OS/2 Objective C Libraries

Version 0.8

Reference Manual

February 1996

Thomas Baier

baier@ci.tuwien.ac.at

Abstract

This manual is a reference manual for the *Objective C class library* for OS/2 PM and database programming.

Here all necessary information concerning the classes provided by the library can be found.

If you are searching for specific information concerning

- *Installation* ··· Read the Installation Manual.
- *Basics of Application development* ··· Read the appropriate sections in the Tutorial. There you can find a gentle introduction into using this library package for developing OS/2 PM applications.
- *Classes and Methods provided by the library* ··· You can find special information about the provided classes and methods in this manual, the Reference Manual.
- *References on the graphical development tools (database editor, interface editor, project editor)* ··· Read the appropriate sections in the Application Programming Tools Manual.
- *Literature* ··· Look in the Literature section of this Manual.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document is the *Reference Manual* for the OS/2 Objective C libraries. It provides a description of all classes and protocols defined by the class libraries “`objcutil.h`”, “`objcpm.a`” and “`objcdb.a`” along with the instance variables and methods.

The manual is organized in three parts; the first part will introduce the classes forming the utility library. The second part starting at Page 23 is merely a description of the classes and protocols of the library designed for creating Presentation Manager programs. The last part on Page 127 will focus on the classes and protocols being part of the database library.

Every part is structured alike. The first chapter of the part will be an overview of the classes and protocols organized just alike Objective C interface declarations of classes. Readers already familiar with the library package and the Objective C language will find most information which is needed there. In most cases, the name of a single method and a specification of the parameters will be enough for using it.

1.1 Extending Library Classes

One of the greatest benefits of the Objective C language is its totally dynamic run-time library. It is even possible to extend already existing classes without the need of subclassing.

To familiarize the reader with the concept of *categories* a simple example will be presented here.

Having a look at the `Printer` class provided by the Presentation Manager library you will notice that not many functions needed for drawing purposes have been implemented.

Some of the methods for printing text and drawing lines will refer to the *current cursor position* in the OS/2 presentation space used for the printer page, but there is not even a single method available to set the current position. Assume, you would like to extend the `Printer` class by adding a `-moveTo::` method taking two parameters, the x and the y coordinate of the point which is to be the current position. The implementation would be really simple, and because using Objective C extending the class itself is also possible.

For this purpose, you must create a so-called *category* for the `Printer` class. Categories are used to *add new methods* to an already existing class without having access to the source code. The category then is linked to the executable file and from then on the new methods are known by the class.

The implementation for the category `Move` for the `Printer` class is implemented just alike any other class. The only difference is that you are not allowed to define new instance variables in the interface files and overriding methods is not possible. The interface declaration stored in the file “`Move.h`” will be

```
@interface Printer (Move)

-moveTo: (LONG) x : (LONG) y;

@end
```

while the simple implementation will be stored in “`Move.m`”

```
#include "Move.h"

@implementation Printer (Move)

-moveTo: (LONG) x : (LONG) y
{
  POINTL point;

  point.x = x;
  point.y = y;

  GpiMove (hps,&point);

  return self;
}

@end
```

If you want to use this extended `Printer` class, simply compile “`Move.m`” and link the object file to your application. The Objective C run-time library will take care of adding the new method to the `Printer` class. From then on, every instance of `Printer` can respond to `-moveTo::`.

This simple example was just made to demonstrate the power of the Objective C run-time library. For more information see the *Literature* section in the *Tutorial* document.

1.2 Some Notes on the Class Reference

If you have a look at the header files installed together with the libraries you will notice that there are some classes defined by the libraries not listed in this manual. Additionally, not all methods and instance variables are documented.

The classes not documented in this manual are used *internally* by the library! You should never use instances of these classes in your applications. Some other classes like `BTree` for example are not tested and will not work as expected. Do not use them!

The instance methods which are not described here are internal methods only called by the classes' public methods. The same applies to undocumented instance variables. The behaviour of your programs is not predictable if you decide to use some of these methods or variables incorrectly. Simply don't even think of using them, even if it seems clear what function they will perform. The undocumented methods and instance variables are subject to change in future versions. Most of the other methods and instance variables will keep the same in the future.

If a class is said to *adopt* some protocols this means that the methods declared by the protocol are also available as instance methods for this class. See the protocol description for more information on these additional methods.

Part I

General Utility Classes

Chapter 2

General Utility Class Library—Overview

In this part of the manual the General Utility class library will be described. Starting with an overview of the classes and protocols in this chapter the classes themselves will be described in the next chapter.

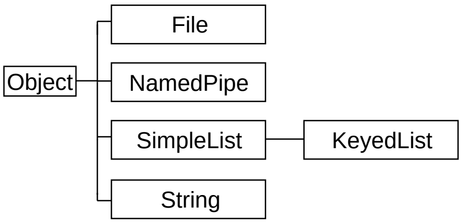

Figure 2.1: Inheritance hierarchy in General Utility Class library

Figure 2.1 shows all classes implemented in the General Utility Class library and their inheritence hierarchy.

In the beginning you will be shown an alphabetically listed overview of all classes and protocols with their instance variables and all supported methods. This was written in the style of an Objective C Interface declaration.

2.1 Class "File"

```
@interface File : Object
{
  FILE *handle;
  char *fileName;
  long  recordSize;
  long  recordCount;
  long  currentRecord;
}

-init;
-initForReading: (char *) string recordSize: (long) aRecSize;
-initForWriting: (char *) string recordSize: (long) aRecSize;
```

```
-createForWriting: (char *) string recordSize: (long) aRecSize;

-free;

-(char *) fileName;
-(long) recordSize;
-(long) recordCount;
-(long) currentRecord;
-setFileName: (char *) string;
-setRecordSize: (long) size;

-(long) append: (void *) record;
-(void *) fetch: (void *) record;
-(void *) fetch: (void *) record index: (long) index;
-(void *) first: (void *) record;
-(void *) next: (void *) record;
-(long) replace: (void *) record;
-(long) replace: (void *) record index: (long) index;

@end
```

2.2 Class “KeyedList”

```
@interface KeyedList : SimpleList
{
  id *key;
}

-initCount: (unsigned long) aCount;

-free;
-freeObjects;

-addObject: anObject;
-addObject: anObject withKey: aKey;
-insertObject: anObject at: (unsigned long) position;
-insertObject: anObject withKey: aKey at: (unsigned long) position;
-lastKey;
-keyAt: (unsigned long) position;
-removeObjectAt: (unsigned long) position;
-replaceKeyAt: (unsigned long) position with: aKey;
-objectForKey: aKey;
-(unsigned long) indexForKey: aKey;

@end
```

2.3 Class “NamedPipe”

```
@interface NamedPipe : Object
{
  HPIPE          handle;
  char          *pipeName;
  unsigned long  openMode;
  unsigned long  pipeMode;
  unsigned long  bufferSize;
```

```
}

-init;
-create;
-open;

-free;

-setPipeName: (char *) aName;
-(char *) pipeName;
-setOpenMode: (unsigned long) aMode;
-setPipeMode: (unsigned long) aMode;
-setBufferSize: (unsigned long) aSize;

-writeBlock: (void *) block size: (unsigned long) aSize;
-writeString: (char *) aString;
-readBlock: (void *) block size: (unsigned long) aSize;

@end
```

2.4 Class "SimpleList"

```
@interface SimpleList : Object
{
  id *data;

  unsigned long count;
  unsigned long maxCount;

  unsigned long slots;
}

-init;
-initCount: (unsigned long) aCount;

-free;
-freeObjects;

-(unsigned long) count;

-addObject: anObject;
-appendList: (SimpleList *) otherList;
-empty;
-insertObject: anObject at: (unsigned long) position;
-lastObject;
-objectAt: (unsigned long) position;
-(unsigned long) indexOf: anObject;
-removeLastObject;
-removeObjectAt: (unsigned long) position;
-replaceObjectAt: (unsigned long) position with: anObject;

@end
```

2.5 Class "String"

```
@interface String : Object
{
  char *stringValue;
}

-init;

-free;

-(char *) stringValue;
-setStringValue: (char *) aValue;

-(BOOL) isEqual: anObject;
-(int) compare: anotherObject;

@end
```

Chapter 3

General Utility Class Library—Classes

This chapter describes all variables and methods of the classes implemented in the General Utility Class library.

The description consists of three to six parts:

1. The name of the class and the precessing inheritance hierarchy
2. A list of protocols the class adopts. The methods defined in the *formal* protocols are not described here. See the protocols' descriptions for more information.
3. A short description of the class and its proposed usage
4. A list of all instance variables and their use
5. All *newly* implemented class and instance methods and their description
6. Methods of a *delegate object*—if such methods are supported—which get called at certain times

The list of instance variables is omitted if there are none of them defined but those inherited from the superclass.

If no return type of some method is specified, the return type defaults to `id`, a generic pointer to an *Objective C object*.

Methods returning an `id` value normally return `self`, which is a pointer to the object itself on successful completion, `nil` otherwise.

All methods having just one parameter called `sender` can be used as targets for command/action bindings. If not stated otherwise, `sender` is ignored. Normally, this parameter should be a pointer to the sending object which can be obtained by `self`.

3.1 File

Inherits from: OBJECT

Class description:

This class is a first attempt to encapsulate some of the C library's input/output functions. `File` can be used whenever record oriented access to a file located somewhere on a diskette or hard drive is needed.

Record oriented in this case implies that you can use instances of this class to store (binary) data of a fixed length. Access methods are provided.

At the moment the file format is just a *flat file* without any sorting or searching capabilibies.

Instance Variables:

FILE * handle;

`handle` is used to store the file handle of the file as opened using the C library's `fopen()` function. If you create subclasses of `File`, your methods can use this variable to access the file structure e.g. using `fseek()`, `fread()` or `fwrite()`.

char * fileName;

In this instance variable a pointer to a `NULL`-terminated C string representing the name of the *physical* file you want to access. The memory block occupied by the file name is part of the object's private data section and gets freed automatically when the object is disposed.

Use `-setFileName:` and `-fileName` for access to this data.

long recordSize;

As instances of `File` should only be used for storing various records of a fixed length, this instance variable is used to store the size of the records.

Access to this variable is provided via `setRecordSize:` and `-recordSize`.

This variable must reflect the record size of the records to be stored in the physical file on disk.

long recordCount;

This variable holds the total number of records already stored in the file. Read-only access is provided by the instance method `-recordCount`.

long currentRecord;

Internally `File` objects will always read, write or modify a single record. The number of the current record is stored in `currenRecord` and can be accessed using `-currentRecord`.

Methods:

- init;

This initializer method initializes the instance variables with default values. It should only be called by other initializer methods, as `-initForReading: recordSize:`, `-initForWriting: recordSize:` and `-createForWriting: recordSize:`.

- initForReading: (char *) string recordSize: (long) aRecSize;

`-initForReading: recordSize:` is one of the three "end-user" initializer methods provided by `File`. Using this one will try to open the file `string` for read-only access. The records are assumed to have a size of `aRecSize` bytes each.

The file must already exist, otherwise no instance of `File` will be created and `nil` is returned.

- initForWriting: (char *) string recordSize: (long) aRecSize;

To open an already existing file `string` for read and write access, use this method. As before, a pointer to the file object is returned on success, otherwise `nil` is returned and the object is freed.

- createForWriting: (char *) string recordSize: (long) aRecSize;

If you want to create a new file for reading and writing records from/to it, you must use `-createForWriting: recordSize:`. The file `string` will be created if possible and the record size is set to `aRecSize`.

On error the object is freed and `nil` is returned.

- free;

`-free` will dispose all memory blocks allocted by the object and close the file. Immediately afterwards the object itself is freed.

- (char *) fileName;

This method returns a pointer to the internal storage area holding the name of the physical disk file.

Never modify the data this method returns! To change the file's name use `-setFileName:`.

- (long) recordSize;

`-recordSize` returns the size of the records the file stores in bytes.

- (long) recordCount;

This method returns the total number of records stored in the file. The result will only be valid if the same record size was specified when opening the file as ever before for the same physical file.

- (long) currentRecord;

As was mentioned previously, instances of `File` always will handle only a single record at once. This record is called the current record. The number of the current record—its index ranges from 0 to $recordCount - 1$—can be queried using `-currentRecord`.

- setFileName: (char *) string;

This method is used to change the object's instance variable storing the physical file name. Only using this method to do this guarantees that the object keeps its internal state consistent.

Do not expect the name of the physical file on disk to be changed when using this method. Setting a new file name does not have any effect on the physical file itself.

- setRecordSize: (long) size;

To modify the instance variable `recordSize` use `-setRecordSize:`. The (constant) size of the records stored in the file is set to `size`. Modifying this value will have effect on any of the methods providing access to the single records themselves, e.g. `fetch:` or `append:`.

- (long) append: (void *) record;

`-append:` is used to add a new record to the file. The record is appended to the file. The data to be stored is taken from the memory block pointed to by `record`. Exactly `recordSize` bytes are stored in the physical file.

This method returns the record index of the newly appended record.

- (void *) fetch: (void *) record;

To fetch the current record into the buffer area pointed to by `record`, use `-fetch:`. If record is `NULL`, a memory block of size `recordSize` will be allocated automatically using `malloc()`.

This method returns a pointer to the memory block the record was stored in. Your application will be responsible for freeing the memory eventually allocated by this method after it is not needed any more.

- (void *) fetch: (void *) record index: (long) index;

To fetch a record specified by its record index `index`, use this method. As with `-fetch:` a new memory block is allocated if `record` is `NULL`.

This method returns a pointer to the memory block the record is stored in or `NULL` if an invalid record index was specified.

- (void *) first: (void *) record;

To retrieve the first record—the record having index 0—from the file object use this method.

If record is `NULL`, a new buffer area is allocated. The record buffer is returned.

Returns a pointer to the memory area used to store the record in on success, `NULL`, if no records available.

- (void *) next: (void *) record;

This method will read the next record into `record`.

On success this method returns a pointer to the record data, of no more records do exist in the file, `NULL` is returned.

- (long) replace: (void *) record;

`-replace:` replaces the data of the current record with the one in the memory area pointed to by `record`.

This method returns the index of the record that was replaced. If `currentRecord` is the same as the number of records currently stored in the file, a new record is appended.

On error—if `currentRecord` is invalid—-1 is returned.

- (long) replace: (void *) record index: (long) index;

To replace the record specified by its `index` with the data pointed to by `record` use `-replace: index:`.

This method returns the index of the record that was replaced. If `index` equals `currentRecord`, a new record is appended to the file.

If `index` is out of range, i.e. `index` is less than 0 or greater than `recordCount`, -1 is returned.

3.2 KeyedList

Inherits from: SimpleList : Object

Class description:

As a subclass of `SimpleList` instances of this class are used to store a various number of Objective C objects. In addition to instances of its superclass `KeyedList` objects will also store a *key* property together with the objects. Using some of the instance methods you can retrieve pointer to the stored objects by querying their key value.

The list is not sorted at the moment. Access by searching for the key object is not very efficient by now, so take care not to use instances of this class for large lists.

As the keys are Objective C objects as well as the "real data" to be stored in the list, you can define any object as a key object. Comparison and search is performed using the key object's `-isEqual:` and `-compareWith:` methods. To store objects having a string key, use instances of `String` for the key objects.

Instance Variables:

id * key;

`key` represents an array of pointers to objects representing the key values of the single objects stored in the list.

Methods:

- initCount: (unsigned long) aCount;

`-initCount:` is the default initializer method—also called automatically by the classes' `-init` method—to initialize a `KeyedList` object and allocate data for the key list.

For efficiency reasons, storage for the objects and the keys is allocated for many objects at once; the number of objects the storage is automatically increased if needed is specified by the parameter `aCount`.

- free;

This method frees all memory allocated by this object. The objects and the key objects stored in the list are not freed. This is done using `-freeObjects`

- freeObjects;

This method frees all objects stored in the list. The list itself isn't freed. The keys are freed, too.

- addObject: anObject;

This method appends `anObject` to the list. The key value for this object is `nil`. The object pointed to by `anObject` must not be freed while it is stored in the list.

- addObject: anObject withKey: aKey;

This method appends `anObject` to the list. The key for `anObject` is `aKey`. You must not free either the key object `aKey` or the object itself while it is stored in the list.

- insertObject: anObject at: (unsigned long) position;

This method inserts `anObject` into the list at index `position`. The index of all objects starting at `position` is increased by one. The key for this object is `nil`.

This method returns `self` on success or `nil` if the specified index is out of range.

- insertObject: anObject withKey: aKey at: (unsigned long) position;

This method inserts `anObject` into the list at index `position`. The index of all objects starting at `position` is increased by one. The key for this object is `aKey`.

Returns `self` on success, or `nil` if `position` is invalid.

- lastKey;

This method returns the key object associated with the last object in the list.

- keyAt: (unsigned long) position;

The key associated with the object stored at index `position` is returned.

- removeObjectAt: (unsigned long) position;

`-removeObjectAt:` removes the object stored at index `position` from the list. The indices of all objects stored in the list after `position` are decreased by one. On success, the removed object is returned, `nil` otherwise.

Your application program is responsible for freeing the removed object and its associated key object. Use `-keyAt:` to retrieve a pointer to the object's key object before removing it from the list.

- replaceKeyAt: (unsigned long) position;

Using this method, the key stored at index `position` is replaced with `aKey`. The key stored previously is returned. If `position` is invalid, `nil` is returned.

- objectForKey: aKey;

This method searches for an object associated with `aKey`. If found, the object is returned, `nil` otherwise.

All comparisons are performed using the key objects' `-isEqual:` methods. Depending on the class-specific implementation this will only work if the physically same key object was specified that was on insertion of the object or just an object similar to the one (e.g. `String`'s `-isEqual:` method only checks for the string value of the objects to be equivalent.).

- (unsigned long) indexForKey: aKey;

This method searches for the first key equal to `aKey` in the list. The index is returned. If no match could be found, `INVALID_INDEX` is returned.

3.3 NamedPipe

Inherits from: Object

Class description:

`NamedPipe` is used to provide a—very simple—interface to one of OS/2's inter-process communication facilities called *named pipes*.

A *name pipe* is some kind of *virtual file*, always located in the directory `\PIPE\`, providing two different processes or threads with a way of transfering data between them.

At the moment only writing and reading blocks of binary data and writing single strings is supported.

Instance Variables:

HPIPE handle;

`handle` is used to store the named pipe's file handle. It can be used in methods you add by writing a category or a subclass to `NamedPipe`.

char * pipeName;

This variable stores the name of the named pipe. It must always start with `\PIPE\`.

Read and write access to this variable is provided by the instance methods `-pipeName` and `-setPipeName:`.

unsigned long openMode;

A named pipe will be opened just like any other OS/2 file. The variable `openMode` stores the modes used for opening or creating a named pipe.

The flag which can be used are the same as for the third parameter called `openmode` of the API function `DosCreateNPipe()`.

unsigned long pipeMode;

In addition to the mode used for opening the named pipe, an additional `pipeMode` is specified.

The flag which can be used are the same as for the fourth parameter called `pipemode` of the API function `DosCreateNPipe()`.

unsigned long bufferSize;

For performance reasons a named pipe will use a buffer space when transferring data. The size of this buffer space is stored in the instance variable `bufferSize`.

Methods:

- init;

`-init` will initialize the object and its instance variables to default values.

In `openMode` the flags `NP_NOWRITEBEHIND`, `NP_NOINHERIT` and `NP_ACCESS_DUPLEX` are set, by default specifying the pipe to be created and opened for both reading and writing data.

`pipeMode` is specified as a binary or of the flags `NP_TYPE_BYTE`, `NP_READMODE_BYTE` and `NP_UNLIMITED_INSTANCES`.

The size of the transfer buffer used by the named pipe is set to `4096`.

You must set the name of the pipe using `-setPipeName:` *before* calling `-create` to create and open the named pipe or `-open` to open an already existing pipe.

- create;

This method is used to create the named pipe having the name as set to the instance variable `pipeName`. The pipe must not exist when doing so, otherwise `nil` will be returned.

After creating the named pipe the pipe is opened.

