.

The first item in the Listbox is at position `0`, the last at `count - 1`.

- (SHORT) count;

`count` returns the number of items which are currently in the Listbox.

- (SHORT) selected;

`selected` returns the position of the selected item. If no item is currently selected, a value below `0` is returned.

Multiple selection is currently not supported by this class. If you want to query multiple selection you have to use the appropriate OS/2 API functions, or just wait untill the next version of this library is released.

Flag	Description
MLS_BORDER	This flag causes a border to be drawn around the MLE window
MLS_READONLY	Disable editing in the MLE window (*read-only* mode)
MLS_WORDWRAP	Enable word wrap
MLS_HSCROLL	Draw a horizontal scroll bar
MLS_VSCROLL	Draw a vertical scroll bar
MLS_IGNORETAB	If this flag is set, the MLE window ignores pressing the `TAB` key
MLS_DISABLEUNDO	Disable the *undo* function of the MLE window.

Table 2.15: `MLE_xxxx` styles used at creation of a MLE window

- initWithId: (ULONG) anId;

This method only calls `initWithId: andFlags:` of it's superclass `StdWindow` while specifying the window flags as shown above.

- initWithId: (ULONG) anId andFlags: (ULONG) flags;

This method only calls `initWithId: andFlags:` of it's superclass `StdWindow` while specifying the window flags as shown above.

2.15 Menu

Inherits from: Window : Object

Class description:

`Menu` is a class designed to provide an interface to OS/2 PM windows of class `WC_MENU`. Windows of these type are the *Actionbar* or simply whole menus.

The menu items not displayed are no windows on their own. They are created newly before they get displayed (when the menu they are in gets selected).

Methods:

- enableItem: (USHORT) identifier;

Calling this method, the menu item specified by `identifier` is enabled. After calling this method, the user can select this item.

- disableItem: (USHORT) identifier;

Calling this method, the menu item specified by `identifier` is disabled. After calling this method, the user is not able to select this item.

The menu item can be re-enabled using `enableItem:`

2.16 MultiLineEntryField

Inherits from: Window : Object

Class description:

Flag	Description
SBS_HORZ	This flags causes a horizontal scrollbar to be created.
SBS_VERT	Create a vertical scrollbar. Either this flag, or `SBS_HORZ` must be specified.
SBS_AUTOTRACK	As more information is displayed, the slider automatically scrolls.
SBS_AUTOSIZE	When this flag is specified, the size of the slider automatically changes to reflect the amount of data to be displayed.

Table 2.16: Flags which can be specified at scrollbar creation

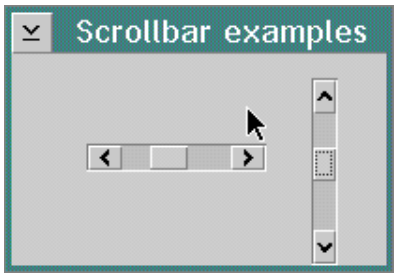

Figure 2.8: This figure shows a window containing a horizontal and a vertical scrollbar.

Class description:

The class `RadioButton` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

For a short description of an instance of this class see table 2.1 on page 26. Figure 2.1 on page 27 shows an instance of this class. See the description of the class `Button` for access methods.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes a newly created instance of `RadioButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_RADIOBUTTON`.

2.20 ScrollBar

Inherits from: Window : Object

Class description:

If more data is to be displayed in OS/2 PM windows or in window controls than would fit inside the control, scrollbars are used to let the user choose, which part of the data is to be shown.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes the object with the PM identifier `anId` in its parent window `parent`.

`flags` is used to specify creation flags for this window.

Figure 2.9 on the following page shows a horizontal and a vertical slider control.

2.22 SpinButton

Inherits from: Window : Object

Class description:

A Spinbutton is an entry field where only numeric values can be entered. The object provides to arrows, which allow the user to increment or decrement the value currently shown in the accompanying entry field.

Currently, only creation of a SpinButton object is supported.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes the object with the PM identifier `anId` in its parent window `parent`.

`flags` is used to specify creation flags for this window.

In figure 2.9 on the next page you can see an example of a spinbutton.

2.23 Static

Inherits from: Window : Object

Class description:

PM windows of this class are used to display static data (e.g. text or bitmaps) on the screen.