- open;

In contrast to `-create` the instance method `-open` is used to connect to (and open) an already existing named pipe. On error, `nil` is returned, `self` otherwise.

- free;

Free will close the named pipe and free the memory allocated by the object.

- setPipeName: (const char *) aName;

After calling the initializer method `-init` but before using either `-create` to create and open a new named pipe or `-open` to open an already existing pipe, you must set the pipes name using this method. `aName` has to start with `\PIPE\` if the pipe is to be opened as a pipe on the local machine or with `\Server\PIPE\` if the pipe is to be opened on the host `Server` in a local network.

The object will not copy the string to a private memory area but only store a pointer to the name as specified in the parameter `aName`.

- (const char *) pipeName;

`pipeName` will return the name of the pipe as currently stored in the object.

- setOpenMode: (unsigned long) aMode;

To set the variable `openMode` to `aMode` use this method.

- setPipeMode: (unsigned long) aMode;

`-setPipeMode:` is used to set the instance variable `pipeMode` to `aMode`.

- setBufferSize: (unsigned long) aSize;

Before opening or creating a named pipe you can specify a buffer size different from the default value of `4096` bytes to be `aSize` bytes using this method.

- writeBlock: (void *) block size: (unsigned long) aSize;

`-writeBlock: size:` will write `aSize` bytes from the memory area pointed to by `block` to the named pipe.

- writeString: (char *) aString;

This method is used to write a `NULL`-terminated string `aString` to the named pipe.

- readBlock: (void *) block size: (unsigned long) aSize;

To read in a block of `aSize` bytes into the buffer area pointed to by `block` use this method.

3.4 SimpleList

Inherits from: Object

Class description:

This class was created to provide means of storing whole lists of Objective C objects. Its interface concerning the instance methods is mostly compatible with *NEXTSTEP*'s or the library `libobjects`' `List` class.

Instances of SimpleList can store objects of any type. To optimize access to the objects stored, arrays are used. When a new memory area has to be allocated for adding new objects the object will always allocated enough storage to store exactly `slots` new objects in the list.

Indices of the list items start at 0.

Instance Variables:

id * data;

This pointer variable represents an array the objects in the list are stored in.

unsigned long count;

The current number of objects stored in the list is stored in this instance variable.

unsigned long maxCount;

Because `SimpleList` objects will always allocate a bulk of memory large enough for storing `slots` new records the number of objects a list can store at most will almost always be larger than the number of objects currently stored. Only after `maxCounts` objects are stored in the list and a new object is to be appended or inserted a new memory area will be allocated suitable to store `maxCount + slots` objects, which is `slots` new objects than there are already stored in the list.

unsigned long slots;

The number of objects the size of the list will be increase on demand is stored in `slots`.

Methods:

- init;

This method initializes the object. `slots` is set to `5`.

- initCount: (unsigned long) aCount;

This method really initializes an instance of `SimpleList`. `data` is set to `NULL`, `count` and `maxCount` are set to `0`, `slots` is initialized with the method's only parameter `aCount`.

- free;

This method frees all memory allocated by this object. The objects stored in the list are not freed.

- freeObjects;

This method frees all objects stored in the list. The list itself isn't freed.

- (unsigned long) count;

`-count` returns the number of objects currently stored in the list.

- addObject: anObject;

This method appends `anObject` to the list.

- appendList: (SimpleList *) otherList;

Append the objects in `otherList` to this list.

- empty;

This method empties the list without freeing the objects.

- insertObject: anObject at: (unsigned long) position;

This method inserts `anObject` into the list at index `position`. The index of all objects starting at `position` is increased by one. Returns `self` on success, or `nil` if `position` is invalid.

- lastObject;

This method returns the last object stored in the list.

- objectAt: (unsigned long) position;

`-objectAt:` returns the object stored at index `position`. If `position` is invalid, `nil` is returned.

- (unsigned long) indexOf: anObject;

To find the index of the object `anObject` use `-indexOf:`. Comparisons and search are performed using the stored objects' `isEqual:` methods.

- removeLastObject;

This method removes the last object in the list. The object removed is returned. If there is no object in the list when calling this method, `nil` is returned.

Your application is responsible for freeing the removed object if it is needed no more.

- removeObjectAt: (unsigned long) position;

`-removeObjectAt:` removes the object stored at index `position` from the list. The indices of all objects stored in the list after position are decreased by one. On success, the removed object is returned, `nil` otherwise.

- replaceObjectAt: (unsigned long) with: anObject position;

Using this method, the object stored at index `position` is replaced with `anObject`. The object stored previously is returned. If position is invalid, `nil` is returned and no changes are performed.

3.5 String

Inherits from: OBJECT

Class description:

To handle `NULL`-terminated strings as Objective C objects you can use instances of the class `String`. The memory management is handled automatically for you, setting and reading the data is only possible using the instance methods `-setStringValue:` and `-stringValue`.

`String` objects are quite useful as the key objects in a `KeyedList`.

Instance Variables:

char * stringValue;

`stringValue` stores a pointer to a `NULL`-terminated string representing the value of the object. The string itself is stored in a private memory area of the object; the memory management is automatically handled by the instance methods of this class.

Methods:

- init;

`-init` initializes the object and sets `stringValue` to an empty string.

- free;

This method will free the object and the memory allocated for storing the string value of the object itself.

- (char *) stringValue;

`-stringValue` returns a pointer to the private memory area of the object where the string value is stored. Do not modify this memory area yourself.

- setStringValue: (char *) aValue;

To set and/or modify the objects value use `-setStringValue:`. It is taken care that enough memory is allocated and the `NULL`-terminated string pointed to by `aValue` is copied into the private memory area of the object.

- (BOOL) isEqual: anObject;

This method is used to check the object `anObject` and this object for equivalence. The objects are said to be equivalent if they store the *same* string value as checked by character-wise comparisons.

`anObject` does not have to be an instance of `String` by itself, it does only have to respond to the selector `-stringValue`.

- (int) compare: anotherObject;

`-compare:` will return `0` if the object itself and `anotherObject` are storing the same string value.

For this purpose, `anotherObject` must respond to the selector `-stringValue`.

In case the string value of this object is less than the string value of `anotherObject` an integer number less than `0` is returned, in case the string value of this object is greater, an integer greater than `0` is returned.

If `anotherObject` does not respond to `-stringValue`, the `-compare:` method of `String`'s superclass is invoked and the result of it is returned.

Part II

PM Programming Classes

Chapter 4

Presentation Manager Library—Overview

In this part of the manual the Presentation Manager library will be described. Starting with an overview of the classes and protocols provided in this chapter the classes will be described in the next chapter (see Page 43). In Chapter 6 on page 121 the protocols defined and used by the PM library will be dealt with.

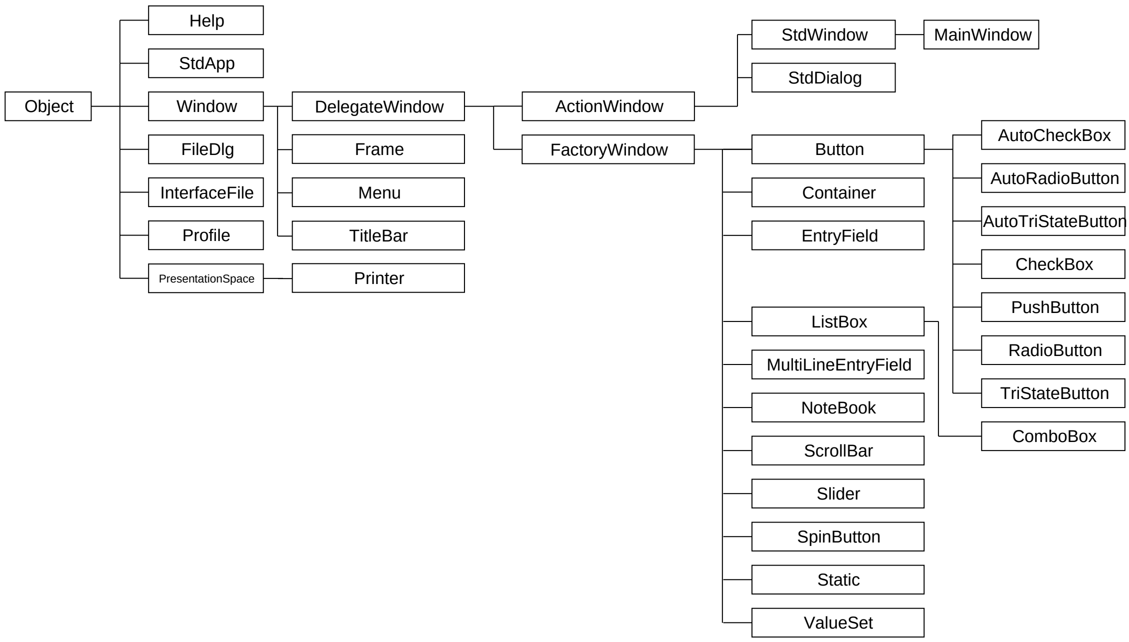

Figure 4.1: Inheritance hierarchy in Presentation Manager Class library

Figure 4.1 shows all classes implemented in the Presentation Manager library and their inheritence hierarchy.

In the beginning you will be shown an alphabetically listed overview of all classes and protocols with their instance variables and all supported methods. This was written in the style of an Objective C Interface declaration.

General information about PM programming can be found in [2] and [1]

4.1 Class “ActionWindow”

```
@interface ActionWindow : DelegateWindow <Archiving>
{
  CommandList *commandBindings;
  id           clientWindow;
  id           menu;
}

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

-menu;
-createMenu;
-destroyMenu;

-setClientWindow: aClient;
-clientWindow;
-performClose: sender;
-performQuit: sender;

@end
```

4.2 Protocol “Archiving”

```
@protocol Archiving

-awake;
-read: (TypedStream *) aStream;
-write: (TypedStream *) aStream;

@end
```

4.3 Class “AutoCheckBox”

```
@interface AutoCheckBox : Button
{
}

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

@end
```

4.4 Class “AutoRadioButton”

```
@interface AutoRadioButton : Button
{
```

```
}

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

@end
```

4.5 Class "AutoTriStateButton"

```
@interface AutoTriStateButton : Button
{
}

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

@end
```

4.6 Class "Button"

```
@interface Button : FactoryWindow <Archiving>
{
  Command command;
}

-initWithId: (ULONG) anId andFlags: (ULONG) flags
          in: (Window *) parent;
-clickdown;
-clickup;
-(USHORT) checked;
-(BOOL) highlighted;
-check;
-checkIndeterminate;
-uncheck;

-(char *) text: (char *) buffer;
-setText: (char *) buffer;

-setTarget: aTarget;
-setAction: (SEL) anAction;

-bindWith: anObject selector: (SEL) aSel;

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

@end
```

4.7 Class "CheckBox"

```
@interface CheckBox : Button
{
```

```
}

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

@end
```

4.8 Class "ComboBox"

```
@interface ComboBox : ListBox
{
}

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

-(HWND) entryField;
-(HWND) listBox;

@end
```

4.9 Class "Container"

```
@interface Container : FactoryWindow <Archiving>
{
  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;

-insertObjectList: aList;
-insertObjectList: aList withTitles: titleList;
-insertObjectList: aList withTitles: titleList
          andIcon: (ULONG) anIcon;

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

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

```
-detailView: sender;

-(ULONG) records;
-object;

-(CONTAINER_MINIREC *) findRecord: anObject;
-(CONTAINER_MINIREC *) firstRecord;
-(CONTAINER_MINIREC *) lastRecord;
-(CONTAINER_MINIREC *) nextRecord;
-(CONTAINER_MINIREC *) previousRecord;
-(CONTAINER_MINIREC *) childRecord;
-(CONTAINER_MINIREC *) parentRecord;
-(CONTAINER_MINIREC *) firstSelected;
-(CONTAINER_MINIREC *) nextSelected;
-(BOOL) recordIsSelected;

-objectWithTitle: (char *) aTitle;

-invalidateRecord;
-invalidateSelectedRecords;

-hideRecord : sender;
-hideSelectedRecords : sender;
-hideNotSelectedRecords : sender;
-showRecord : sender;
-showAllRecords : sender;
-(BOOL) recordIsHidden;

-removeAllColumns: sender;
-removeAllRecords: sender;
-removeSelectedRecords: sender;

-(ULONG) columns;

-(FIELDINFO *) firstColumn;
-(FIELDINFO *) lastColumn;
-(FIELDINFO *) nextColumn;
-(FIELDINFO *) previousColumn;

-(char *) columnTitle;
-(ULONG) columnTitleAttributes;
-(ULONG) columnDataAttributes;

-hideColumn : sender;
-showColumn : sender;
-showAllColumns : sender;
-(BOOL) columnIsHidden;

-invalidateColumns;
-setColumnTitleAttributes: (ULONG) attr;
-setColumnDataAttributes: (ULONG) attr;

-select;
-deselect;
-selectAll: sender;
-deselectAll: sender;

-sort: (ULONG) column;
```

```
-sortByTitleWithCase: sender;
-sortByTitleWithoutCase: sender;

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

@end
```

4.10 Class “DelegateWindow”

```
@interface DelegateWindow : Window
{
  id delegate;
}

-init;
-setDelegate: anObject;
-delegate;

@end
```

4.11 Class “EntryField”

```
@interface EntryField : FactoryWindow <Selection,Archiving,Value>
{
  int textLimit;
}

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

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

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

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

@end
```

4.12 Class “FactoryWindow”

```
@interface FactoryWindow : DelegateWindow
{
  HWND owner;
}

-initIn: (Window *) parent;
```

```
-associate: (HWND) hwnd;

-destroy;

-setWindow: (HWND) aWindow;

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

-specialClass;

@end
```

4.13 Class "FileDlg"

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

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

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

-runModalFor: sender;

-(char *) fileName;
-(ULONG) result;

@end
```

4.14 Class "Frame"

```
@interface Frame : Window
{
}

@end
```

4.15 Class "Help"

```
@interface Help : Object <Archiving>
{
  BOOL  helpEnabled;
```

```
  HWND  helpInstance;

  id    helpFileName;
  id    helpWindowTitle;
  ULONG helpTable;
  HAB   hab;
}

-init;
-initForApp: (StdApp *) anApp helpFile: (char *) fileName;
-initForApp: (StdApp *) anApp helpFile: (char *) fileName
  withTitle: (char *) helpTitle;
-initForApp: (StdApp *) anApp helpFile: (char *) fileName
  withTitle: (char *) helpTitle helpTable: (ULONG) helpTableID;

-free;

-associateWith: aWindow;

-createHelpInstance;

-helpForHelp: sender;
-helpExtended: sender;
-helpIndex: sender;
-helpFor: sender;
-helpForTopic: (int) topic;

-helpFileName;
-helpWindowTitle;
-(ULONG) helpTable;
-setHelpTable: (ULONG) anId;

@end
```

4.16 Class “InterfaceFile”

```
@interface InterfaceFile : Object <Archiving>
{
}

-init;
-free;
-freeObjects;

-objectWithTitle: (char *) aTitle;

@end
```

4.17 Class “ListBox”

```
@interface ListBox : FactoryWindow <Archiving>
{
}

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

```
          in: (Window *) parent;

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

-(unsigned long) count;
-(SHORT) selected;
-(SHORT) itemTextLength: (SHORT) pos;
-(char *) item: (SHORT) pos text: (char *) buffer;

-selectItem: (SHORT) pos;
-deleteItem: (SHORT) pos;
-deleteAll;

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

@end
```

4.18 Class "MainWindow"

```
@interface MainWindow : StdWindow
{
}

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

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

@end
```

4.19 Class "Menu"

```
@interface Menu : Window <Archiving>
{
  id itemList;
}

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

-free;

-insertItem: anItem at: (SHORT) pos;

-enableItem: (USHORT) identifier;
-disableItem: (USHORT) identifier;

-popup: sender;

-itemList;
```

```
@end
```

4.20 Class “MenuItem”

```
@interface MenuItem : Object
{
  ULONG  pmId;
  char  *title;
  id     subMenu;
}

-init;
-initWithId: (ULONG) anId andTitle: (char *) aTitle;
-initWithId: (ULONG) anId andTitle: (char *) aTitle subMenu: aSubMenu;

-free;

-setTitle: (char *) aTitle;
-(char *) title;
-setPmId: (ULONG) anId;
-(ULONG) pmId;
-setSubMenu: aSubMenu;
-subMenu;

@end
```

4.21 Class “MultiLineEntryField”

```
@interface MultiLineEntryField : FactoryWindow <Archiving,Selection>
{
}

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

-(long) indexForLine: (long) line;
-delete: (unsigned long) count at: (long) index;
-insertText: (char *) aText;
-insertText: (char *) aText at: (long) index;
-appendText: (char *) aText;

-clearAll: sender;

@end
```

4.22 Class “NoteBook”

```
@interface NoteBook : FactoryWindow <Archiving>
{
}
```

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

-(ULONG) createTopPageWithStatusText: (BOOL) stFlag;
-(ULONG) createMajorTopPageWithStatusText: (BOOL) stFlag;
-(ULONG) createMinorTopPageWithStatusText: (BOOL) stFlag;

-(ULONG) createLastPageWithStatusText: (BOOL) stFlag;
-(ULONG) createMajorLastPageWithStatusText: (BOOL) stFlag;
-(ULONG) createMinorLastPageWithStatusText: (BOOL) stFlag;

-(ULONG) createPageBefore: (ULONG) pageID
           withStatusText: (BOOL) stFlag;
-(ULONG) createMajorPageBefore: (ULONG) pageID
                withStatusText: (BOOL) stFlag;
-(ULONG) createMinorPageBefore: (ULONG) pageID
                withStatusText: (BOOL) stFlag;

-(ULONG) createPageAfter: (ULONG) pageID
          withStatusText: (BOOL) statusFlag;
-(ULONG) createMajorPageAfter: (ULONG) pageID
               withStatusText: (BOOL) stFlag;
-(ULONG) createMinorPageAfter: (ULONG) pageID
               withStatusText: (BOOL) stFlag;

-deletePage: (ULONG) pageID;
-deleteAllPages;

-setStatusLineTextFor: (ULONG) pageID
                   to: (char *) aString;
-setTabTextFor: (ULONG) pageID
            to: (char *) aString;

-setMajorTabDimensions: (USHORT) width : (USHORT) height;
-sizeMajorTabs;
-setMinorTabDimensions: (USHORT) width : (USHORT) height;
-setPageButtonDimensions: (USHORT) width : (USHORT) height;

-setPage: (ULONG) pageID to: (Window *) aWindow;
-selectPage: (ULONG) pageID;

@end
```

4.23 Class “PresentationSpace”

```
@interface PresentationSpace : Object <Archiving>
{
  HPS   hps;
  float spacing;
  LONG  xResolution;
  LONG  yResolution;
  LONG  textHeight;
}

-init;

-setHPS: (HPS) aHPS;
```

```
-(HPS) hps;

-(LONG) widthInPels;
-(LONG) widthInMm;
-(LONG) heightInPels;
-(LONG) heightInMm;

-(LONG) xResolution;
-(LONG) yResolution;

-(LONG) textHeight;

-setSpacing: (float) ratio;
-(float) spacing;

-setFont: (char *) fontName;
-setFont: (char *) fontName at: (LONG) pointSize;
-setFontSize: (LONG) pointSize;

-(LONG) stringWidth: (char *) aString;
-(LONG) stringBoxWidth: (char *) stringBox;
-(LONG) stringBoxHeight: (char *) stringBox;

-string: (char *) aString;
-string: (char *) aString at: (LONG) x : (LONG) y;
-stringBox: (char *) stringBox;
-stringBox: (char *) stringBox at: (LONG) x : (LONG) y;

-lineTo: (LONG) x : (LONG) y;
-lineFrom: (LONG) x0 : (LONG) y0 to: (LONG) x : (LONG) y;

@end
```

4.24 Class “Printer”

```
@interface Printer : PresentationSpace <Archiving>
{
  id           setupDialog;
}

-initForApp: (StdApp *) anApp;
-free;

-beginSpool: (char *) spoolTitle
   appTitle: (char *) appTitle;
-endSpool;
-newPage;

-printerSetup: sender;

-setSetupDialog: aSetupDialog;
-setupDialog;
-jobProperties: sender;
-setDefault;

-(LONG) widthInPels;
-(LONG) widthInMm;
```

```
-(LONG) heightInPels;
-(LONG) heightInMm;

@end
```

4.25 Class “Profile”

```
@interface Profile : Object
{
  HINI profileHandle;
}

-init;
-init: (char *) name forApp: (StdApp *) app;
-initUserProfile;
-initSystemProfile;
-free;

-deleteKey: (char *) key section: (char *) section;
-(void *) readData: (void *) data key: (char *) key
           section: (char *) section;
-(void *) readData: (void *) data size: (long) size
               key: (char *) key section: (char *) section;
-(char *) readString: (char *) data key: (char *) key
             section: (char *) section;
-(char *) readString: (char *) data size: (long) size
                 key: (char *) key section: (char *) section;
-(long) sizeForKey: (char *) key section: (char *) section;
-writeData: (void *) data size: (long) size
       key: (char *) key section: (char *) section;
-writeString: (char *) data key: (char *) key
     section: (char *) section;

@end
```

4.26 Class “PushButton”

```
@interface PushButton : Button
{
}

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

@end
```

4.27 Class “RadioButton”

```
@interface RadioButton : Button
{
}
```

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

@end
```

4.28 Class "ScrollBar"

```
@interface ScrollBar : FactoryWindow <Archiving>
{
}

-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
```

4.29 Protocol "Selection"

```
@protocol Selection

-clearSelection: sender;
-copySelection: sender;
-cutSelection: sender;
-pasteSelection: sender;

@end
```

4.30 Class "Slider"

```
@interface Slider : FactoryWindow <Archiving>
{
}

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

@end
```

4.31 Class "SpinButton"

```
@interface SpinButton : FactoryWindow <Archiving>
{
}

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

@end
```

4.32 Class “Static”

```
@interface Static : FactoryWindow <Archiving>
{
}

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

@end
```

4.33 Class “StdApp”

```
@interface StdApp : Object
{
  HAB   hab;
  HMQ   hmq;
}

-init;
-free;
-run;

-(HAB) hab;

-loadIBFile: (char *) fileName;

@end
```

4.34 Class “StdDialog”

```
@interface StdDialog : ActionWindow <Archiving>
{
  ULONG result;
  BOOL  running;

  ULONG createFlags;
}

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

```
-(ULONG) createFlags;
-setCreateFlags: (ULONG) flags;

-updateFrame;

-(ULONG) result;

-setTitle: (char *) aTitle;

-makeKeyAndOrderFront: sender;
-runModalFor: sender;
-dismiss: sender;
-dismissOK: sender;
-(BOOL) running;

-centerOnScreen: sender;

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

@end
```

4.35 Class “StdWindow”

```
@interface StdWindow : ActionWindow <Archiving>
{
  HWND  frame;

  ULONG createFlags;
}

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

-(ULONG) createFlags;
-setCreateFlags: (ULONG) flags;

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

-updateFrame;

-(LONG) framexoffset;
-(LONG) frameyoffset;
-(LONG) framewidth;
-(LONG) frameheight;
-frameSize: (PSWP) aSize;

-(HWND) frame;

-setTitle: (char *) aTitle;

-makeKeyAndOrderFront: sender;

-centerOnScreen: sender;
```

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

@end
```

4.36 Class “TitleBar”

```
@interface TitleBar : Window
{
}

@end
```

4.37 Class “TriStateButton”

```
@interface TriStateButton : Button
{
}

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

@end
```

4.38 Protocol “Value”

```
@protocol Value

-(char *) stringValue;
-(int) intValue;
-(long) longValue;
-(float) floatValue;

-setStringValue: (char *) aValue;
-setIntValue: (int) aValue;
-setLongValue: (long) aValue;
-setFloatValue: (float) aValue;

@end
```

4.39 Class “ValueSet”

```
@interface ValueSet : Window
{
}

@end
```

4.40 Class "Window"

```
@interface Window : Object <Archiving>
{
  HWND    window;
  Window *child;
  Window *sibling;

  int     tag;
}

-init;
-associate: (HWND) hwnd;
-free;

-destroy;

-(int) tag;
-setTag: (int) aTag;

-createObjects;
-insertChild: aChild;
-insertSibling: aSibling;
-deleteChild: aChild;
-deleteSibling: aSibling;

-findFromID: (ULONG) anId;
-findFromHWND: (HWND) aHwnd;

-setChild: (Window *) aChild;
-(Window *) child;
-setSibling: (Window *) aSibling;
-(Window *) sibling;

-(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;

-(ULONG) windowStyle;

-setWindowStyle: (ULONG) styleFlags;

-(HWND) window;
-setWindow: (HWND) aWindow;
-(ULONG) pmId;
-setPmId: (ULONG) anId;

-enable;
-disable;
-activate;
```

```
-deactivate;

-invalidate;
-show;
-hide;

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

@end
```

Chapter 5

Presentation Manager Library—Classes

This chapter describes all variables and methods of the classes implemented in this library.

The description consists of three to six parts:

1. The name of the class and the precessing inheritance hierarchy
2. A list of protocols the class adopts. The methods defined in the *formal* protocols are not described here. See the protocols' descriptions for more information.
3. A short description of the class and its proposed usage
4. A list of all instance variables and their use
5. All *newly* implemented class and instance methods and their description
6. Methods of a *delegate object*—if such methods are supported—which get called at certain times

The list of instance variables is omitted if there are none of them defined but those inherited from the superclass.

If a class doesn't support delegate objects the corresponding section in the class description is omitted.

If in some method description no return type is specified, the return type defaults to `id`, a generic pointer to an *Objective C object*.

Methods returning an `id` value normally return `self`, which is a pointer to the object itself on successful completion, or `nil` otherwise.

All methods having just one parameter called `sender` can be used as targets for command/action bindings. If not stated otherwise, `sender` is ignored. Normally, this parameter should be a pointer to the sending object which can be obtained by `self`.

5.1 ActionWindow

Inherits from: DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

`ActionWindow` is the common superclass for `StdWindow` and `StdDialog`. This class implements the ability to bind command messages to methods in other objects.

Everytime a command message occurs in a `StdWindow` or `StdDialog` the Event-Handler searches for a command binding and—if found—executes the corresponding *Action* in the *Target* object.

In addition to handling command messages, instances of `ActionWindow` do contain basic support for accessing a so-called *client window* and a *menu bar*.

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.

id clientWindow;

Although a pointer to a *client window* is stored in instances of this class, subclasses must care to handle these special windows for themselves. Normally, a client window should be the size of the parent, which is the instance of `ActionWindow` itself. If the `ActionWindow` is resized or moved, the `clientWindow` should behave accordingly.

The subclasses `StdDialog`, `StdWindow` and `MainWindow` implement handling the `clientWindow` so that the *client window* is always the size of the parent window. This is quite useful when only displaying a single control element—e.g. a `ListBox` or a `MultiLineEntryField`—in a window or dialog window. Access to this variable is provided via `-clientWindow` and `-setClientWindow:`.

id menu;

At the moment, `menu` is just a dummy variable. In future releases, this variable will be used to store a pointer to the main menu of the `ActionWindow`. All manipulations concerning this menu will be directed to `menu`.

For those wishing to experiment with menus and the Objective C classes `Menu` and `MenuItem`—be aware, these classes are subject to change—the methods `-menu`, `-createMenu` and `-destroyMenu` are provided.

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.

If a main menu for the window was created using `-createMenu`, it is *not freed* automatically. Use `-destroyMenu` before calling free!

- 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*.

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

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.

- menu;

This method returns the instance of `Menu` representing the main menu for the `ActionWindow`. Creating and deleting this menu can be performed with `-createMenu` and `-destroyMenu`.

- createMenu;

`-createMenu` will create an instance of `Menu` having the presentation manager identifier `FID_MENU` and the flag `MS_ACTIONBAR` set.

The menu can be accessed using `-menu`.

- destroyMenu;

If a menu was created for this `ActionWindow` using `-createMenu`, it is destroyed.

- setClientWindow: aClient;

`-setClientWindow` sets the instance variable `clientWindow` to `aClient`.

Instances of `ActionWindow`'s subclasses `StdDialog`, `StdWindow` and `MainWindow` will cause the `clientWindow` always to be the same size as the window itself.

- clientWindow;

This method returns the value stored in the instance variable `clientWindow`.

- performClose: sender;

A `WM_CLOSE` message is posted to the window. Depending on the subclass of `ActionWindow` which is used, the window either gets closed (`StdDialog` and `StdWindow`) or the application quits (`MainWindow`).

This is an *action method*, `sender` is ignored.

- performQuit: sender;

`WM_QUIT` is posted. This normally causes the application to terminate.

This method can be used as an action method. `sender` is ignored.

5.2 AutoCheckBox

Inherits from: BUTTON : FACTORYWINDOW : DELEGATEWINDOW : WINDOW : OBJECT

Class description:

The class `AutoCheckBox` is a subclass of `Button`. Its 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 5.1 on page 48. Figure 5.1 on page 48 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`.

5.3 AutoRadioButton

Inherits from: Button : FactoryWindow : DelegateWindow : Window : Object

Class description:

The class `AutoRadioButton` is a subclass of `Button`. Its 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 5.1 on page 48. Figure 5.1 on page 48 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 `AutoRadioButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_AUTORADIOBUTTON`.

5.4 AutoTriStateButton

Inherits from: Button : FactoryWindow : DelegateWindow : Window : Object

Class description:

The class `AutoTriStateButton` is a subclass of `Button`. Its 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 5.1 on page 48. Figure 5.1 on page 48 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 `AutoTriStateButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_AUTO3STATE`.

5.5 Button

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

The Objective C class `Button` represents a special type of a `Window`. Instances of this class are normally associated with PM Windows of class `WC_BUTTON`. The instance methods can be used to set the state of a Button (to simulate a User Action to the Button) or to query the Button's state if it is a *Radiobutton*, a *Checkbox* or a *Tri-State Button*.

Setting and querying the text displayed in the Button can be done using `-setText:` and `-text:`.

Support for displaying icons instead of a text on a Button is currently not implemented when creating a Button Object "from Scratch", which means by not using a definition for this object in a OS/2 Resource File.

Instance Variables:

Command command;

This variable is used to store a command binding. The binding consists of a *target object* set via `-setTarget:` and an *action selector* (use `-setAction:` to set this part).

If the button is clicked and a target object and a target action are set, the appropriate method of the target object is called. `self` is passed as the only parameter to the action.

For subclasses of `Button` this means, that a command is issued, when

- a `PushButton` is pressed,
- a `RadioButton` or `AutoRadioButton` is put into *checked* state, a
- a `CheckBox` or `AutoCheckBox` changes its state either to *unchecked* or *checked* or
- a `TriStateButton` or an `AutoTriStateButton` changes its state to *unchecked*, *checked*, or *indeterminate*.

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 its parent. Use `[parent insertChild: button]` where `parent` is the parent window and `button` is the newly created Button Object.

The following table list all possible `BS_xxxx` styles and a short description of these.

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 its 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_PAINT` message is received by the parent window.

Table 5.1: Main button styles used to define the type of button

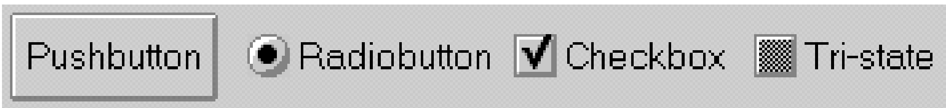

Figure 5.1: This figure shows (from left to right) the following button types:*push button*, *radio button*, *check box* and *tri-state button*.

First the primary Button styles, which define the type of the Button. One of these must be given. All other style options in the following tables can be combined with one of the primary style via logical OR. Tables 5.1 (page 48), 5.2 (page 48), 5.3 (page 49) and 5.4 (page 49).

Figure 5.1 on page 48 shows the look of the main Button styles.

- clickdown;

By calling this method a click down with the left mouse button is simulated for this Button.

- clickup;

`-clickup` simulates—as a counterpart to `-clickdown`—a release of the left mouse button when the mouse pointer is in the Button ("Click Up").

- (ULONG) checked;

`-checked` queries the check state of the Button if it is a *Radiobutton*, a *Checkbox* or a *Tri-State Button*.

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

Table 5.2: Button styles which can be combined with an `AutoRadiobutton`

Flag	Description
BS_HELP	Instead of posting a command message (`WM_COMMAND`), a help message is posted (`WM_HELP`).
BS_SYSCOMMAND	When this style is set, a `WM_SYSCOMMAND` message is posted instead of a command message (`WM_COMMAND`).
BS_NOBORDER	The push button does not have a drawn border.

Table 5.3: Button styles which can be combined with a `PushButton`

Flag	Description
BS_DEFAULT	Only one button per window should have this style set. In dialogs this button is automatically pushed whenever the user presses the *Enter* key.

Table 5.4: Button styles which can be combined with a `PushButton` or a user button control

This method returns `0` if the Button is in unchecked state, `1` when in checked state and `2` when in indeterminate state.

- (BOOL) highlighted;

The result of `-highlighted` is `YES` if the current state of the Button is highlighted, `NO` otherwise.

- check;

`-check` sets the *checked* state of the Button.

- checkIndeterminate;

`-checkIndeterminate` sets the *indeterminate* state of the Button.

- uncheck;

`-uncheck` sets the *unchecked* state of the Button.

- (char *) text: (char *) buffer;

This method returns the button text. Defining the *short-cut* character by prefixing it with a tilde (~) is preserved.

If `buffer` is `NULL`, enough memory will be allocated to store the title string. This memory area must be freed after using it by calling the C library function `free()`.

On success, a pointer to `buffer` or a newly allocated buffer area is returned.

- setText: (char *) buffer;

Set the title text of the `Button` to `buffer`.

`self` is returned.

- setTarget: aTarget;

This method sets the target object of the button control (see the instance variable `command`) to `aTarget`.

The binding is activated as soon as the *target* object is set to a non-`nil` value. Therefore you should always set the action method (`-setAction:`) first.

To disable a command binding, simply set the target object to `nil` e.g. using

```
[button setTarget: nil];
```

- setAction: (SEL) anAction;

The *action method* stored in `command`is set to `anAction`.

- bindWith: anObject selector: (SEL) aSel;

This method is just a short-cut for using