Currently, only creation of a Static object is supported.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes the object with the PM identifier `anId` in its parent window `parent`.

`flags` is used to specify creation flags for this window.

- (HAB) hab;

`hab` returns the *Handle Anchor Block* of the application.

2.25 StdDialog

Inherits from: ActionWindow : Window : Object

Class description:

Instances of this class are used to represent *OS/2 Dialog windows*. At the moment dialogs are loaded from a resource file. This also initializes all controls (`Buttons`, `EntryFields`,...) in the dialog which are defined in the resource file.

Dialogs can be run *modal* for a given window, which means, while the dialog is active, no actions can be processed in the specified parent window, or *not modal*, where dialogs behave just like normal OS/2 PM main windows.

Figure 2.10 shows a simple dialog window.

Instance Variables:

id delegate;

`delegate` stores the handle of the *delegate object* of the dialog. Any events not processed by methods of this class are forwarded to the *delegate*.

See also *Methods implemented by the delegate*.

ULONG result;

After a dialog is dismissed (closed), the result of the dialog is stored in the instance variable `result`. This result can be queried by using the instance method `result`.

BOOL running;

When a dialog is run either modal or not modal, this variable is set to `YES`. When the dialog is dismissed again, it is set back to `NO`.

This variable is used as a flag to prevent one instance of a dialog to be run only once at a given time.

Methods:

- initWithId: (ULONG) anId;

`initWithId:` loads a dialog resource from the main resource file, which is linked into the executable file. `anId` is a key value, which uniquely identifies the dialog to be loaded in the resource file.

This method returns `self` if successful, `nil` otherwise.

- loadMenu;

If the loaded dialog shall contain an *Application menu*, the menu must be explicitly loaded from the resource file by calling this method. The menu resource is assumed to have the same resource identifier as the dialog window itself.

`loadMenu` returns `self`.

- windowDidMove: sender;

After a window has been successfully moved, the delegate method `windowDidMove:` gets called.

- windowDidResize: sender;

`windowDidResize:` gets called after resizing a dialog. The newly achieved size of the window can be queried by sending the window (`sender`) appropriate messages (`width`, `height`).

- windowDidResizeFrom: (LONG) oldX : (LONG) oldY to: (LONG) newX : (LONG) newY : sender;

`windowDidResizeFrom:: to:::` is just the same as the previously described method `windowDidResize:`. In contrast to this method, `windowDidResizeFrom:: to:::` also sends the old (`oldX`, `oldY`) and new (`newX`, `newY`) width and height of the resized window.

These values can be directly used without querying the width and height of the window via `[sender width]` and `[sender height]`.

It can also be useful for some special purposes to know the width and height of the window before the process of resizing it. These parameters cannot be queried by using any of the methods of `sender`.

- windowWillClose: sender;

This function gets called if the `StdDialog` is about to close. If this function returns a non-`nil` value or the `delegate` object doesn't implement this method, the window will be closed.

If – otherwise – the `delegate` returns `nil`, closing the window is stopped and the normal execution of the program continues.

`sender` is a pointer to the sending instance of `StdDialog`.

- buttonWasPressed: (ULONG) buttonId : sender;

Everytime a `WM_COMMAND` message is received by `handleMessage: withParams: and:` from a Pushbutton, this message is sent to the delegate of the `StdDialog`.

`buttonId` is the OS/2 PM ID of the Button sending the `WM_COMMAND` message. `sender` is a pointer to the sending instance of `StdDialog`.

This method should return `nil` if the button event could be handled, a non-`nil` value otherwise.

- menuWasSelected: (ULONG) menuId : sender;

Analogous to `buttonWasPressed::` this delegate method is called whenever a menu item gets selected by the user.

`menuWasSelected::` should return `nil` if the menu selection could be processed successfully, a non-`nil` value otherwise.

- commandPosted: (USHORT) origin : sender;

Every time a command was posted and it could not be processed by `buttonWasPressed::` or `menuWasSelected::`, or if one of these methods or both are not implemented by the window delegate, or the command does not result from a button or a menu item, this delegate method is called.

`commandPosted::` should return `nil`, if the event could be processed successfully, a non-`nil` value otherwise.

- sysButtonWasPressed: (ULONG) buttonID : sender;

Class description:

An instance of this class is a simple OS/2 PM Window, consisting of a *frame window* and a *client window*. It is possible to load resources like an *Icon*, a *Menu Bar* or an *Accelerator Table*.

Normally there's only one *StdWindow* in an application, showing and handling the application's Menu Bar and some default informations.

All messages of interest can be captured by an object called the delegate of the window. This object can then react to these messages. Normally there's no need to subclass this class.

Figure 2.11 shows a `StdWindow` containing a menu bar.

For information about simplifying creation of a `StdWindow` see the class description of `MainWindow`.

Instance Variables:

HWND frame;

The instance variable `frame` is used to store the window handle of the *frame window*, where the inherited variable `window` is used to store the handle of the *client window*.

id delegate;

This variable is used to store a pointer to the delegate object of this window.

Methods:

- initWithId: (ULONG) anId;

This method is used to initialize an instance of the class `StdWindow`.

`anId` is the PM identification number of the window.

This method creates the frame window and the client window. The client window is an instance of the OS/2 PM-class `WINDOW_CLASS`. (Note the difference between Objective C classes and OS/2 PM-classes!)

The frame window handle is stored in `frame`, the client window handle in `window`.

The title of the window can be set via `setTitle:`.

- initWithId: (ULONG) anId andFlags: (ULONG) flags;

This method is used to initialize an instance of the class `StdWindow`. In contrast to `init:id:` you can specify some *frame creation flags* to specify the resources to be loaded.

`flags` can be a combination of `FCF_MENU`, `FCF_ICON` and `FCF_ACCELTABLE`. `FCF_MENU` tells the object, that a Menu Bar should be loaded. The *resource id* of the Menu Bar must match the parameter `anId`. `FCF_ICON` is used to specify an Application Icon to be loaded and shown, whereas `FCF_ACCELTABLE` loads an Accelerator Table.

You should also specify the type of border to be drawn for the window. This can either be `FCF_SIZEBORDER` for a resizable border or `FCF_BORDER` for a normal border. A thin border can be created by specifying `FCF_THINBORDER`.

If you, for example, want to load a Menu Bar and an Icon you have to specify `FCF_MENU | FCF_ICON` as flags.

- free;

`handleMessage: withParams: and:` gets called by the default window procedure for the OS/2 PM-class `WINDOW_CLASS`.

This function evaluates the type of message received and reacts by calling a `delegate` method, if implemented (see "Functions implemented by the delegate").

If the received message is of type `COMMAND` or `SYS_COMMAND`, and a command binding for the command identifier has been set up, the corresponding *Action* in the set up *Target* gets called. (see class `ActionWindow`)

If the corresponding `delegate` function could not be found, `handleMessage: withParams: and:` of its precessor in the class hierarchy is called.

Methods implemented by the delegate:

- windowDidMove: sender;

After a window has been successfully moved, the delegate method `windowDidMove:` gets called.

- windowDidResize: sender;

`windowDidResize:` gets called after resizing a window. The newly achieved size of the window can be queried by sending the window (`sender`) appropriate messages (`width`, `height`).

- windowDidResizeFrom: (LONG) oldX : (LONG) oldY to: (LONG) newX : (LONG) newY : sender;

`windowDidResizeFrom:: to:::` is just the same as the previously described method `windowDidResize:`. In contrast to this method, `windowDidResizeFrom:: to:::` also sends the old (`oldX`, `oldY`) and new (`newX`, `newY`) width and height of the resized window.

These values can be directly used without querying the width and height of the window via `[sender width]` and `[sender height]`.

It can also be useful for some special purposes to know the width and height of the window before the process of resizing it. These parameters cannot be queried by using any of the methods of `sender`.

- windowWillClose: sender;

This function gets called if the `StdWindow` is about to close. If this function returns a non-`nil` value or the `delegate` object doesn't implement this method, the window will be closed.

If – otherwise – the `delegate` returns `nil`, closing the window is stopped and the normal execution of the program continues.

`sender` is a pointer to the sending instance of `StdWindow`.

- buttonWasPressed: (ULONG) buttonId : sender;

Everytime a `WM_COMMAND` message is received by `handleMessage: withParams: and:` from a Pushbutton, this message is sent to the delegate of the `StdWindow`.