```
[button setAction: aSel];
[button setTarget: anObject];
```

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

This message-handling function is used to directly call the specified *action method* of the *target object* whenever the button is pressed. A pointer to the button is passed as the only parameter to the action.

The *target-action* binding is assumed to exist if the target object has been set either by using `-setTarget:` or by `-bindWith: selector:`.

5.6 CheckBox

Inherits from: BUTTON : FACTORYWINDOW : DELEGATEWINDOW : WINDOW : OBJECT

Class description:

The class `CheckBox` is a subclass of `Button`. Its 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`.

5.7 ComboBox

Inherits from: LISTBOX : FACTORYWINDOW : DELEGATEWINDOW : WINDOW : OBJECT

Class description:

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

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`.

Flag	Description
CBS_SIMPLE	If this flag is specified, the entry field *and* the list box are visible at any time.
CBS_DROPDOWN	This flag causes the list box 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 list box. Here the list box is only displayed on user demand, and the entry field displays one of the list box items. It is not editable.

Table 5.5: Combo box control styles

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 5.5), which can be a combination of one of the flags special to *Combo boxes* or general window flags (e.g. `WS_VISIBLE`).

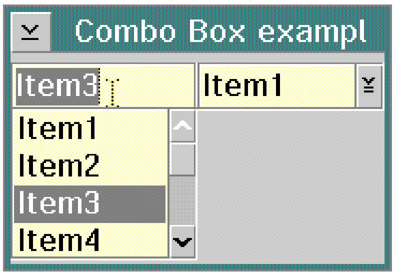

Figure 5.2: Combo box controls (simple and dropdown)

As in all other `initWithId: andFlags: in:` methods, `parent` is a pointer to the parent window, which must be an instance of `Window` or one of its subclasses.

Figure 5.2 shows two combo boxes. The left one was created using `CBS_SIMPLE`, the right one using the flag `CBS_DROPDOWN`. Using `CBS_DROPDOWNLIST` would have the same apperance as the right combo box.

- (HWND) entryField;

This method returns the window handle (type `HWND`) of the entry field control.

- (HWND) listBox;

This method returns the window handle (type `HWND`) of the list box control.

5.8 Container

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

The OS/2 container class is one of the most flexible user interface classes provided by the OS/2 API. Because of its flexibility, there's a lot of work to do for the application programmer, before he can use a container object in his programs successfully. The Objective C class `Container` was created to simplify the handling of the API functions concerning this PM class.

Information concerning this specific PM class and how to use it in standard C programs can be found in [3]. This series of articles is part of EDM/2 and can be obtained via anonymous ftp from *hobbes.nmsu.edu* and many other sites which provide OS/2 software.

Instances of this class are designed to store Objective C objects as data. So anything you can encapsulate in an Objective C class definition can be stored in container objects.

Don't forget to specify `CCS_MINIRECORDCORE` at initialization. Only this style is supported at the moment.

When using detail's view, the objects inserted must conform to the protocol `ContainerItem`.

To insert single container items, use

- `-insertObject:`,
- `-insertObject: withTitle:` and
- `-insertObject: withTitle: andIcon:`.

Because inserting many objects using these methods is quite slow, methods for inserting many objects at once are provided. These are

- `-insertObjectList:`,
- `-insertObjectList: withTitles:` and
- `-insertObjectList: withTitles: andIcon:`.

Methods to arrange the container objects in the *tree view* as known from the workplace shell's directory folders have also been implemented.

- `-insertObject: parent:`,
- `-insertObject: withTitle: parent:` and
- `-insertObject: withTitle: andIcon: parent:`

will only insert single objects at the moment.

Instance Variables:

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;

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 5.6: 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 5.7: Container creation styles concerning selection

`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 its 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 5.6 and 5.7 shortly describe the flags specific to container controls.

- addColumn: (char *) aTitle;

When in detail's view, the container displays data in multiple columns. This method can be used to add a new column at run-time. `aTitle` is the title text of the column which is shown if this feature is enabled.

- insertObject: anObject;

`-insertObject:` inserts a new object into the container. `anObject` must be an allocated and initialized Objective C object. The title text of the newly inserted object is empty, the application uses the clock pointer (`SPTR_WAIT`) as the icon to be displayed.

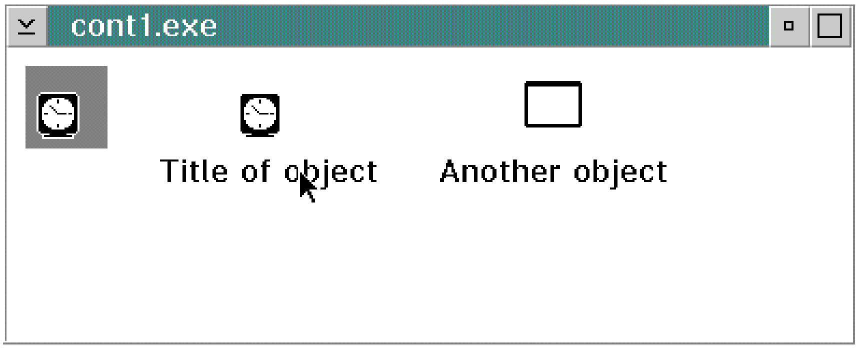

Figure 5.3: Container window in icon view showing objects created with the different `-insertObject:` methods

This method is preferred for inserting new objects into a container when in detail's view, where it doesn't matter, which icon or title is displayed.

Figure 5.3 shows a cotainer window in icon view. The leftmost object was inserted using `-insertObject:`.

When in one of the other container views, consider using `-insertObject: withTitle:` or `-insertObject: withTitle: andIcon:`.

When removed from the container object, the objects are *not* freed. You have to do this by yourself.

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

Just as with `-insertObject:` a new object is inserted into the container. `aTitle` specifies the title text to be displayed. This method should be used if the container is in text view. The clock pointer is used as icon.

Figure 5.3 shows a cotainer window in icon view. The middle object was inserted using `-insertObject: withTitle:`.

After inserting the object into the container, the memory block pointed to by `aTitle` can be reused in some other way.

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

Using `-insertObject: withTitle: andIcon:` the application programmer has full control over the data displayed in any of the views. `anObject` is a pointer to the Objective C object, which shall be inserted into the container.

The title text of the item is specified via `aTitle`, a resource handle of the icon (as can be queried using `WinQuerySysPointer ()`) is specified in `anIcon`.

Use this method for inserting new objects when in icon view or tree view, of if you plan to change the view of the container at run-time to one of these.

Figure 5.3 shows a cotainer window in icon view. The object to the right was inserted using `-insertObject: withTitle: andIcon:`.

- insertObjectList: aList;

Because inserting many objects one by one using the methods with names starting with `-insertObject:` is really slow, corresponding methods for inserting a list of objects are provided.

`-insertObjectList:` corresponds to inserting a list of objects using `-insertObject:`. `aList` is a pointer to some list object which must be an instance of either `SimpleList` or `KeyedList` (see pages 16 and 12 for more information on these classes).

- insertObjectList: aList withTitles: titleList;

`-insertObjectList: withTitles:` will insert a list of objects stored in `aList` into the container object. The titles for these objects must be stored in the list called `titleList`.

Every single title must be an object implementing the method `-stringValue` for access to the title string. `-stringValue` has to return a pointer to a string `(char *)`. This should be a pointer to the memory area the string is stored internally by the object.

Do not allocate a new memory block when `-stringValue` is called. The memory will not be freed by the container object.

In the utility library `objcutil.a` a class suitable for this task is already provided. This class is called `String`. See Section 3 on Page 18 for a description of the class and its instance methods.

- insertObjectList: aList withTitles: titleList andIcon: (ULONG) anIcon;

This method will insert a list of objects `aList` with the titles stored in `titleList` into the container object.

The icon resource specified by `anIcon` is used for display of the objects.

- insertObject: anObject parent: parentObject;

To establish a hierarchical ordering of the objects displayed in the container's *tree view* you must use either this method or one of the next two to insert objects.

`-insertObject: parent:` will insert an object `anObject` into the container control. The object is inserted as the *last child object* of `parentObject`. `parentObject` can either be `nil`, which causes the object to be inserted at the top level, or a pointer to an object already existing in the container.

- insertObject: anObject withTitle: (char *) aTitle parent: parentObject;

Just alike `-insertObject: withTitle:` this method inserts a single object `anObject` into the container control. `aTitle` is used as the title of the object. `parentObject` is the parent of the object as shown in the container's *tree view*.

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

To insert an object `anObject` having the title `aTitle` and an icon specified by `anIcon` into the container object use this method.

Just as with the last two methods, `parent` must be either `nil` or a pointer to an object already stored in the container control. In the first case, the `anObject` will be inserted at the top level, in the latter case `anObject` will be the *last child object* of `parentWindow`.

- arrange;

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`.

To use *tree view* the objects should be inserted using `-insertObject: parent:`, `-insertObject: withTitle: parent:` or `-insertObject: withTitle: andIcon: parent:`.

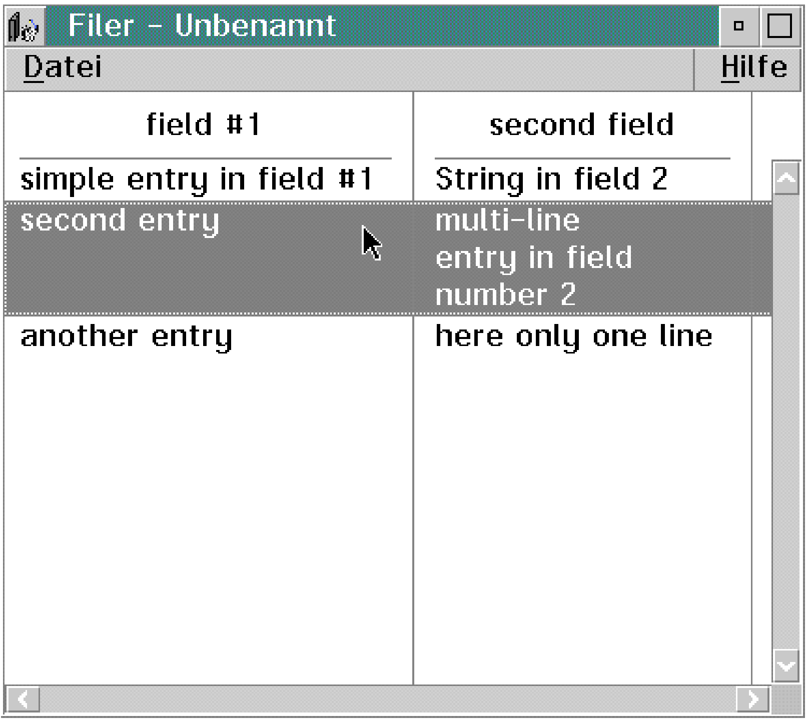

Figure 5.4: Container window in details view. The second record is selected.

- 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`.

Figure 5.4 shows a container window in detail's view. It holds three records, each of them consisting of two columns. The second record is selected. Notice, that the second column of the second record holds an entry taking up multiple lines.

- (ULONG) records;

`-records` queries the number of items currently stored in the container. The number of items (*records*) is returned.

- object;

This method returns a pointer to the object of the current record. The current record is set using the methods `-firstRecord`, `-lastRecord`, `-nextRecord`, `-previousRecord`, `-firstSelected` and `-nextSelected`.

- (CONTAINER_MINIREC *) findRecord: anObject;

If you want to modify the (display) properties of a single record and only know a pointer to the object, you can use `-findRecord:` to retrieve a pointer to the associated `(CONTAINER_MINIREC)` structure.

`-findRecord:` returns `NULL`, if the object `anObject` could not be found in the container, a pointer to the record structure otherwise.

The record is searched for by a simple incremental search. To determine, if an object has been found, the `-isEqual:` method of every object is used.

The default implementation of `-isEqual:` will return `YES` only if the two objects are the same one, i.e., if the pointer to the first object is equal to the pointer to the second object.

Some objects in the libraries, e.g. instances of `String`, will use a more *content-sensitive* implementation. For `String` objects, two objects are thought to be equal, if the string stored in both of them is the same.

You should implement `-isEqual:` and `-compare:` methods for your own classes if you plan to use some kind of *content-sensitive* comparisons. This would be the *cleanest* way to do this with Objective C.

- (CONTAINER_MINIREC *) firstRecord;

This method retrieves the first record in the container. A pointer to the data struction `CONTAINER_MINIREC`, which is used to store the data internally, is returned. The data of the record can be accessed using `-object`. It is stored in `recordBuffer`.

If you want to visit every record, consider using the following piece of code:

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

Here, the container object is assumed to be stored in a variable called `container`. The specific code for manipulating or querying each record is put in the `do` loop.

`-firstRecord` returns a pointer to the `CONTAINER_MINIREC` structure of the first record, if none exists, `NULL` is returned.

- (CONTAINER_MINIREC *) lastRecord;

In contrast to -firstRecord, this method searches for the last record stored in the container. If none is found, NULL is returned, otherwise a pointer to the CONTAINER_MINIREC structure of the last record.

Visiting all records, starting at the end can be accomplished using this piece of code.

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

The variables have the same meaning as shown before in -firstRecord.

- (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 *) childRecord;

-childRecord returns a pointer to the record structure for the child record of the selected record or NULL, if there is no child.

Be careful, calling this method will also set the instance variable recordBuffer anew. Using -findNext or findPrevious will from now on search the list of child records.

To visit every child record, you first have to *descend* into the *child level* and then use -nextRecord to visit all children of the current record. Use -parentRecord to leave the child level and go back to the parent level.

- (CONTAINER_MINIREC *) parentRecord;

-parentRecord returns a pointer to the record structure for the parent record of the selected record or NULL, if the record is at the top level of the hierarchy.

Be careful, calling this method will also set the instance variable recordBuffer anew. Using -findNext or findPrevious will from now on search the list of records in the upper level.

- (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.

- objectWithTitle: (char *) aTitle;

Using `-objectWithTitle:` you can retrieve a pointer to the first object stored with the title text `aTitle`. Only the top level hierarchy is searched at the moment.

`recordBuffer` is initialized with a pointer to the record structure of the record matching `aTitle` or `NULL`, if no record could be found.

- 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;

Calling `-hideRecord:` causes the current record to be hidden. The record is not deleted, it only will not be displayed by the container.

This method automatically calls `-invalidateRecord`.

If you want to hide more than one record, consider using `-hideSelectedRecords:` or `-hideNotSelectedRecords:` for performance issues. Hiding each record using `-hideRecord:` is painfully slow.

This method can be used as an *action method*.

- hideSelectedRecords: sender;

This method hides all selected records and thereafter causing them to be invalidated.

If you want to hide some records which are currently not selected, walk through the list of records using `-firstRecord` and `-nextRecord` and select or deselect some of the records using `select` or `deselect`. Afterwards call `-hideSelectedRecords:`.

Because this methods accepts a parameter `sender` it can be used as target of a command/action binding. The parameter `sender` itself is ignored.

- hideNotSelectedRecords: sender;

In contrast to `hideSelectedRecords:` this method hides all records which are *not selected.*

`sender` is ignored. This method can be used as an *action method.*

- showRecord: sender;

This method is the counterpart to `-hideRecord:`. It will cause the current record to be displayed in the containerobject if it was hidden previously.

- showAllRecords: sender;

`-showAllRecords:` changes the state of all hidden records to visible again and displays them.

- (BOOL) recordIsHidden;

If the current record is hidden, this method returns `YES`, otherwise `NO` is returned.

- removeAllColumns: sender;

This method will cause all columns to be removed. If in *details view*, the container display area will be empty after calling `-removeAllColumns:`. The records themselves still do exist.

- removeAllRecords: sender;

`-removeAllRecords:` will remove every record from the container control. The records are *not* freed automatically. You have to do this yourself.

- removeSelectedRecords: sender;

Calling this method will remove all selected methods from the container control. The objects are *not* freed automatically.

- (ULONG) columns;

`-columns` is used to query the total number of columns currently existing in detailss view.

- (FIELDINFO *) firstColumn;

As `-firstRecord` is used to set the current record to the first record, this method affects the internal buffer variable `columnBuffer`. The first column is put into this variable and `columnBuffer` is returned.

`columnBuffer` stores a pointer to the `FIELDINFO` structure of the current column.

If there are no columns, `NULL` is returned.

- (FIELDINFO *) lastColumn;

In contrast to `-firstColumn`, `-lastColumn` queries information about the last column stored in the container. If no columns exist, `NULL` is returned.

- (FIELDINFO *) nextColumn;

This method queries information about the next column. The search must have been initialized using `-firstColumn`.

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

```
.
.
  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;

`-columnTitle` returns a pointer to the `NULL`-terminated title string of the current column.

- (ULONG) columnTitleAttributes;

Every column stored in the container has two different attribute settings, one for the title data, and one for the data itself.

`-columnTitleAttributes` is used to query the title attribute settings of the current column. These settings can be modified using `-setColumnTitleAttributes`.

Table 5.8 on the next page shows the attributes which can be set or queried for the title data. Additionally, the appearance can be modified using the alignment flags shown in table 5.9 on the following page.

- (ULONG) columnDataAttributes;

Every column stored in the container has two different attribute settings, one for the title data, and one for the data itself.

`-columnDataAttributes` is used to query the data attribute settings of the current column. These settings can be modified using `-setColumnDataAttributes`.

Each column can display data of a specific data type. The data type is specified using the flags shown in table 5.10 on the next page.

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 5.8: 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 5.9: Container flags specifying alignment of data and title text

Flag	Description
CFA_BITMAPORICON	The data stored is a bitmap or icon.
CFA_STRING	The data stored is a text string. This is the only datatype currently supported. Don't specify one of the other flags!
CFA_ULONG	Data is an unsigned long integer.
CFA_DATE	Data is a date structure.
CFA_TIME	Data is a time structure.

Table 5.10: Flags specifying the data type of a field in a container

Flag	Description
CFA_SEPARATOR	Draw a vertical separator line to the right of the current column.
CFA_HORZSEPARATOR	Draw a horizontal separator line underneath the column title.
CFA_OWNER	Enable ownerdraw for this field. This is currently not supported by this class.
CFA_INVISIBLE	If this flag is set, the column is not visible.
CFA_FIREADONLY	This sets the data displayed to read-only mode.

Table 5.11: Miscellaneous flags specified for container column data.

Table 5.9 on the facing page shows all possible flags used for alignment of the data, table 5.11 shows all other possible flags which can be specified for column data.

- hideColumn: sender;

Calling `-hideColumn:` causes the current column to be hidden in the future. The data inside this column is preserved and can be restored using `-showColumn:`

This method can be used as an *action method*.

- showColumn: sender;

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

This method can be used as an *action method*.

- showAllColumns: sender;

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

This method can be used as an *action method*.

- (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 5.8 on the facing page and table 5.9 on the preceding page 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 5.9 on page 62, table 5.10 on page 62 and table 5.11 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.

This method can be used as an *action method*.

- 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.

This method can be used as an *action method*.

- sort: (ULONG) column;

Calling this method causes the records to be sorted by column number `column`. Do not use this method if your records are not prepared to hold different columns as needed in detail's view.

The sorting is simply done in ascending order by the ASCII values of the data represented as strings.

- sortByTitleWithCase: sender;

This *action method* will cause the objects to be sorted according to their object title. The sorting is performed *case-sensitive*. The compare function is *not* NLS-enabled, for simplicity, `strcmp()` is used at the moment.

- sortByTitleWithoutCase: sender;

This *action method* will cause the objects to be sorted according to their object title. The sorting is performed *case-insensitive*. The compare function is *not* NLS-enabled, for simplicity, `stricmp()` is used at the moment.

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

This method will handle most of the notification messages a container window can send. See the section *Methods implemented by the delegate* to get informed of the messages handled by this method.

Methods implemented by the delegate:

- beginEdit: sender;

Whenever the user starts an action of *direct editing* on some container item (either an object's title or some column data) this message is sent to the *delegate* object.

The return value is ignored. The cause of this message is the OS/2 message `CN_BEGINEDIT` originating from the container window.

- changeString: (CNREDITDATA *) editData;

If *direct editing* requires a new memory block to store the string this delegate method is called by the container object. The delegate object now has to make sure a memory block large enough to hold `editData->cbText + 1` bytes is available at the address pointed to by `editData->ppszText`. Keep in mind, that `editData->ppszText` is a *pointer to a string*, defined with `(PSZ *)` or in more common C datatypes using a `(char **)`.

To handle this message, you should check, if the memory block is large enough to hold `editData->cbText` bytes and perform corrective action if this is not the case. E.g. use

```
/*
 * free the memory block allocated previously for the string
 */
free (*(editData->ppszText));

/*
 * create a new block of memory large enough for storing the
 * newly edited string.
 */
*(editData->ppszText) = (char *) malloc (editData->cbText + 1);

/*
 * let the container control modify the string!
 */
return self;
```

to dynamically adjust to the edited data, or write something like this

```
/*
 * check, if the string is too large to fit in the fixed-length
 * field storing at most MAX_LENGTH characters. The field was
 * allocated as an array of characters large enough for MAX_LENGTH
 * + 1 characters.
 */
if (editData->cbText > MAX_LENGTH)
  return nil;  /* do not modify the string */
else
  return self; /* modify the string
```

You are not allowed to use any of the container methods or send the container window any OS/2 messages when this method is called until the `-stringChanged:` delegate method is called.

If the return value of this method is `nil` no change will be performed, otherwise, the newly allocated memory block will be filled with the new string which was edited. The cause of this message is the OS/2 message `CN_REALLOCPSZ` originating from the container window.

- emphasisChanged: (NOTIFYRECORDEMPHASIS *) recordEmphasis;

Whenever the user selects an object in the container, this object is said to receive the *record emphasis*.

After the record emphasis is changed, the delegate object is notified of this by calling its `-emphasisChanged:` method.

The return value is ignored. The cause of this message is the OS/2 message `CN_EMPHASIS` originating from the container window.

- enterPressed: (NOTIFYRECORDENTER *) recordEnter;

As soon as the user presses the *enter* key in the container object while a container item is selected or an object is double-clicked, the delegate object is sent the *-enterPressed:* message. The `NOTIFYRECORDENTER` structure provides information about the selected record.

The return value of this method is ignored. The cause of this message is the OS/2 message `CN_ENTER` originating from the container window.

- showContextMenu: (PRECORDCORE) recordCore;

When the user activates a *context menu* while in the container control, the delegate object is sent a `-showContextMenu:` message. `recordCore` is either `NULL` if the pointer is not over a container record, otherwise a pointer to the `CONTAINER_MINIREC` structure of the record beneath the mouse pointer is provided here.

The return value of this method is ignored. The cause of this message is the OS/2 message `CN_CONTEXTMENU` originating from the container window.

- stringChanged: (CNREDITDATA *) editData;

After direct editing a container item has ended, the delegate object is sent this message. After receiving this method, you can safely call methods of the container object or send the container window OS/2 messages.

The new string can be found in `*(editData->ppszText)`.

The return value is ignored. The cause of this message is the OS/2 message `CN_ENDEDIT` originating from the container window.

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

Every other OS/2 message sent by the container window can be processed by this delegate method. The message is specified by `msg`, the parameters are `mp1` and `mp2`.

If the return value of this method is not null, the messages are assumed to be processed. Otherwise, the messages are passed to the container's owner window which is normally an instance of `StdDialog` or `StdWindow`.

5.9 DelegateWindow

Inherits from: WINDOW : OBJECT

Class description:

This class is the common superclass for most of the window objects in this library. It provides a single instance variable `delegate` and methods to read and set this variable.

The concept of delegate objects is explained in detail in the *Tutorial* document.

Instance Variables:

id delegate;

This instance variable stores a pointer to the window's delegate object.

Methods:

- init;

This method just initializes the object by calling its superclasses' `-init` method and sets `delegate` to `nil`.

- setDelegate: anObject;

Using `-setDelegate:` you set the delegate object of the window to `anObject`.

- delegate;

This method returns a pointer to the *delegate object*. It is stored in the instance variable `delegate`.

5.10 EntryField

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving, Selection, Value

Class description:

The class `EntryField` was designed to simplify access to OS/2 PM entry field controls. Using the methods implemented for this class the programmer can control all interesting features of this predefined window class.

The text typed into the entry field can be accessed via the inherited methods `setTest:` and `text:`. In future releases of this class library methods for automatically checking typed input will be provided for *integers*, *floating point numbers*,...

By adopting the procotol `Selection` simple access to Clipboard operations as *copy* or *paste* is provided. See also the description of this protocol on page 122.

The `Value` protocol will allow differnt access to the string entered by the user.

Instance Variables:

int textLimit;

This variable stores the maximum length of the string which can be entered in the entry field. Use `-textLimit` and `setTextLimit:` to access this variable.

Methods:

Flag	Description
ES_LEFT	The text in the entry field is left-justified. This style is used when neither ES_LEFT, nor ES_RIGHT nor ES_CENTER is specified.
ES_RIGHT	The text in the entry field is right-justified.
ES_CENTER	The text in the entry field is centered.
ES_AUTOSIZE	When this flag is set, the text will be sized to fit in the entry field.
ES_AUTOSCROLL	The text in the entry field is scrolled to the left or right if it is longer than would fit in the entry field.
ES_MARGIN	A margin is drawn around the entry field.
ES_READONLY	The entry field will be created in *read-only* mode.
ES_UNREADABLE	Every character in the text is displayed as an asterisk. This is useful when querying passwords.
ES_COMMAND	This style classifies the entry field as a command entry field. This style should be applied to at most one entry field per dialog or window.
ES_AUTOTAB	When this flag is set, the focus is moved to the next window when a character is appended to the text.

Table 5.12: `ES_xxxx` styles used at creation of an entry field

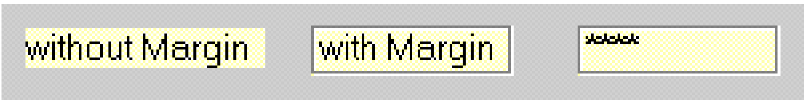

Figure 5.5: In this figure you can see (from left to right) an `EntryField` without a margin, one with a margin and an entry field with margin and the style option BS_UNREADABLE

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

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

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

Association to an existing PM entry field window should be done by using `-associate:`.

A newly created `EntryField` object is not automatically inserted as a child window of its parent. Use `[parent insertChild: entryfield]` where `parent` is the parent window and `entryfield` is the newly created entry field object.

Table 5.12 (Page 68) shows most of the available `ES_xxxx` flags used at creation of the entry field.

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

Figure 5.5 on page 68 shows three possible forms of how an entry field can look.

- (BOOL) changed;

`-changed` returns `YES` if the text displayed in the entry field has changed since the last call to this method, `NO` otherwise.

- (BOOL) readOnly;

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

This method returns `YES` if the entry field is in *read-only* mode, `NO` otherwise (*read-write*).

- setReadOnly;

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

- setReadWrite;

`-setReadWrite` switches the entry field 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 entry field. `limit` is this maximum number of characters.

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

- (int) textLimit;

This method returns the value of the instance variable `textLimit`. This is the total number of characters the user can enter into the entry field.

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

This method performs the handling of some OS/2 messages. The messages are forwarded to the delegate window.

At the moment, the control messages (message `WM_CONTROL`) `EN_CHANGE`, `EN_KILLFOCUS` and `EN_SETFOCUS` are handled.

Methods implemented by the delegate:

- focusLost: sender;

Whenever the entry field leaves the focus, i.e., it is deselected, this delegate method is called. `sender` is a pointer to the entry field control.

- focusReceived: sender;

If the entry field gets selected it is said to receive the *input focus*. In this case, the delegate method `-focusReceived:` is called, passing a pointer to the entry field object to the delegate method.

- textChanged: sender;

This method is called when the user has changed the contents of the entry field window. `sender` is a pointer to the sending `EntryField` instance.

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

All other notification messages not handled by the object are forwarded to this method.

5.11 FactoryWindow

Inherits from: DelegateWindow : Window : Object

Class description:

To be able to catch the OS/2 window notification messages, every window control displayed on screen is associated with a so-called *owner* window. Creating and handling this owner window is done by the methods of this class, `FactoryWindow`. Therefore most of the window classes are derived directly or indirectly from this class.

In more detail this means, that every OS/2 message sent from an OS/2 window object is sent to its owner window. These messages are processed by the owner window and routed to the "local" `-handleMessage: withParams: and:` methods of the objects themselves. There they can be processed easily.

Don't be afraid of creating an extra window object (the owner) for every OS/2 window you wish to display, this extra window is neither time nor space-consuming to be worth thinking of.

Instance Variables:

HWND owner;

This instance variable will hold the window handle (type `HWND`) of the owner window. The OS/2 window class used for this kind of window named `ObjcWindow` and defined in "`pm/Window.h`" using the constant `WINDOW_CLASS`.

Methods:

- initIn: (Window *) parent;

`-initIn:` creates the owner window as a child of `parent`. The window is created invisible. It cannot be displayed on the screen.

- associate: (HWND) hwnd;

As an alternative constructor method `-associated:` is used to associated an already existing OS/2 window control with an Objective C object. This method takes care that an owner window is created and the OS/2 messages originating from `hwnd` are routed towards the owner window.

- destroy;

This method destroys the owner window and subsequently calles its superclasses' `-destroy` method.

- setWindow: (HWND) aWindow;

`-setWindow:` sets the inherited instance variable `window` to `aWindow` and modifies the message processing order so that all OS/2 notification messages originating from `aWindow` are routed towards the owner window and—subsequently—to the "local" `-handleMessage: withParams: and:` methods.

- specialClass;

This method just returns `self`. If any window object can respond to this message, it is known that it is derived from `FactoryWindow` and the message handling will not work as expected for "normal" OS/2 window controls.

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

This method only catches the *WM_DESTROY* message and returns `0` in this case.

Otherwise, all messages are routed towards the owner of `owner`.

Most descendants of `FactoryWindow` will overwrite this method to handle the associated OS/2 window's notification messages.

5.12 FileDlg

Inherits from: OBJECT

Class description:

To enable the users of your programs to perform simple file selection as customary in OS/2 PM applications, the class `FileDlg` is provided. This dialog lets the user select a single file on one of the disks, supporting this action with various features shown in figure 5.6.

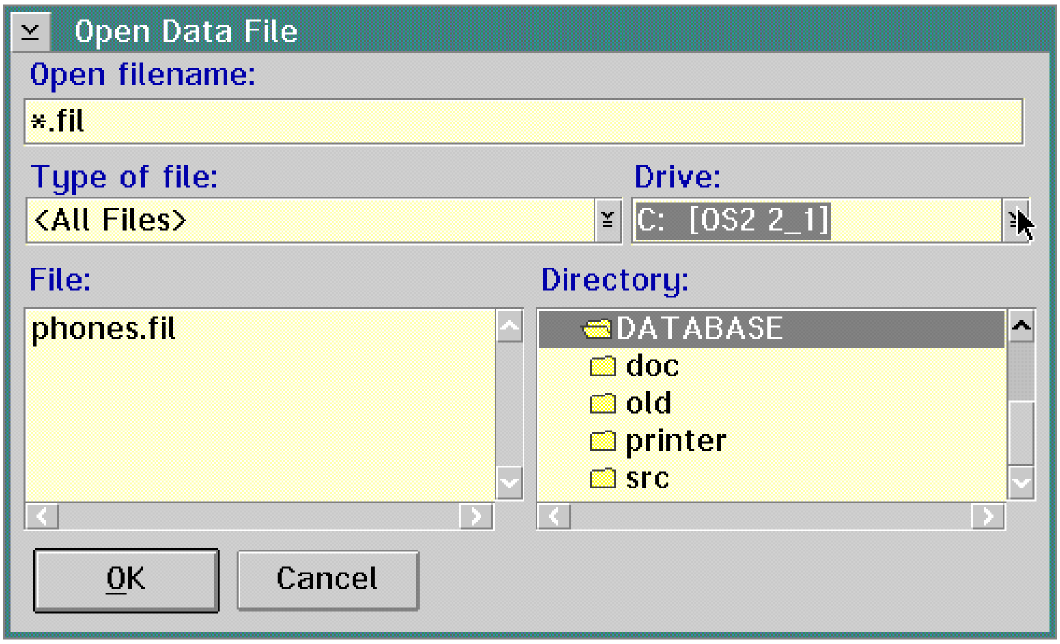

Figure 5.6: Sample file dialog

A file dialog can be created either for opening a file (see `-initForOpen: withFilter:`) or for saving a new file (see `-initForSaveAs: withFilter:`).

The dialog provides user interface objects to select the *disk drive*, the *directory* and the *filename* of the file to be opened or saved.

Depending on the initializing method either a `Open` or a `Save` button is present to close the dialog.

Instance Variables:

FILEDLG fileDlg;

This variable is used to store the structure used to create and display a file dialog. The result of a file dialog after closing is also stored here. There is normally no use to access this structure directly.

Methods:

- init;

This method initializes the structure `FILEDLG` to zeros and calls the `-init` method of its superclass.

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

This method initializes a dialog for opening a file. The dialog is always centered in its parent window, which is specified as a parameter to `-runModalFor:`.

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 its 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`.

- free;

As expected, `-free` frees all resources claimed for by this object. The `-free` method of its superclass is called in the end.

- 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;

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 list box 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 5.13: Window flags which can be specified for file dialogs

`-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).

Every dialog *must* either contain `FDS_OPEN_DIALOG` or `FDS_SAVEAS_DIALOG`. Otherwise an error code is returned when calling `-runModalFor:`.

Other flags, which can be specified are shown in table 5.13.

- setOKTitle: (const char *) aTitle;

Using `-setOKTitle:` the application can set the title string of the *OK*-Button displayed in the file dialog. Normally the title string is either `Open` or `Save`.

- runModalFor: sender;

`-runModalFor:` displays the file dialog and runs it as a modal dialog box in respect to `sender`, which must be an instance of `Window` or one of its subclasses.

On successful completion this method returns a pointer to the `FileDlg` object, otherwise `nil` is returned.

The result of running the dialog can be queried using `-result`.

- (char *) fileName;

After successful execution of the file dialog, this method returns a pointer to the filename which was selected.

This method only returns a valid string after successful execution using `-runModalFor:`.

- (ULONG) result;

Using this method after running a file dialog using `-runModalFor:` you can query the result of the dialog. This will be `DID_OK` on successful completion, if the user selected a file and pressed the *OK* button or `DID_CANCEL` if the user cancelled the dialog execution using *Cancel*.

5.13 Frame

Inherits from: WINDOW : OBJECT

Class description:

`Frame` is a class designed to provide an interface to OS/2 PM windows of class `WC_FRAME` (Frame windows).

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

5.14 Help

Inherits from: OBJECT

Adopted Protocols: ARCHIVING

Class description:

This class is used to enable your applications to use the PM online Help feature. Just create a help file using e.g. ipfc.exe and add an instance of "Help" to your application.

Follow these steps to add help to your application:

1. Add a help menu to your main window or help buttons to your dialog windows.
2. Allocate a help object and inialize it.
3. Associate the help object with your main window. This step can eventually left out. Help will also work without setting an association. For normal use, you should associate the `Help` object with your main window.
4. Create command bindings for your menu items/buttons

Before closing your application, free the help object again.

The object can display help for the standard topics:

- "*Help for Help*" will provide help for using the OS/2 help facility itself.
- "*Extended Help*" will display the first (the topmost) help page.
- "*Help index*" will display the help index of your help file.

Additionally the object can display a specified help page.

To enable online help in your programs, you have to create a `HELPTABLE` resource and link the binary resource template to your application. In the following a sample resource definition is shown:

```
/*
 *  Help table definition
 */

HELPTABLE   1000
{
  HELPITEM   1000,   20800,   20800
}

/*
```

```
 *  Main window subtable, includes menu item help
 */

HELPSUBTABLE 20800
SUBITEMSIZE     2
{
}
```

This resource template defines a `HELPTABLE` resource having the resource identifier `1000` which is the default help table identifier. The first help page will have the *help resource identifier* `20800`. This *help resource identifier* is specified in the *IPF source code* of your help file.

Instance Variables:

BOOL helpEnabled;

This boolean variable is a flag showing, if the help object was initialized correctly and the online help feature can be used.

HWND helpInstance;

`helpInstance` is used to store the OS/2 window handle (type `HWND`) of the help window.

id helpFileName;

This variable is a pointer to an instance of `String`. The object is used to store the name of the help file. Access is provided via `-setStringValue:` and `-stringValue`.

id helpWindowTitle;

Here a `String` object is pointed to used to store the title of the help window.

ULONG helpTable;

`helpTable` is the identifier of the help table of the help file. This help table must be specified in the resource file.

HAB hab;

`hab` is used to store the anchor block handle of the instance of `StdApp` used in your program.

Methods:

- init;

This method initializes all instance variables to default values.

`-createHelpInstance` is *not* called automatically.

- initForApp: (StdApp *) anApp helpfile: (char *) fileName;

This method only calls the default initializer method `-initForApp: helpFile: withTitle: helpTable:` and passes the parameters to this method. The help window title is set to "*Help*". The default help table is assumed to have the resource identifier `1000`.

`-createHelpInstance` is called automatically in the end, so the help object will be usable after this method exits.

- initForApp: (StdApp *) anApp helpFile: (char *) fileName withTitle: (char *) helpTitle;

This method passes the parameters to the default initializer method and assumes the resource identifier of the default *HELPTABLE* resource to be `1000`.

`-createHelpInstance` is called automatically in the end, so the help object will be usable after this method exits.

- initForApp: (StdApp *) anApp helpFile: (char *) fileName withTitle: (char *) helpTitle helpTable: (ULONG) helpTable;

This will be the default initializer method for objects of the `Help` class.

The instance variables are initialized, `hab` is set to `[anApp hab]`; the name of the help file is specified by `fileName`, the title of the help window will be `helpTitle`. `helpTable` is the resource identifier of the default `HELPTABLE` resource.

`-createHelpInstance` is called automatically in the end, so the help object will be usable after this method exits.

- free;

`-free` deletes the objects allocated by this instance of `Help` and frees all other resources consumed by this object.

- associateWith: aWindow;

Before using a help object with your main window, you should associate the help object with it. This is accomplished using this method. `aWindow` is a pointer to the window object you wish the `Help` object to be associated with. Normally, this will be a pointer to an instance of `StdWindow` or `MainWindow`.

- createHelpInstance;

A call to this method will create the OS/2 help instance used to provide help. You only have to call this method manually if you initialized the object using `-init`. All other initializer methods will call `-createHelpInstance` automatically after completion.

- helpForHelp: sender;

Display information on how to use the help system. This method can be bound directly to a menu item.

- helpExtended: sender;

Display the first help page according to the help table specified. This method can be used as an action method.

- helpIndex: sender;

This method will cause the help index to be displayed if one is available. It can be bound directly to a menu item.

- helpFor: sender;

This method is used to display a special help topic. The topic is specified by the sending object. The number of the topic (which is the number also specified in the *IPF source file*) is specified by the `tag` instance variable of `sender`. It is queried via `[sender tag]`. This method can be bound directly to a menu item or a button control.

If this method is bound to a button or a menu item via `-bindCommand: withObject: selector:`, the sending object is the window, in which the control resides.

If bound to a button using the button's `-bindWith: selector:` method, the sending object will be the button itself.

- helpForTopic: (int) topic;

If you want to provide help for a special help topic you can call `-helpForTopic:` directly. In this case `topic` is the help resource identifier of the help page you want to be displayed.

Again, notice that `topic` is not related to the binary resource definition of the help table. The value of topic is only influenced by the help topic resource identifiers you specify in the *IPF source code* of your help file.

- helpFileName;

This method returns a pointer to the `String` object storing the name of the help file.

Modifying this object, i.e., setting a new help file name, will only make sense, if no help instance was created previously, which means that the object must have been initialized with `-init` only.

A subsequent call to `-createHelpInstance` will create an OS/2 help instance usable in your application.

- helpWindowTitle;

This method returns a pointer to the instance of `String` used to store the title of the help window.

Modifying this object, i.e., setting a new help window title, will only make sense, if no help instance was created previously, which means that the object must have been initialized with `-init` only.

A subsequent call to `-createHelpInstance` will create an OS/2 help instance usable in your application.

- (ULONG) helpTable;

`-helpTable` returns the identifier of the default help table in the binary resource definition.

- setHelpTable: (ULONG) anId;

This method sets the help table identifier of the default `HELPTABLE` in the binary resource file to `anId`. Changing this only makes sense before calling `-createHelpInstance`.

5.15 InterfaceFile

Inherits from: OBJECT

Adopted Protocols: ARCHIVING

Class description:

As most of the classes in the library now provide support for archiving, a class is provided to group many dialog objects (and also command bindings etc.) into one object to simplify archiving and unarchiving.

This class is used as the root class for archiving by all files created by the *Interface Editor program.*

Instances of this class can store objects, command bindings, bindings of instance variables of type `id` to other objects also stored and objects not archived but created at run-time.

Only standard objects are supported now.

The objects are stored with a unique string key associated. The objects can be queried either by index or by key. Querying by key is recommended because the index can change after archiving/unarchiving.

Methods:

- init;

Initialize the object

- free;

Free the memory allocated by this object

- freeObjects;

Free this object and all objects stored. This will change in the future so that this method will only free the objects stored in the interface file and not the interface file itself.

At the moment, this is the preferred destructor method.

- objectWithTitle: (char *) aTitle;

Search for an object having the title `aTitle`. If such an object could be found a pointer to it is returned. Otherwise `nil` is returned.

5.16 ListBox

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

`ListBox` is a class designed to be associated to the OS/2 PM class `WC_LISTBOX`. The class provides methods to give access to the items in the list box window.

Methods:

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

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

Figure 5.7 on page 79 shows two forms of list box windows. The left is a standard list box with only one scroll bar—a vertical one. The right list box control also has a horizontal scroll bar.

How a list box window appears on screen depends on what control flags you specify in the parameter `flags`. Table 5.14 shows which control flags are possible and what effect is caused by specififying 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;

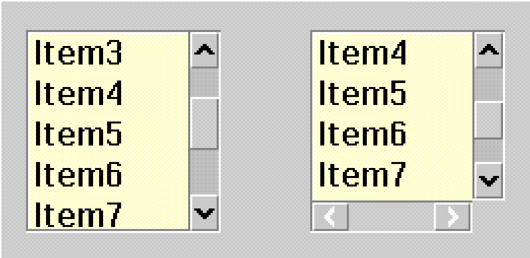

Figure 5.7: Here you can see a standard list box (left) and a list box window with an additional horizontal scroll bar.

Flag	**Description**
LS_HORZSCROLL	This flags adds a horizontal scroll bar to the list box window, if it is specified at creation.
LS_MULTIPLESEL	Normally only one item in the list box 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 list box window.
LS_OWNERDRAW	This flag tells the list box not to draw the items itself. Appropriate messages are sent to the owner of the list box, which has to draw them.
LS_NOADJUSTPOS	This flag tells the list box 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 5.14: `LS_xxxx` styles used at creation of a list box window

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

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

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

- (SHORT) count;

`-count` returns the number of items which are currently in the list box.

- (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.

- (SHORT) itemTextLength: (SHORT) pos;

This method returns the length of the item text of the item at position `pos`. Only the number of characters in the item text is returned. Don't forget to allocate an extra character for the `NULL` at the end of the string before querying via `-item: text:`.

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

`-item: text:` copies the item text of the item at position `pos` in the list box into the array of characters pointed to by `buffer`. This method assumes, there is enough space in `buffer` to hold all of the item text, including the `NULL` at the end of the text.

This method returns `buffer`.

If `buffer` is `NULL`, a string is allocated via `malloc()` to hold all of the item text. This string must be freed by the programmer later using `free()`.

- (SHORT) selectItem: (SHORT) pos;

On calling this method the specified item at position `pos` will be selected. If `pos` is out of the range of the list box items, nothing happens.

- (SHORT) deleteItem: (SHORT) pos;

`-deleteItem:` will delete the item at position `pos`. If `pos` is out of the range of the list box items, no item gets deleted.

Deletion of the currently selected item can be accomplished by sending this message:

```
[listbox deleteItem: [listbox selected]];
```

Here `listbox` is a pointer to the list box object.

- (SHORT) deleteAll;

`-deleteAll` deletes all items in the list box.

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

This method is the *message handler* for the OS/2 messages originating from the OS/2 list box object. The `WM_CONTROL` messages `LN_KILLFOCUS`, `LN_SETFOCUS`, `LN_ENTER`, `LN_SCROLL` and `LN_SELECT` are processed and the appropriate delegate methods are called if implemented.

Methods implemented by the delegate:

- enterPressed: sender;

Whenever the user presses the *enter* key or double-clicks on an item in the list box control this delegate method gets called.

`sender` is a pointer to the `ListBox` object. The return value of this method is ignored.

- focusLost: sender;

If the list box control loses the focus, i.e., the list box is deselected, this delegate method will be called to permit your application perform some specific actions which are necessary.

`sender` is a pointer to the `ListBox` object. The return value of this method is ignored.

- focusReceived: sender;

`-focusReceived:` is called when the list box control receives the focus.

`sender` is a pointer to the `ListBox` object. The return value of this method is ignored.

- itemWasSelected: sender;

Every time an item in the list box control is selected by a user action this method of the delegate object is called. Here you can perform additional functions necessary on selection of an item in the list box.

`sender` is a pointer to the `ListBox` object. The return value of this method is ignored.

- listBoxWillScroll: sender;

Before the list box control will scroll the list of items displayed in its interior, `-listBoxWillScroll:` gets called.

`sender` is a pointer to the `ListBox` object. The return value of this method is ignored.