`buttonId` is the OS/2 PM ID of the Button sending the `WM_COMMAND` message. `sender` is a pointer to the sending instance of `StdWindow`.

This method should return `nil` if the button event could be handled, a non-`nil` value otherwise.

- menuWasSelected: (ULONG) menuId : sender;

Analogous to `buttonWasPressed::` this delegate method is called whenever a menu item gets selected by the user.

Class description:

The class `TriStateButton` is a subclass of `Button`. It's only purpose is to simplify creating a PM Button window for a special purpose.

For a short description of an instance of this class see table 2.1 on page 26. Figure 2.1 on page 27 shows an instance of this class. See the description of the class `Button` for access methods.

Methods:

- initWithId: (ULONG) anId andFlags: (ULONG) flags in: (Window *) parent;

This method initializes a newly created instance of `TriStateButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_3STATE`.

2.29 ValueSet

Inherits from: Window : Object

Class description:

`ValueSet` is a class designed to provide an interface to OS/2 PM windows of class `WC_VALUESET`.

At the moment no additional functionality to it's superclass *Window* has been added. Special support for OS/2 PM Valueset windows will be added in the future.

2.30 Window

Inherits from: Object

Class description:

`Window` is an abstract superclass for all classes representing some kind of window (e.g. an Entryfield, a StdWindow or a Dialog).

This class should never be instantiated. It doesn't provide enough functionality to be really useful. It can be compared to the Objective C root class `Object`, it's the root class for all PM windows.

Only PM Windows with minimal functionality should be associated directly with instances of this class (e.g. Static Texts, Pushbuttons, ...).

Instance Variables:

HWND window;

`window` is an OS/2 PM window handle. It stores the handle of the PM window associated with an instance of this class.

Window * child;

This variable points to the first *child window* of this window.

Window * sibling;

`sibling` points to the first *sibling window* of this window.

The window text is copied into `buffer`, which must be large enough to hold all of the text, and buffer, or a pointer to the newly allocated area is returned.

The length of the window text can be queried via `textLength`.

- (int) textLength;

This method returns the number of characters the window text consists of. Don't forget to allocate an extra byte for the *End-of-String*-character before using `text:`.

- setText: (char *) buffer;

`setText:` is used to set the window text to a new string. This string is stored in `buffer`.

- setSize: (LONG) x : (LONG) y : (LONG) w : (LONG) h;

The instance method `setSize::::` is used for resizing a PM window by the application program. The parameters `x` and `y` represent the lower left corner of the window relative to its parent, `w` and `h` the width and the height of the window.

- setRect: (LONG) w : (LONG) h;

`setRect:` is used to set the size of the window without changing the relative position in its parent window.

The new size of the window is specified by `(w/h)`.

- size: (PSWP) aSize;

`size:` fills the `SWP-structure` `aSize` with the appropriate values by querying this window's instance variables.

- (LONG) width;

`width` returns the width of the window in pixels.

- (LONG) height;

`height` returns the height of the window in pixels.

- (LONG) xoffset;

`xoffset` returns the horizontal offset of the lower left corner of the window from the lower left corner of the desktop in pixels.

- (LONG) yoffset;

`yoffset` returns the vertical offset of the lower left corner of the window from the lower left corner of the desktop in pixels.

- (HWND) window;

This method returns the handle of the Presentation Manager window associated with this window object. If no PM window is associated with this object, `NULLHANDLE` is returned.

- (ULONG) pmId;

`pmId` returns the OS/2 PM identification key of the window.

- enable;

`enable` (re-) enables this window. Message processing for this window continues after receiving this message, if the window was previously in *disabled* state.

```
{
}

@end
```

4.3 DBDateField

```
@interface DBDateField : DBField
{
}

@end
```

4.4 DBField

```
@interface DBField : Object
{
   DBField      *next;
   char         *data,
                 length,
                 decimals,
                *name,
                *string;
}

- initWithName: (char *) aName
     andLength: (char) aLength
   andDecimals: (char) someDecimals;
- free;

- setData: (void *) aPointer;
- (char *) data;

- add: (DBField *) newField;
- next;

- setString: (char *) aString;
- (char *) string;

@end
```

4.5 DBFile

```
@interface DBFile : Object
{
   DBHEADER *dbHeader;
   DBField  *fieldList;

   FILE     *fileHandle;
   void     *buffer;
```

```
- setDatabase: (DBFile *) aDatabase;
- (DBFile *) database;
- (int) count;

@end
```