5.17 MainWindow

Inherits from: StdWindow : ActionWindow : DelegateWindow : Window : Object

Class description:

This class is only provided for simplification purposes. It extends its superclass `StdWindow` by means of specifying default flags when using the methods `-initWithId:` and `-initWithId:andFlags:`. Additionally the window will post a `WM_QUIT` message when it is closed causing the application program to quit.

By default, the flags

- `FCF_MINMAX`
- `FCF_SHELLPOSITION`
- `FCF_SYSMENU`
- `FCF_TASKLIST`
- `FCF_TITLEBAR`

are specified at window creation when using `-initWithId:`.

Methods:

- initWithId: (ULONG) anId;

This method only calls `initWithId: andFlags:` of its superclass `StdWindow` while specifying the window flags as described above.

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

This method only calls `initWithId: andFlags:` of its superclass `StdWindow`.

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

This method extends the functionality of its superclass `StdWindow` by posting a `WM_QUIT` message if the window gets closed.

5.18 Menu

Inherits from: Window : Object

Adopted Protocols: Archiving

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).

Instance Variables:

id itemList;

`itemList` is used to store a list of menu items (instances of `MenuItem` belonging to the menu. For this purpose a `SimpleList` object is used.

Methods:

- init;

`-init` just initializes a `Menu` object. `itemList` is initialized to be an empty list. No window object for the menu is created.

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

This method just calls the default initializer method for this class, `-initWithId: andFlags: in:` and passes the parameters `anId` and `flags`. The parent window is specified to be `nil`.

Table 5.15 shows which flags can be used when creating a menu window.

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

Flag	Description
MS_ACTIONBAR	This style causes the single menu items in the menu to be displayed side-by-side. This style should be used when creating a *menu bar* for some window.
MS_CONDITIONALCASCADE	The style flag `MS_CONDITIONALCASCADE` causes the (sub-)menu to be a conditional cascade menu. Such menus get opened only if the arrow to the right is explicitly selected.
MS_TITLEBUTTON	This one can only be used together with `MS_ACTIONBAR`. Such menus can be used as button controls in the title bar of the parent window.
MS_VERTICALFLIP	Specifying this flag will cause pull-down menus to be displayed *above* the menu identifier not below as would be normal.

Table 5.15: Menu creation flags

`-initWithId: andFlags: in:` is the default initializer method of the class `Menu`. It will create an empty menu.

See Table 5.15 for a list of style flags that can be used. `parent` is a pointer to the parent window of the menu.

`anId` is the PM identifier of the menu window.

- free;

`-free` will first free all menu items and submenus stored in `itemList` and then free the resources and destroy the menu window.

- insertItem: anItem at: (SHORT) pos;

Using `-insertItem: at:` you can insert new menu items (instances of `MenuItem`) `anItem` at the position specified by `pos`.

To append a new menu item to the end of the items already in the menu, specify `pos` as `MIT_END`.

This method will automatically take care of

- 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:`

- popup: sender;

This *action method* is used to display a popup menu on the screen. `sender` must be a window object. It is used as the owner of the popup menu. All commands are sent to `sender`.

- itemList;

This method returns a pointer to the list of menu items for this menu. The returned object is an instance of `SimpleList`.

5.19 MenuItem

Inherits from: OBJECT

Class description:

`MenuItem` is used to represent single menu items or sub-menus in a menu—either of pull-down or pop-up type.

To create a menu having three menu items of which the second is a sub-menu containing two items, e.g. use the following piece of code:

```
.
.
.

id menu    = [[Menu alloc] initWithId: 1000 andFlags: 0];
id submenu = [[Menu alloc] initWithId: 2000 andFlags: 0];

/*
 * First, insert two menu items into the sub-menu
 */
[submenu insertItem: [[MenuItem alloc] initWithId: 2001
                                         andTitle: "sub-menu item 1"]
                 at: MIT_END];

[submenu insertItem: [[MenuItem alloc] initWithId: 2002
                                         andTitle: "sub-menu item 2"]
                 at: MIT_END];

/*
 * now insert three menu items into the main menu, the second
 * one being a sub-menu
 */
[menu insertItem: [[MenuItem alloc] initWithId: 1001
                                      andTitle: "menu item 1"]
              at: MIT_END];

[menu insertItem: [[MenuItem alloc] initWithId: 1002
                                      andTitle: "sub-menu"
                                       subMenu: submenu]
              at: MIT_END];

[menu insertItem: [[MenuItem alloc] initWithId: 1003
                                      andTitle: "menu item 3"]
              at: MIT_END];

.
.
.
```

This menu could e.g. be displayed by using `[menu popup: nil]`.

Instance Variables:

ULONG pmId;

`pmId` is used to store the Presentation Manager identifier of the menu item. This identifier is used to bind commands from menu items to action methods.

char * title;

This variable stores the title text of the menu item. It is allocated newly when initialized and freed by `-free`.

id subMenu;

If the menu item represents a sub-menu, `subMenu` is used to store a pointer to the sub-menu object.

Methods:

- init;

`-init` simply initializes an instance of `MenuItem`. `pmId` is set to `0`, no title text and no sub-menu are assumed to exist.

Use `-setPmId:`, `-setTitle:` and `-setSubMenu:` to complete initialization or use one of the other initializer methods.

- initWithId: (ULONG) anId andTitle: (char *) aTitle;

`-initWithId: andTitle:` is the default initializer for "normal" menu items not representing a sub-menu.

`anId` is the Presentation Manager identifier of the new menu item, `aTitle` will be the title text of the item.

- initWithId: (ULONG) anId andTitle: (char *) aTitle subMenu: aSubMenu;

This method is the default initializer for menu items representing sub-menus. `anId` is the Presentation Manager identifier of the newly created menu item, `aTitle` the title text and `aSubMenu` a pointer to the sub-menu (an instance of `Menu`).

- free;

This method frees the memory allocated for the object and a sub-menu if one was associated with this item.

- setTitle: (char *) aTitle;

`-setTitle:` sets the title of the menu item to `aTitle`.

- (char *) title;

`-title` returns the title of the menu item.

- setPmId: (ULONG) anId;

This method sets the Presentation Manager identifier of this menu item to `anId`.

- (ULONG) pmId;

`-pmId` returns the Presentation Manager identifier of this menu item.

- setSubMenu: aSubMenu;

For a menu item to become a sub-menu, just create a sub-menu (a normal instance of `Menu` and use `-setSubMenu:` to set this menu item to represent a sub-menu.

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 5.16: `MLE_xxxx` styles used at creation of an MLE window

- subMenu;

If the menu item is a sub-menu, a pointer to the `Menu` instance of this sub-menu is returned by this method. Otherwise `nil` is returned.

5.20 MultiLineEntryField

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving, Selection

Class description:

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

Basic editing commands for the text in a multi-line entry field is supported via instance methods. Thse are `-delete: at:`, `-insertText:`, `-insertText: at:` and `-appendText:`.

The whole text in the MLE can be accessed via `setText:` and `text:`.

You must take care that it is not possible to pass more than *64KBytes* of data between the multi-line entry field and your application at once. `-setText:` and `-text:` will only affect the first *64KBytes of data.*

Methods:

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

Using `-initWithId: andFlags: in:` you can create an instance of class `MultiLineEntryField` and an OS/2 PM *MLE window* from scratch. `anId` is the PM identifier of the window, `flags` are the flags specified at creation of the MLE. `parent` represents the parent window of the object, where the MLE shall be inserted.

Table 5.16 lists all possible style flags to be used for instances of this class.

- (long) indexForLine: (long) line;

The text in a multi-line entry field is not organized in any way. Because most access to the text will refer to a special line, this method is provided to get the index of the first character in line `line`.

E.g. to delete the first 5 characters in line 6, you could use

```
[mle delete: 5 at: [mle indexForLine: 6]]
```

- delete: (unsigned long) count at: (long) index;

`-delete: at:` will delete `count` characters starting at position `index` in the text stored in the multi-line entry field.

The indexing is a per-character index starting at 0 for the first character in the multi-line entry field and ends at the last character. Use `-indexForLine:` to query the index of the first character of a specific line of text.

- insertText: (char *) aText;

`-inserText:` will insert the string `aText` into the multi-line entry field at the current position. `aText` must be a pointer to a `NULL`-terminated string not longer than *64KBytes*.

- insertText: (char *) aText at: (long) index;

`-inserText:` will insert the string `aText` into the multi-line entry field at the position specified by `index`. `aText` must be a pointer to a `NULL`-terminated string not longer than *64KBytes*.

- appendText: (char *) aText;

Using `-appendText:` you can add the string `aText` to the end of the text already stored in the multi-line entry field.

- clearAll: sender;

This *action method* is used to delete all text in the multi-line entry field. The method can be directly connected to a button or a menu item.

5.21 NoteBook

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

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

Methods to add, delete and modify notebook pages are provided.

A notebook is a window control able of displaying many different dialog windows. The dialog windows are ordered and put onto notebook pages and the user can turn the pages and switch to the dialogs as he wants. See the *Settings* notebook of most WPS objects for an example of this.

Every notebook page can display a single dialog window and can have a line of text displayed in the *status line* of the window. Additionally, you can create *major* and *minor* tabs to support easy navigation through the different notebook pages.

Methods:

Flag	**Description**
BKS_SOLIDBIND	Use a solid binding for the notebook window. This is the default, if BKS_SPIRALBIND is not specified.
BKS_SPIRALBIND	Use spiral binding.

Table 5.17: Flags determining the type of binding applied to a notebook control

Flag	**Description**
BKS_BACKPAGESBR	This is the default intersetion flag if no flag described in this table is specified. It causes the intersection of the back pages to be in the bottom right corner.
BKS_BACKPAGESBL	Given this flag is specified, the intersection of the back pages will be located in the bottom left corner.
BKS_BACKPAGESTR	Put the intersection of the back pages into the top right corner.
BKS_BACKPAGESTL	This flag causes the intersection to be in the top left corner.

Table 5.18: Notebook flags controlling the intersetion of the notebook's back pages

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

Using this method you can create a notebook window having the Presentation Manager identifier `anId` as a child window of `parent`.

The look & feel of the notebook window is determined by the `flags` parameter.

The flags parameter is constructed by logically *or*ing at most one flag of each group of flags. These groups are:

1. flags controlling the *binding* of the notebook (see Table 5.17),
2. flags determining where the *intersection* of the notebook's back pages is displayed (Table 5.18),
3. the flags for controlling the *location* (Table 5.19) and
4. the *shape* of the major tabs (see Table 5.20).
5. Additionally the *alignment for the tab text* as shown in Table 5.21 and
6. the *alignment of the status line text* (Table 5.22) can be specified.

Only one flag from each group can be specified. If no flag out of a single group is specified, the one denoted as being the *default* flag is assumed.

Flag	**Description**
BKS_MAJORTABRIGHT	Put the major tabs to the right side of the notebook window. This is the default.
BKS_MAJORTABLEFT	This flag will put the major tabs to the left.
BKS_MAJORTABTOP	`BKS_MAJORTABTOP` will cause the major tabs to be arranged on top of the notebook window.
BKS_MAJORTABBOTTOM	To put the major tabs onto the bottom of the notebook, specify this flag at window creation.

Table 5.19: Flags controlling the location of the major tabs in a notebook control

Flag	Description
BKS_SQUARETABS	Use square shaped major tabs. This is the default.
BKS_ROUNDEDTABS	Use rounded major tabs.
BKS_POLYGONTABS	This flag causes the major tabs to be shaped in a polygonal form.

Table 5.20: To define the shape of the major tabs in a notebook control use the flags shown in this table

Flag	Description
BKS_TABTEXTCENTER	Center the tab text. This is the default flag.
BKS_TABTEXTLEFT	Align the text appearing on the major tabs to the left.
BKS_TABTEXTRIGHT	This flag will cause the tab text to be displayed right-justified.

Table 5.21: These flags are used to specify the alignment of the tab text in a notebook

- (ULONG) createTopPageWithStatusText: (BOOL) stFlag;

This method is used to create a new *top page*[2] for the notebook window.

`stFlag` determines whether status text shall be displayed on the top page. `-createTopPageWithStatusText:` returns the notebook's page identifier of the newly created page. This identifier is used to

- set a dialog window to be displayed on the page,
- set or modify the status text of the page if display of it is enabled, and
- is used to set the text displayed on the major tab if the page has one.

- (ULONG) createMajorTopPageWithStatusText: (BOOL) stFlag;

This method is used to create a new *top page* containint a major tab for the notebook window.

`stFlag` determines whether status text shall be displayed on the top page. `-createMajorTopPageWithStatusText:` returns the notebook's page identifier of the newly created page. This identifier is used to

- set a dialog window to be displayed on the page,
- set or modify the status text of the page if display of it is enabled, and
- is used to set the text displayed on the major tab if the page has one.

[2]the first page in a notebook control is called the *top page*

Flag	Description
BKS_STATUSTEXTLEFT	Display the status line text left-justified. This is the default flag.
BKS_STATUSTEXTRIGHT	Align the status line text to the right.
BKS_STATUSTEXTCENTER	Center the status line text.

Table 5.22: Flags controlling the alignment of the status line text in a notebook window

- (ULONG) createMinorTopPageWithStatusText: (BOOL) stFlag;

This method is used to create a new *top page* containing a minor tab for the notebook window.

`stFlag` determines whether status text shall be displayed on the top page. `-createMinorTopPageWithStatusText:` returns the notebook's page identifier of the newly created page. This identifier is used to

- set a dialog window to be displayed on the page,
- set or modify the status text of the page if display of it is enabled, and
- is used to set the text displayed on the major tab if the page has one.

- (ULONG) createLastPageWithStatusText: (BOOL) stFlag;

In contrast to `-createTopPageWithStatusText:` this method is used to create a new *last page* for the notebook control window.

`stFlag` determines whether status text shall be displayed on the new bottom page. `-createLastPageWithStatusText:` returns the notebook's page identifier of the newly created page. This identifier is used to

- set a dialog window to be displayed on the page,
- set or modify the status text of the page if display of it is enabled, and
- is used to set the text displayed on the major tab if the page has one.

- (ULONG) createMajorLastPageWithStatusText: (BOOL) stFlag;

In contrast to `-createMajorTopPageWithStatusText:` this method is used to create a new *last page* for the notebook control window. The page will have a major tab shaped according to the flags listed in Table 5.20.

`stFlag` determines whether status text shall be displayed on the new bottom page. `-createMajorLastPageWithStatusText:` returns the notebook's page identifier of the newly created page. This identifier is used to

- set a dialog window to be displayed on the page,
- set or modify the status text of the page if display of it is enabled, and
- is used to set the text displayed on the major tab if the page has one.

- (ULONG) createMinorLastPageWithStatusText: (BOOL) stFlag;

In contrast to `-createMinorTopPageWithStatusText:` this method is used to create a new *last page* for the notebook control window. The page will have a minor tab.

`stFlag` determines whether status text shall be displayed on the new bottom page. `-createMinorLastPageWithStatusText:` returns the notebook's page identifier of the newly created page. This identifier is used to

- set a dialog window to be displayed on the page,
- set or modify the status text of the page if display of it is enabled, and
- is used to set the text displayed on the major tab if the page has one.

- (ULONG) createPageBefore: (ULONG) pageID withStatusText: (BOOL) stFlag;

This method will create a new page *before* the page specified by `pageID`. The page will neither have a major nor a minor tab. The display of status line text is controlled by `stFlag`.

- (ULONG) createMajorPageBefore: (ULONG) pageID withStatusText: (BOOL) stFlag;

This method will create a new page *before* the page specified by `pageID`. The page will have a major tab. The display of status line text is controlled by `stFlag`.

- (ULONG) createMinorPageBefore: (ULONG) pageID withStatusText: (BOOL) stFlag;

This method will create a new page *before* the page specified by `pageID`. The page will have a minor tab. The display of status line text is controlled by `stFlag`.

- (ULONG) createPageAfter: (ULONG) pageID withStatusText: (BOOL) statusFlag;

This method will create a new page *after* the page specified by `pageID`. The page will neither have a major nor a minor tab. The display of status line text is controlled by `stFlag`.

- (ULONG) createMajorPageAfter: (ULONG) pageID withStatusText: (BOOL) stFlag;

This method will create a new page *after* the page specified by `pageID`. The page will have a major tab. The display of status line text is controlled by `stFlag`.

- (ULONG) createMinorPageAfter: (ULONG) pageID withStatusText: (BOOL) stFlag;

This method will create a new page *after* the page specified by `pageID`. The page will have a minor tab. The display of status line text is controlled by `stFlag`.

- deletePage: (ULONG) pageID;

Using `-deletePage:` you can delete the notebook page denoted by its identifier `pageID` from the notebook window.

A dialog window currently displayed on this page will not be freed.

- deleteAllPages;

`-deleteAllPages` will delete every single page in the notebook control.

Dialog windows currently displayed on one of the pages will not be freed.

- setStatusLineTextFor: (ULONG) pageID to: (char *) aString;

This method is used to set the status line text for page `pageID` to `aString`.

- setTabTextFor: (ULONG) pageID to: (char *) aString;

This method is used to set the text displayed on the major tab of page `pageID` to `aString`.

- setMajorTabDimensions: (USHORT) width : (USHORT) height;

`-setMajorTabDimensions::` will set the dimensions of the major tabs according to the parameters `width` and `height`. These parameters are in *window coordinates*, i.e., in pixels displayed on the screen.

Use `-sizeMajorTabs` to automatically adjust the size of the major tabs to the text displayed there.

- sizeMajorTabs;

`-sizeMajorTabs` is used to automatically adjust the size of the major tabs so that every tab's text will fit into the major tab. This method should be called after setting the tab text for every tab.

- setMinorTabDimensions: (USHORT) width : (USHORT) height;

This method is the counterpart to `-setMajorTabDimensions::` and is used to set the dimensions of the notebook's minor tabs.

- setPageButtonDimensions: (USHORT) width : (USHORT) height;

The *page buttons* normally displayed on the bottom right edge of the notebook window are used to turn the pages without using the major or minor tabs. This method is used to set the size of the page buttons to `width` and `height`.

- setPage: (ULONG) pageID to: (Window *) aWindow;

As was mentioned before, the items displayed on notebook pages are dialog windows. Use `-setPage: to:` to display the dialog window denoted by `aWindow` on the page specified by its identifier `pageID`.

- selectPage: (ULONG) pageID;

`-selectPage:` is used turn the notebook to display the page specified by its identifier `pageID`. Using this your application can select and display a specific notebook page.

5.22 PresentationSpace

Inherits from: OBJECT

Adopted Protocols: ARCHIVING

Class description:

Most of the operating system's API functions for drawing figures or bitmaps (`Gpi` functions) will use a *virtual drawing space* called the *presentation space*. This Objective C class is used to encapsulate some few of these drawing functions, mainly concerned with printint text in various sizes, in a user-friendly way.

`PresentationSpace` can be seen as some kind of abstract superclass for all classes concerned with drawing. At the moment the only class available is `Printer`, a class providing easy access to OS/2 printer objects and permit your programs to produce some printed output in a simple manner.

As the implementation of this class does not provide methods for all drawing functions supported by the OS/2 graphics programming interface (*GPI*) there is the possibility to use the `Gpi` functions for producing more complex figures. Use this classes' instance method `-hps` as the *presentation space* parameter needed by most of these functions.

The origin of the coordinate space for presentation spaces as well as for its subclasses (e.g. Printer) is in the "lower left" corner of the presentation space. Only positive coordinates along the x-axis (to the right) and the y-axis (to the top) are allowed.

Instance Variables:

HPS hps;

This variable stores a handle for the OS/2 presentation space used by instances of this class. It can be queried using `-hps` and set via `-setHps:`.

float spacing;

Some of the high-level methods provided by this class are concerned with so-called *string boxes*. These string boxes are simply strings storing more than one line of text. The lines are separated by a carriage return (use "`\n`" in your Objective C programs) and will be printed on the presentation space as a block of lines.

The lines are separated by $(spacing - 1)$textHeight pixels. For normal text you should use a spacing ratio of 1.2. This is the default value set by the various initialization methods.

LONG xResolution;

This instance variable is used to store the horizontal resolution (in dots per inch) of the currently selected font.