4.7 DBMemoField

```
@interface DBMemoField : DBField
{
}

@end
```

4.8 DBNumField

```
@interface DBNumField : DBField
{
}

@end
```

4.9 DBRecord

```
@interface DBRecord : Object
{
  DBRecord *nextRecord;
  DBFile   *database;
  long      recNo;
  void     *buffer;
  BOOL      changed;
}

- initForDatabase: (DBFile *) aDatabase;
- free;

- insert: (DBRecord *) aRecord at: (int) index;
- deleteAt: (int) index;
- findAt: (int) index;

- copyToDB;
- copyFromDB;
- replace;

- setChanged: (BOOL) value;
- (BOOL) changed;

- (long) recNo;
```

5.3 DBDateField

Inherits from: DBField : Object

Class description:

`DBDateField` is a a special class for handling of fields storing dates.

At the moment, no additional functionality to its superclass `DBField` is provided.

5.4 DBField

Inherits from: Object

Class description:

`DBField` provides an interface to any database field stored in a DBase III compatible database. Providing methods for simple access, the program is enabled to query the information stored in a record and modify it.

Access to all fields of a database is provided by a linked list of `DBField` objects.

Instance Variables:

DBField * next;

This variable is used to hold a pointer to the `DBField` object representing the next database field. If the current field is the last one, `next` is initialized to `nil`.

char * data;

Information for the fields are stored in a global record buffer. `data` is a pointer to a location in the buffer, where the data for this field stands.

The data for a field is always stored in an array of characters of length `length`. To provide simpler access the data can be copied into a `NULL`-terminated string and written back into the record buffer after modifying it.

char length;

`length` is used to store the complete length of the field data in bytes.

char decimals;

If the field is used to store numeric values, `decimals` can be used to specify the number of decimals stored.

char * name;

The instance variable `name` holds a pointer to a `NULL`-terminated string containing the name of the field.

char * string;

As mentioned above in the description of `data`, this variable points to a `NULL`-terminated string holding the data as needed by the library functions to modify strings (`strlen ()`, `strcat ()`,...)

Reading and writing this variable should only be done using the `string` and `setString:` methods. This guarantees that the internal record buffer is up to date.

The class `DBFile` is designed to provide access to *DBase III* databases. It provides methods to read, modify and write single records in such database files.

At the moment, records are not really deleted from the database if you call the appropriate `delete` method. They are only marked as deleted. Future versions of this library will add a `pack` method, where the space allocated for those records is reused again.

So at this time, deletion of a record can easily be redone by using `undelete`.

No synchronization or locking is done by this class. So you have to take care not to open a single database file by two different `DBFile` objects, neither in the same process, nor in another one.

Instance Variables:

DBHEADER * dbHeader;

Every *DBase III* database file consists of a header and a body part.

The header stores information as the last date of update to this file, a record count and the length of a single record.

The body of the file is used to store the records themselves.

The instance variable `dbHeader` is used to store the header information for the database file. This information is modified whenever records are appended or modified.

You should never modify the header information by yourself.

DBField * fieldList;

`fieldList` is a pointer to a linked list of fields. This variable points to the first `DBField` object for this database.

FILE * fileHandle;

The variable `fileHandle` is used internally to read data from and write data to the database file. There is also no need to use it directly.

void * buffer;

When retrieving a record or storing it back into the database, an internal record buffer is used which is big enough to hold exactly one database record. `buffer` points to this area in memory.

long currentRecord;

As the internal record buffer can hold exactly one record at a given time, the `DBFile` object mus know, which record was read (to write it back into the database again). `currentRecord` stores the number of the last record which was retrieved into the record buffer.

Methods:

- init: (char *) fileName;

This initializer method `init:` is used to set up all necessary data for the database.

First, the file referenced by `fileName` is opened and the database header is read. Then the instance variables are initialized to the appropriate values. The field list is created and set up correctly.

After calling this method, you can be sure that the database you want to access is ready-to-use.

```
fieldinfo[1].length = 5 + 1 + 2;
fieldinfo[1].dec_point = 2;

[newDatabase create: "newdb.dbf"
         withFields: 2
               list: fieldinfo];
[newDatabase free];
.
.
```

Note the calculation of the field length for the second database field. The total length is calculated by adding the number of digits before the comma with 1 for the comma itself to the number of digits after the comma.