LONG yResolution;

This instance variable is used to store the vertical resolution (in dots per inch) of the currently selected font.

LONG textHeight;

`textHeight` stores the height in pixels of the currently selected font.

Methods:

- init;

This method initializes the object. `spacing` is set to 1.2, all other instance variables are reset to 0.

- setHPS: (HPS) aHPS;

The OS/2 presentation space which will be used for drawing is specified using `-setHPS:`. `aHPS` is the identifier of the presentation space as will be returned e.g. by `WinGetPS()`.

- (HPS) hps;

`hps` returns the handle of the currently used presentation space.

- (LONG) widthInPels;

To query the width of the presentation space, which is the width of the drawing area, either use this method or `-widthInMm`.

`-widthInPels` returns the width of the drawing area in pixels (*pels* is just a synonym for *pixel*, *display point* or *picture element*).

- (LONG) widthInMm;

To query the width of the presentation space, which is the width of the drawing area, either use this method or `-widthInPels`.

`-widthInMm` returns the width of the drawing area in millimeters.

- (LONG) heightInPels;

To query the height of the presentation space, which is the height of the drawing area, either use this method or `-heightInMm`.

`-heightInPels` returns the height of the drawing area in pixels (*pels* is just a synonym for *pixel*, *display point* or *picture element*).

- (LONG) heightInMm;

To query the height of the presentation space, which is the height of the drawing area, either use this method or `-heightInPels`.

`-heightInMm` returns the height of the drawing area in millimeters.

- (LONG) xResolution;

`-xResolution` returns the horizontal resolution of the currently selected font in *dots per inch.*

- (LONG) yResolution;

`-xResolution` returns the vertical resolution of the currently selected font in *dots per inch.*

- (LONG) textHeight;

This method returns the height of the currently selected font in *pels.*

- setSpacing: (float) ration;

`-setSpacing:` is used to set the spacing factor for string boxes to `ratio`. For normal text you should use a spacing ratio of 1.2.

- (float) spacing;

`-spacing` returns the spacing ration currently in use for string boxes.

- setFont: (char *) fontName;

This method selects a font by name. The point size currently selected is not changed.

To select the *Helvetica* system font, you could e.g. use:

```
[presentationSpace setFont: "Helvetica"];
```

- setFont: (char *) fontName at: (LONG) pointSize;

In addition to `-setFont:` this method will set the point size of the font `fontName` to `pointSize`.

To change the currently used font to *Courier* at a size of 12 points, use:

```
[presentationSpace setFont: "Courier" at: 12];
```

`pointSize` is measured in typographic points; *72pt* are approximately *1inch.*

- setFontSize: (LONG) pointSize;

To change the size of the currently used font without changing the font itself use `-setFontSize:`. This method will change the size of the currently used font to `pointSize` points.

`pointSize` is measured in typographic points; *72pt* are approximately *1inch.*

- (LONG) stringWidth: (char *) aString;

`-stringWidth:` returns the width of the string `aString` if printed using the currently selected font and size in pixels.

The string must not contain special characters as carriage returns or line-feed characters. Use `-stringBoxWidth:` to determine the width of a string box.

The height of a single string is always `textHeight` pixels.

- (LONG) stringBoxWidth: (char *) stringBox;

This method returns the width of the string box in pixels.

`stringBox` is allowed to contain line breaks (carriage return and line-feed characters). The single lines are always left-justified.

- (LONG) stringBoxHeight: (char *) stringBox;

`-stringBoxHeight:` returns the height of the string box if printed using the currently selected font and size in pixels.

- string: (char *) aString;

This method is used to print a single line of text denoted by `string` using the currently selected font and size at the cursor position in the presentation space. Use `-string: at::` to specify the position the text should be printed.

- string: (char *) aString at: (LONG) x : (LONG) y;

Draw `aString` at the position denoted by the vector (`x`, `y`) with the current font and size. `x` and `y` are measured in pixels.

- stringBox: (char *) stringBox;

Draw the string box at the current position with the current font and pointsize. The line spacing ratio can be set using `-setSpacing:`.

- stringBox: (char *) stringBox at: (LONG) x : (LONG) y;

Draw the string box starting at the coordinates specified by the vector (`x`, `y`). using the current font and pointsize. The line spacing ratio can be set using `-setSpacing:`.

The following sample source code will produce formatted output in a 12 point Helvetica font at position (200/100).

```
.
.
.
/*
 * set the default font to Helvetica at 12pt
 */
[ps setFont: "Helvetica" at: 12];

/*
 * print a string box consisting of three lines at (200/100)
 */
[ps stringBox: "string box\nsecond line\nlast line of string box"
          at: 200:100];
.
.
.
```

In the above example a presentation space object `ps` is assumed to have been initialized.

- lineTo: (LONG) x : (LONG) y;

To draw a line from the current position to the point denoted by (`x`, `y`) use `-lineTo::`.

- lineFrom: (LONG) x0 : (LONG) y0 to: (LONG) x : (LONG) y;

`-lineFrom:: to::` will draw a line from position (`x0`, `y0`) to (`x`, `y`).

5.23 Printer

Inherits from: PresentationSpace : Object

Adopted Protocols: Archiving

Class description:

The `Printer` class is provided to simplify access to the OS/2 printing capabilities. Using instances of this class will save you the troubles with the quite complicated OS/2 printing facilities.

As a subclass of `PresentationSpace` all drawing and printing methods are inherited. If you need more functionality, use the `Gpi` functions of the *OS/2 graphics programming interface* along with the presentation space `hps` created for the printer object.

Printing a single job must start with a call to `-beginSpool: appTitle:` and end with `-endSpool`.

As normally printing should not be done in the primary thread of your application and the Objective C run-time library is not thread-safe at the moment, the practical use of this class is a bit limited at the moment. Nevertheless it can be used for small printing jobs.

A simple work-around for this limitation would be to create the printer object in the primary thread and start and end spooling there, but do the work in a secondary thread. You have to take care not to use any of the function of the Objective C run-time library in the secondary thread. This also includes sending messages to objects. Despite of this you could put the handle of the presentation space as can be queried via `-hps` into a global variable and just use the `Gpi` functions for drawing from the secondary thread.

Instance Variables:

id setupDialog;

`setupDialog` is an *outlet* instance variable which should take a pointer to a dialog window displayed in case the user wishes to make changes to the current printer set up.

The dialog window *must* contain a list box control with the Presentation Manager identifier `1000` and a *Job Properties* push button having PM id `1001`. The list box is automatically initialized when the setup dialog is displayed.

Use `-setSetupDialog:` and `-setupDialog` for access to this variable and `-printerSetup:` to display the setup dialog.

Methods:

- initForApp: (StdApp *) anApp;

This method is the default initializer method for `Printer` objects. You must pass a single parameter `anApp` which is a pointer to the instance of `StdApp` in use by your application.

- free;

`-free` will terminate current print jobs and free the object. The printer setup dialog is not freed by this method.

- beginSpool: (char *) spoolTitle appTitle: (char *) appTitle;

Before you start printing, you have to initialize a printing session by calling `-beginSpool: appTitle:`. `spoolTitle` is the job title of the print job which is to be started, `appTitle` the name of the application submitting the print job.

You are not allowed to draw to the presentation space of the printer object before you initialized a print job. The job must be ended using `-endSpool`.

After using `-beginSpool: appTitle:` you are not allowed to change the printer properties using `-printerSetup:` or `-jobProperties:`.

- endSpool;

A call to `-endSpool` will end the current print job and send all data to the printer. You must call this method to close a job.

- newPage;

Calling `-newPage` will finish the current page and send the data to the printer. Afterwards a new empty printing page will automatically be created for further use in the print job.

- printerSetup: sender;

The *action method* `printerSetup:` is used to display the printer setup dialog and handle all changes the user made to the setup of the default printer. You are not allowed to call this method while a print job is running.

- setSetupDialog: aSetupDialog;

`setSetupDialog:` is used to set the outlet variable `setupDiaog` to `aSetupDialog`.

- setupDialog;

This method returns a pointer to the current printer setup dialog stored in `setupDialog`.

- jobProperties: sender;

Using this method the *Job Properties* dialog is displayed. This method is normally called when the user presses the *Job Properties* push button in the printer setup dialog.

- setDefault;

`setDefault` re-initializes the printer object and uses the system's default printer as the currently selected printer.

- (LONG) widthInPels;

To query the width of the presentation space of the printer page, which is the width of the drawing area, either use this method or `-widthInMm`.

`-widthInPels` returns the width of the drawing area in pixels (*pels* is just a synonym for *pixel*, *display point* or *picture element*).

- (LONG) widthInMm;

To query the width of the presentation space of the printer page, which is the width of the drawing area, either use this method or `-widthInPels`.

`-widthInMm` returns the width of the drawing area in millimeters.

- (LONG) heightInPels;

To query the height of the presentation space of the printer page, which is the height of the drawing area, either use this method or `-heightInMm`.

`-heightInPels` returns the height of the drawing area in pixels (*pels* is just a synonym for *pixel*, *display point* or *picture element*).

- (LONG) heightInMm;

To query the height of the presentation space of the printer page, which is the height of the drawing area, either use this method or `-heightInPels`.

`-heightInMm` returns the height of the drawing area in millimeters.

5.24 Profile

Inherits from: PRESENTATIONSPACE : OBJECT

Class description:

As you will already have noticed, OS/2 applications tend to store their configuration in configuration files having the file extension "`INI`". These files are binary files providing access to the single configuration options via a textual key.

This class was implemented to provide easy access to these configuration files. You can simply create your own "`INI`" files and store your configuration there or use the system-provided configuration files "`OS2.INI`" and "`OS2SYS.INI`".

You should normally create your own configuration files to store your application's preferences, I do not think it is good programming practice to "abuse" the system-provided configuration files. Before using one of this files make up your mind if you would like any application you start mess up your system-wide configuration files.

Every entry in a configuration file is qualified by two different keys, the first one is the called the *section name* which can be quite useful when storing configurations for multiple applications in a single file. Here you should use a unique string depending on your application's name.

The second part is the entry's *key string.* This key must be unique for the section the entry is stored in. In different sections the same key can be used for storing different configuration data.

Methods:

- init;

Initialize the object and its instance variables to default values. You should always use the initializer method `-init: forApp:` for an application-specific configuration file or `-initUserProfile` or `-initSystemProfile` for the system-wide configuration files "`OS2.INI`" and "`OS2SYS.INI`" and not this method.

- init: (char *) name forApp: (StdApp *) app;

This method is used to initialize the `Profile` object for an application-specific configuration file. The name of the configuration file is specified with the `name` parameter. `app` is a pointer to your program's instance of `StdApp`.

You can either specify the full path to your configuration file or simply specify the files name without a path. In the latter case, the file will be opened or created in the current working directory.

This method will either open an existing configuration file or create a new one if none could be found.

You have to use lower case letters for the name of the configuration file or opening/creating the file will fail.

- initUserProfile;

`-initUserProfile` will initialize the object to read and write configuration data from/to the user profile "`OS2.INI`" normally found in your systems installation directory.

- initSystemProfile;

`-initSystemProfile` will initialize the object to read and write configuration data from/to the system profile "`OS2SYS.INI`" normally found in your systems installation directory.

- free;

The `-free` method will update the configuration data and close the configuration file. Afterwards the object itself is freed.

- deleteKey: (char *) key section: (char *) section;

`-deleteKey: section:` will delete the entry specified by the *key string/section title* pair (`key/section`). Although the entry is deleted, the file is not compressed automatically. This means that deleting keys will not cause the configuration file to shrink in space.

- (void *) readData: (void *) data key: (char *) key section: (char *) section;

This method is used to read a binary configuration entry from the configuration file. The configuration entry is stored in the memory block pointed to by `data`. The entry itself is specified by `key` and `section`.

The memory block pointed to by data must be large enough for storing all the configuration data stored at the (`key/section`) pair.

This method is normally used when retrieving non-character data. The following piece of source code will read in a variable of type `long`.

```
long configdata;

[profile readData: &configdata key: "my key" section: "my app"];
```

If the entry could be found `data` is returned, `NULL` otherwise.

If you don't know how much data is stored for the specified configuration entry, use `-sizeForKey: section:` to query the data size or just read some part of the data using `-readData: size: key: section:`.

If `data` is `NULL` enough space to hold the entry is allocated via `malloc()`. It is in the programmer's responsibility to free this area later using `free()`.

- (void *) readData: (void *) data size: (long) size key: (char *) key section: (char *) section;

Just as the last method described, `-readData: size: key: section:` is used to read a binary configuration entry from the configuration file. In addition to `-readData: key: section:` an additional parameter `size` is specified which determines the maximum number of bytes the memory area pointed to by `data` can hold.

If `data` is `NULL` enough space to hold the entry is allocated via `malloc()`. It is in the programmer's responsibility to free this area later using `free()`.

- (char *) readString: (char *) data key: (char *) key section: (char *) section;

`-readString: key: section:` is used to read a configuration entry being a string from the configuration file.

`data` must either be a pointer to a memory area large enough for holding the entry or `NULL`, in which case the memory is allocated dynamically using `malloc()`.

- (char *) readString: (char *) data size: (long) size key: (char *) key section: (char *) section;

`-readString: key: section:` is used to read a configuration entry being a string from the configuration file.

`data` must either be a pointer to a memory area large enough for holding `size` bytes or `NULL`, in which case the memory is allocated dynamically using `malloc()`.

If `data` is not `NULL` at most `size` characters of the configuration entry are copied.

- (long) sizeForKey: (char *) key section: (char *) section;

This method returns the number of bytes stored for the entry specified by the (`key`/`data`) pair.

- writeData: (void *) data size: (long) size key: (char *) key section: (char *) section;

Use this method to write a block of binary data to a configuration file. The entry is stored in section `section` with the key identifier `key`. `size` bytes starting at the memory block pointed to by `data` will be stored.

Use `-readData: key: section:` or `-readData: size: key: section:` to retrieve the configuration entry later.

- writeString: (char *) data key: (char *) key section: (char *) section;

`-writeString: key: section:` will store the string `data` in the configuration file with the key pair (`key`/`section`).

Use `-readString: key: section:` or `-readString: size: key: section:` to retrieve the configuration entry later.

5.25 PushButton

Inherits from: Button : FactoryWindow : DelegateWindow : Window : Object

Class description:

The class `PushButton` is a subclass of `Button`. Its 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 5.1 on page 48. Figure 5.1 on page 48 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 `PushButton`. Using this class and method is similar to creating a Button object while specifying the flag `BS_PUSHBUTTON`.

5.26 RadioButton

Inherits from: Button : FactoryWindow : DelegateWindow : Window : Object

Class description:

The class `RadioButton` is a subclass of `Button`. Its 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 5.1 on page 48. Figure 5.1 on page 48 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`.

5.27 ScrollBar

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

If more data is to be displayed in OS/2 PM windows or in window controls than would fit inside the control, scroll bars 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;

This method is used to initialize a newly created instance of `ScrollBar`.

The PM identifier of the window is specified with `anId`, the parent window, in which the scroll bar shall be displayed is set via `parent`.

`flags` is used to specify, what kind of scroll bar shall be created. See table 5.23 on the next page for more information on the flags.

Figure 5.8 on the following page shows examples of a *horizontal scroll bar* (left) and a *vertical scroll bar* (right).

- (SHORT) position;

This method returns the current scroll bar position. This position is always in the range of [lowerBound;upperBound].

- (SHORT) lowerBound;

Return the lower bound of the scroll bar range.

Flag	Description
SBS_HORZ	This flags causes a horizontal scroll bar to be created.
SBS_VERT	Create a vertical scroll bar. 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 5.23: Flags which can be specified at scroll bar creation

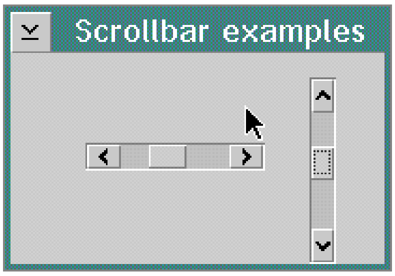

Figure 5.8: This figure shows a window containing a horizontal and a vertical scroll bar.

- (SHORT) upperBound;

Return the upper bound of the scroll bar range.

- setPosition: (SHORT) position;

`-setPosition:` sets the current position of the slider in respect to the upper and lower bounds.

- setScrollBar: (SHORT) position withBounds: (SHORT) lower : (SHORT) upper;

This method sets the slider position and the upper and lower bounds of the scroll bar. The slider position must be in the range of [lower;upper].

- setThumbSizeForVisible: (SHORT) visible of: (SHORT) all;

Using `-setThumbSizeForVisible: of:` the size of the slider is adjusted to match `visible` visible items of a total of `all` items.

5.28 Slider

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

Class description:

A Slider object is used to let the user visually choose a discrete value in a specified range.

Currently, only creation of a `Slider` object is supported.

Methods:

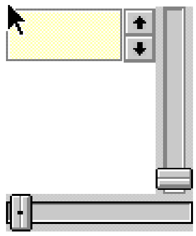

Figure 5.9: This figure shows a horizontal and a vertical slider and a spinbutton

- 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 5.9 shows a horizontal and a vertical slider control.

5.29 SpinButton

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

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 5.9 you can see an example of a spinbutton.

5.30 Static

Inherits from: FactoryWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

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.

5.31 StdApp

Inherits from: OBJECT

Adopted Protocols: ARCHIVING

Class description:

This class is used to initialize and free all necessary PM recources needed to run the application. Every Application written using this library should use exactly one instance of this class.

Instance Variables:

HAB hab;

This variable is used to store the *Handle Anchor Block* of the application. Read-only access to this instance variable is provided via `hab`.

HMQ hmq;

`hmq` stores the handle of the *Application Message Queue*. Through this message queue all application-relevant messages are passed to the designated receiver of these messages.

Because there is normally no need for the programmer to have direct access to this message queue, no methods for access to `hmq` are provided.

Methods:

- init;

This is the standard initializer of this class. `-init` creates the *Handle Anchor Block* and the *Application Message Queue*. The appropriate handles are stored in `hab` respectively `hmq`.

- free;

`-free` destroys the *Application Message Queue* and the *Anchor Block*. After calling this method, the program is ready to exit.

- run;

`-run` fetches all messages and posts them to the appropriate receivers. This method exits when a `WM_QUIT` message is received.

- (HAB) hab;

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

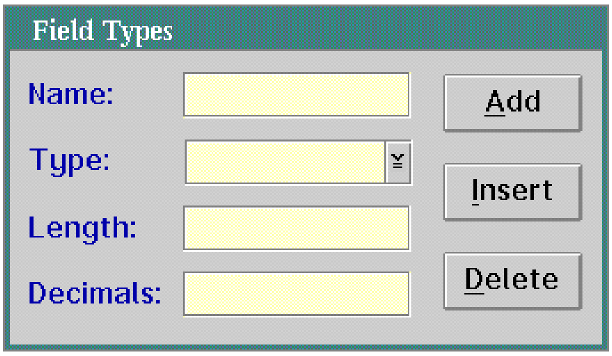

Figure 5.10: This figure shows a simple dialog window containing three *buttons*, three *entry fields* and a *drop-down combo box*.

- loadIBFile: (char *) fileName;

Using `-loadIBFile:` your program can load an interface file as created with the Interface Editor application. The objects residing in the interface file are created and all connections made between them are set up.

This method returns a pointer to the `InterfaceFile` instance created while loading the interface file. Your application is responsible for freeing this object in the end.

On failure, `nil` is returned.

The interface file's name `fileName` is first treated to be a fully qualified path name. So if no path is specified, the file will be searched for in the current working directory. If the interface file cannot be found there, the directory the application is started from—where the executable file resides—is searched.

5.32 StdDialog

Inherits from: ActionWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

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. Using `-initWithId: andFlags:` you can also create a dialog from scratch without having to create a dialog template in your application's 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 5.10 shows a simple dialog window.

Instance Variables:

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.

ULONG createFlags;

This instance variable is used to preserve the flags specified at dialog creation. The variable can be queried using `-createFlags` and modified—even if it does not seem to make much sense to do so—via `-setCreateFlags:`.