- free;

Calling `free` causes all memory allocated previously by this object to be freed again and closes the file.

An eventually modified record in the record buffer is *not* saved automatically. By default, the information is discarded.

- field: (int) fieldNumber;

This method returns a pointer to the `DBField` object for field number `fieldNumber`.

Enumeration starts at `0`.

If `fieldNumber` is out of range, `nil` is returned.

- (int) fieldCount;

`fieldCount` returns the total number of fields for the current database file.

The field numbers are in a range of `0` to `[database fieldCount] - 1`.

- readRecord: (long) offset;

Retrieve the record specified by `offset` into the record buffer.

Enumeration of records starts at `0` and ends at `[database recordCount]`.

Normally, this method should not be used by the application programmer.

- writeRecord: (long) offset;

Write the information in the internal record buffer to the database file.

`offset` must be in a range of `0` to `[database recordCount]`.

If `offset` is equal to `[database recordCount]`, a new record is appended to the database file.

Normally, this method should not be used by the application programmer.

- (long) currentRecord;

This method returns the number of the record in the internal record buffer.

- (BOOL) deleted;

if the current record is marked as deleted, `YES` is returned. Otherwise `deleted` returns `NO`.

- append;

- setBuffer: (void *) aBuffer;

`setBuffer:` copies the data in the memory ared pointed to by `aBuffer` to the internal record buffer.

- (long) recordCount;

`recordCount` returns the number of records currently stored in the database file.

5.6 DBList

Inherits from: OBJECT

Class description:

To provide access not only to single records in a database file and to avoid time-consuming fetching and storing before and after modifying a record, this class was created.

`DBList` administers a list of records, which can be retrieved in the beginning, and then modification is only done in memory, till at the end of the program, all records are stored in the database again.

At the moment, no methods for saving all records are implemented. This will change in the next release of the library.

Instance Variables:

DBRecord * firstRecord;

`firstRecord` stores a pointer to the linked list of records.

DBFile * database;

`database` stores a pointer to the associated instance of a `DBFile` object. Before any operations to records, this association must be set up.

int count;

The instance variable `count` holds the total number of records currently stored in the list.

Methods:

- init;

This method initializes an instance of `DBList`. No association with a database object is made. Before inserting or modifiying any records you *must* set up an association with `setDatabase:`.

- initForDatabase: (DBFile *) aDatabase;

This method initializes an instance of `DBList` to an existing and initialized instance of `DBFile`.

This sets up an association of this object with the database object `aDatabase`.

- free;

`free` frees the record list and all other memory allocated by this object.

- insertRecord: (DBRecord *) aRecord;

`DBNumField` is a a special class for handling of fields storing numeric values.

At the moment, no additional functionality to its superclass `DBField` is provided.

5.9 DBRecord

Inherits from: OBJECT

Class description:

The previously class `DBList` is used to store many records in memory at once. `DBRecord` is used to to the internal storage of the records and to modify the data stored here.

Instance Variables:

DBRecord * nextRecord;

The instance variable `nextRecord` points to the next record in the linked list of records.

DBFile * database;

This is a pointer to the associated instance of `DBFile`.

long recNo;

`recNo` stores the number of the record this object was created to store.

void * buffer;

`buffer` is a pointer to a memory area used as the data buffer for the record. This buffer is initialized by copying the data from the internal record buffer of the database file and all modifications to the database are accomplished by simply copying this buffer to the internal record buffer.

BOOL changed;

This variable stores the change-state of the record. After retrieving, this variable holds the boolean value `NO`. Whenever the record is changed, this variable is set to `YES`.

After saving the changes, `changed` is reset to `NO`.

Methods:

- initForDatabase: (DBFile *) aDatabase;

This method is used to initialize a newly created record object for the database file `aDatabase`.

- free;

`free` frees the complete record list and all memory previously allocated by this object.

- insert: (DBRecord *) aRecord at: (int) index;

Using `insert: at:` you can insert a new record object at index `index` into the record list.

- deleteAt: (int) index;

Delete the record at index `index`. This does not modify the deleted flag of the record associated with this object in the database object.