Methods:

- initWithId: (ULONG) anId;

`-initWithId:` will load 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.

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

If you want to create a dialog window "from scratch", i.e., without using a binary dialog template in a resource file, you should use this method.

Again, `anId` is the Presentation Manager identifier of the dialog window to be created. `flags` can be used to specify various frame creation flags (`FCF_xxx` flags) as described for the class `StdWindow`.

This method will return a pointer to a newly allocated and initialized dialog window.

- 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`.

- free;

`-free` destroys the PM window and frees all resources allocated previously.

- (ULONG) createFlags;

`-createFlags` wil return the value currently stored in the variable `createFlags`.

- setCreateFlags: (ULONG) flags;

To modify the instance variable `createFlags` use this method. `createFlags` is set to the parameter `flags`. Changing the creation flags will not have any impact on the apperance of the dialog window. This method is only useful when writing the dialog window to a typed stream afterwards. The changes will be visible after reloading a dialog window from a stream.

- updateFrame;

If you made any changes to the dialog's style flags by directly modifying data in the dialog's *window words*, you should call `-updateFrame` to execute these changes.

E.g. to specify a resizable border for an already existing dialog window having a dialog border, you could write

```
[dialog setWindowStyle: ([dialog windowStyle] & ~FS_DLGBORDER)
                        | FS_SIZEBORDER];
[dialog updateFrame];
```

- (ULONG) result;

`-result` returns the value stored in the instance variable `result`. This variable `result` is set after the dialog gets dismissed.

Therefore calling this method should be done only *after* the dialog has been dismissed.

- setTitle: (char *) aTitle;

This method is used to set the dialog title to `aTitle`. To ensure that the changes are displayed immediately, you should call `-updateFrame` after setting a new dialog title.

- makeKeyAndOrderFront: sender;

Calling `-makeKeyAndOrderFront:` results in the dialog becoming the active window (*key window*), where all PM messages are sent to. It is also brought to the front, if hidden by other windows, or currently invisible.

Using this method, the dialog is *not* run as a modal dialog box.

- runModalFor: sender;

`-runModalFor:` does the same as the previously described method `-makeKeyAndOrderFront:`. In addition, the dialog is run modal for the window specified by `sender`. While the dialog is run, no message processing takes place in the sending window.

`-runModalFor:` terminates, when the dialog gets dismissed.

When `sender` is `nil`, the dialog is not run modal for any window, but `runModalFor:` still doesn't terminate while the dialog is not dismissed. This can be used for applications consisting of only a single (or more) dialogs, but no `StdWindow`. In this case, don't call `[application run]`, but `[dialog runModalFor: nil]` (`application` is the current instance of a `StdApp`, `dialog` the dialog to be run instead of a `StdWindow`).

- dismiss: sender;

Calling `-dismiss:` causes the dialog—either running modal or not modal—to be dismissed. The instance variable `running` is reset to `NO` and the dialog is hidden from display.

`result` is set to `DID_CANCEL`.

You can bind this *action method* directly to a button or a menu item.

- dismissOK: sender;

Calling `-dismissOK:` causes the dialog—either running modal or not modal—to be dismissed. The instance variable `running` is reset to `NO` and the dialog is hidden from display.

`result` is set to `DID_OK`.

- (BOOL) running;

This method returns the state of the boolean instance variable `running`. After showing a dialog window using `-makeKeyAndOrderFront:` or `-runModalFor:` this variable is set to `YES`; when the dialog window gets dismissed again, it is set to `NO`.

- centerOnScreen: sender;

`-centerOnScreen:` will cause the dialog window to be centered on the screen.

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

`-handleMessage: withParams: and:` gets called by the default dialog procedure.

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, the OS/2 default dialog procedure `WinDefDlgProc()` is called.

Methods implemented by the delegate:

- windowDidMove: sender;

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

- windowDidResize: sender;

tt 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;

This method gets called, if a button posts a system command. It should react just alike `-buttonWasPressed::`.

- sysMenuWasSelected: (ULONG) menuId : sender;

`-sysMenuWasSelected::` is the counterpart to `-menuWasSelected::`, but this method only gets called, whenever a system menu item was selected.

- sysCommandPosted: (USHORT) origin : sender;

`-sysCommandPosted::` is called by the window's `-handleMessage: withParams: and:` whenever a system command was posted, and neither `-sysButtonWasPressed::` and `-sysMenuWasSelected::` return `nil`.

Its behaviour should be analogous to `-commandPosted::`.

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

Every time an event coult not be handle either by the window itself or by one of the delegate functions, `-handleMessage: withParams: and::` gets called. So all types of events can be processed without the need to subclass `StdDialog`.

The return type should always be converted explicitly to type `MRESULT`.

See also the StdDialog build in method `-handleMessage: withParams: and:`.

5.33 StdWindow

Inherits from: ActionWindow : DelegateWindow : Window : Object

Adopted Protocols: Archiving

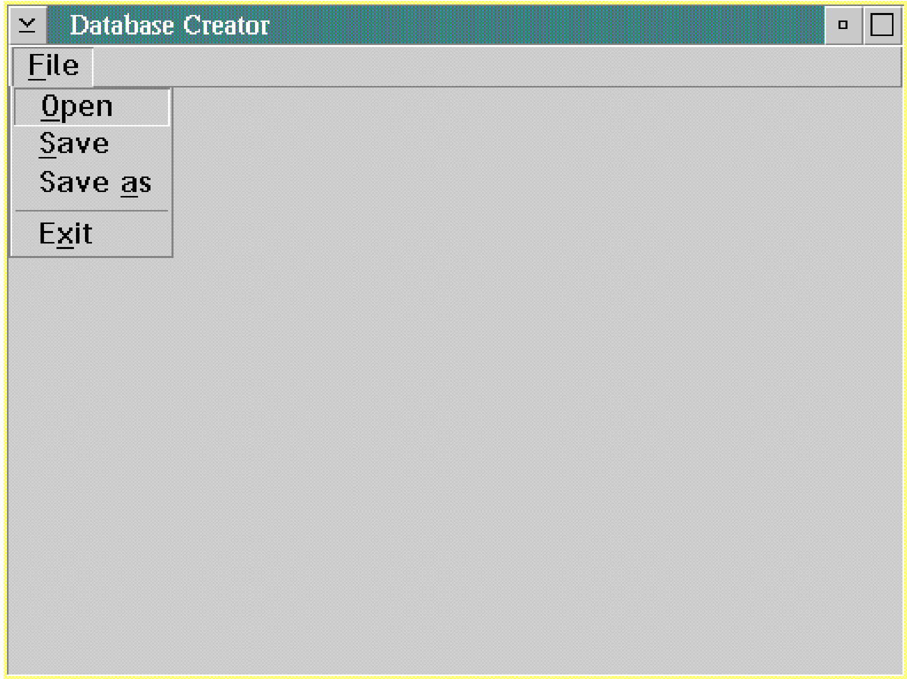

Figure 5.11: This figure shows an instance of the class `StdWindow`. At creation of the window, the flags `FCF_MENU`, `FCF_SIZEBORDER` and `FCF_ACCELTABLE` were specified

.

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 5.11 shows a `StdWindow` containing a menu bar.

For information about simplifying creation of a `StdWindow` see the class description of `Main-Window`.

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*.

ULONG createFlags;

This instance variable is used to preserve the flags specified at dialog creation. The variable can be queried using `-createFlags` and modified—even if it does not seem to make much sense to do so—via `-setCreateFlags:`.

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;

`-free` destroys the PM window and frees all resources allocated previously.

- (ULONG) createFlags;

`-createFlags` wil return the value currently stored in the variable `createFlags`.

- setCreateFlags: (ULONG) flags;

To modify the instance variable `createFlags` use this method. `createFlags` is set to the parameter `flags`. Changing the creation flags will not have any impact on the apperance of the window. This method is only useful when writing the window to a typed stream afterwards. The changes will be visible after reloading a window from a stream.

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

`-setSize::::` sets the size of the window to `(w/h)` and its position to `(x/y)`.

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

Just as with `-setSize::::` `-setRect:` is used to set the size of the window. In contrast to `-setSize::::` the position of the window is not changed.

The size of the window is set to `(w/h)`.

- updateFrame;

If you made any changes to the window's style flags by directly modifying data in the dialog's *window words*, you should call `-updateFrame` to execute these changes.

E.g. to specify a resizable border for an already existing window having a dialog border, you could write

```
[window setWindowStyle: ([window windowStyle] & ~FS_DLGBORDER)
                        | FS_SIZEBORDER];
[window updateFrame];
```

- (LONG) framexoffset;

`-framexoffset` returns the horizontal offset of the frame window relative to the lower left corner of its parent window (normally the desktop).

- (LONG) frameyoffset;

This method returns the vertical offset of the frame window relative to its parent window.

- (LONG) framewidth;

`-framewidth` returns the width of the frame window.

- (LONG) frameheight;

`-frameheigth` returns the heigth of the frame window.

- (HWND) frame;

`-frame` returns the OS/2 PM window handle of the frame window of the `StdWindow`.

- setTitle: (char *) aTitle;

Using `-setTitle:` you can set the title of the window. This title appears in the `TitleBar` of the window and also in the tasklist.

`aTitle` is the title to be set.

- makeKeyAndOrderFront: sender;

Calling `-makeKeyAndOrderFront:` results in the `StdWindow` becoming the active window (*key window*), where all PM messages are sent to. It is also brought to the front, if hidden by other windows, or currently invisible.

- centerOnScreen: sender;

`-centerOnScreen:` will cause the window to be centered on the screen.

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

`-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.

`-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;

This method gets called, if a button posts a system command. It should react just alike `-buttonWasPressed::`.

- sysMenuWasSelected: (ULONG) menuId : sender;

`-sysMenuWasSelected::` is the counterpart to `-menuWasSelected::`, but this method only gets called, whenever a system menu item was selected.

- sysCommandPosted: (USHORT) origin : sender;

`-sysCommandPosted::` is called by the window's `-handleMessage: withParams: and:` whenever a system command was posted, and neither `-sysButtonWasPressed::` and `-sysMenuWasSelected::` return `nil`.

Its behaviour should be analogous to `-commandPosted::`.

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

Every time an event coult not be handle either by the window itself or by one of the delegate functions, `-handleMessage: withParams: and::` gets called. So all types of events can be processed without the need to subclass `StdWindow`.

The return type should always be converted explicitly to type `MRESULT`.

See also the StdWindow build in method `-handleMessage: withParams: and:`.

5.34 TitleBar

Inherits from: WINDOW : OBJECT

Class description:

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

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

5.35 TriStateButton

Inherits from: BUTTON : FACTORYWINDOW : DELEGATEWINDOW : WINDOW : OBJECT

Class description:

The class `TriStateButton` is a subclass of `Button`. Its 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 5.1 on page 48. Figure 5.1 on page 48 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`.

5.36 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 its superclass *Window* has been added. Special support for OS/2 PM Valueset windows will be added in the future.

5.37 Window

Inherits from: OBJECT

Adopted Protocols: ARCHIVING

Class description:

`Window` is an abstract superclass for all classes representing some kind of window (e.g. an entry field, 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 is 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.

int tag;

This special instance variable can be used by the application programmer to store some integer data for your window objects. Instances of `Help` can make use of this *tag value* to determine which help page should be displayed.

Methods:

- init;

This method initializes the instance variables to default values, which means it sets `window` to `NULLHANDLE`. `init` returns `self`.

- associate: (HWND) hwnd;

This instance method is used to associate an already existing Presentation Manager Window (Pushbutton, ...) with an instance of the class `Window`.

The only parameter `hwnd` is the window handle of the OS/2 PM window.

By using this method the programmer can create an Objective C Object without creating a PM window. After associating a PM window with a window Object, window data can be set and queried and manipulation can be done by using instance methods.

- free;

`-free` frees all resources allocated by this object. `-free` returns `self`.

`free` does *not* destroy an associated window using the OS/2 API function `WinDestroyWindow`.

If *child windows* or *sibling windows* exist, they are freed before this window.

- destroy;

`-destroy` is used to desctroy the Presentation Manager window associated with the Objective C object.

- (int) tag;

Using `-tag` you can get the value stored in the instance variable `tag`.

- setTag: (int) aTag;

Use `-setTag:` to set the value of the instance variable `tag` to `aTag`.

- createObjects;

`-createObjects` searches if any PM child windows of this window exist, and then creates appropriate Objective C objects for each of them and inserts them in the window hierarchy of this window as child windows.

This method is maily used after loading a `StdDialog` from a resource file to build the complete object hierarchy.

- insertChild: aChild;

`-insertChild:` inserts `aChild` as a child into the window hierarchy of this window. `aChild` must be an instance of `Window` or one of its subclasses.

- insertSibling: aSibling;

`-insertSibling:` inserts `aSibling` as a child into the window hierarchy of this window. `aSibling` must be an instance of `Window` or one of its subclasses.

- deleteChild: aChild;

Using this method, `-deleteChild:`, the window will remove its child window `aChild` from its list of children. If the window could be removed, `aChild` is returned, otherwise `nil`.

- deleteSibling: aSibling;

To delete a sibling of the window, use `-deleteSibling:`. This method will return the sibling window `aSibling` if it could be removed or `nil` otherwise.

- findFromID: (ULONG) anId;

`-findFromID:` returns a pointer to an Objective C window identified by its OS/2 identifier `anId`, if there's a window identified by `anId` beyond the children of this window.

- findFromHWND: (HWND) aHwnd;

`-findFromHWND:` returns a pointer to an Objective C window identified by its OS/2 window handle `aHwnd`, if there's a window identified by `aHwnd` beyond the children of this window.

- setChild: (Window *) aChild;

To set the `Window` object's instance variable `child` to `aChild` use this instance method.

- (Window *) child;

`-child` returns the (first) child window of the `Window` object.

- setSibling: (Window *) aSibling;

To set the `Window` object's instance variable `sibling` to `aSibling` use this instance method.

- (Window *) sibling;

`-child` returns the (first) sibling window of the `Window` object.

- (char *) text: (char *) buffer;

By using `-text:` the *Window Text* of the associated PM window can be queried. If `buffer` is `NULL`, enough memory to hold the window text is allocated via `malloc` and can be freed later by the application program using `free`.

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.

- (ULONG) windowStyle;

The window's appearance depends on the window class which was used at creation and the window style which can be queried using this method. From this style, your application can e.g. determine the visibility state of the object or if any other style flags have been set during creation or later using `-setWindowStyle:`.

- setWindowStyle: (ULONG) styleFlags;

If you wish to change the window's appearance at run-time, i.e., after creating and initializing the window object, you have to use `-setWindowStyle:`. To perform an update of the window as displayed on screen, you should call `-invalidate` afterwards.

- (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.

- disable;

`-disable` disables this window. No message processing is done by this window before *re-enabling* the window by using `-enable`.

- activate;

`-activate` activates the window.

- deactivate;

`-deactivate` deactivates the window.

- invalidate;

Calling `-invalidate` causes the display area occupied by the window to be invalidated. As a consequence of this, the window is redrawn.

- show;

If the window was previously hidden (either by using the `-hide` method or by not specifying `WS_VISIBLE` at creation time, the window object is shown.

If the window is already visible, this method has no effect.

- hide;

`-hide` hides the window object. It can be made visible again using `-show`.

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

`-handleMessage: withParams: and:` gets called by the *default Window procedure* for the OS/2 PM-class `WINDOW_CLASS` if a message was sent to this window. This function only reacts to `WM_ERASEBACKGROUND`. If this message is received, `YES` is returned, otherwise the result of the default window procedure (`WinDefWindowProc()`).

The result should always be converted explicitly to the PM type `MRESULT`.

Chapter 6

Presentation Manager Library—Protocols

This chapter describes all protocols available in the Presentation Manager library. These descriptions consist of two parts,

1. The name of the protocol and a list of all classes which adopt it
2. A list of all methods declared and a short description of these

6.1 Archiving

Adopted by: ActionWindow, Button, Container, Help, InterfaceFile, ListBox, Menu, MultiLineEntryField, NoteBook, PresentationSpace, Printer, ScrollBar, Slider, SpinButton, Static, StdDialog, StdWindow, Window

Protocol description:

The Objective C run-time library provides functions for storing objects to a file and reloading these objects later. This kind of *archiving* is utilized by one of the programs shipped together with the library package.

The Interface Editor application will allow you to create objects (dialog windows, window controls, etc.) and save them in a so-called interface file. The interface file itself just stores the objects you created in a typed stream as supported by the run-time library.

In an object-oriented environment the objects itself are responsible for writing themselves to a stream or reading them in.

There are three instance method an object should implement if archiving shall be supported. Most of the classes of the Presentation Manager library do already support archiving, the rest of the classes will support this feature soon.

Methods:

- awake;

After an object has been read from a typed stream using its `-read:` method, this method is automatically called. Here you should put custom initialization for your object. `-init` is not called automatically by `-read:` and should not be called from your source code.

The object is guaranteed be initialized *after* the `-awake` method has been called. Before this methods gets called, you should not use the object.

- read: (TypedStream *) aStream;

`-read:` is used to read in an object from a typed stream. The stream is represented by `aStream`. Before unarchiving any of the instance variables you should call the inherited `-read:` method of your classes' superclass.

To read data, use the functions

- `objc_read_type()`,
- `objc_read_types()`,
- `objc_read_array()` and
- `objc_read_object()`

as provided by the Objective C run-time library.

- write: (TypedStream *) aStream;

Use `-write:` to write an object to the typed stream `aStream`. You should only archive the instance variables you cannot initialize from scratch. E.g. you should not even think of storing temporary buffers and the like.

For writing your instance variables the Objective C run-time library provides the functions

- `objc_write_type()`,
- `objc_write_types()`,
- `objc_write_object_reference()`,
- `objc_write_root_object()`,
- `objc_write_array()` and
- `objc_write_object()`.

6.2 Selection

Adopted by: EntryField, MultiLineEntryField

Protocol description:

This protocol is used to declare all OS/2 Clipboard functions which can be used by the implemented Window classes.

Methods:

- clearSelection;

`-clearSelection` clears the current Selection of items in the object which adopts this protocol.

- copySelection;

Using `-copySelection` the selected items are copied into the system clipboard. The items themselves remain unchanged.

- cutSelection;

`-cutSelection` works alike a combination of `-copySelection` and `-clearSelection`. The selected items are copied into the system clipboard and they are deleted from the source window.

- pasteSelection;

When calling `-pasteSelection` all selected Items in the system clipboard are pasted into the object implementing this method.

6.3 Value

Adopted by: EntryField

Methods:

Protocol description:

The protocol `Value` was declared to define a common interface to classes storing some data which can be accessed in various ways. At the moment, only the `EntryField` class does adopt this protocol. In addition to this, many of the instance methods defined here are provided by some other classes mainly concerned with handling of data, e.g. `String` from the utility library or the descendants of `DBField` from the database library.

The main purpose of adopting this protocol is to simplify access to an objects instance data. Some times the programmer will need access to the data in form of a string, some times a numeric representation is of greater value.

Methods:

- (char *) stringValue;

This method returns a pointer to a `NULL`-terminated string. You are not allowed to modify the data as pointed to by the return value of this method. Modifications are only allowed via the `setXXXX:` methods.

The string is only valid as long as the instance data is not modified by any of the `setXXXX:` methods.

- (int) intValue;

`-intValue` returns the instance data as an integer value.

If the exact numeric representation is a floating point number, the value returned by this method will be rounded.

- (long) longValue;

Just as `-intValue`, this method is used to query the instance data in a numeric representation. The return type of this method is `long`.

If the exact numeric representation is a floating point number, the value returned by this method will be rounded.

- (float) floatValue;

For instance data represented as a floating point value, this method should be used for read-access. `-floatValue` will return a floating point number representing the instance data of the object.

- setStringValue: (char *) aValue;

`-setStringValue:` is one of the methods used to modify the objects's instance data. In this case a pointer to a `NULL`-terminated string is passed which is then converted to the internal representation suitable for the object itself. `aValue` does not have to be a constant string. The object is responsible for *copying* the string data into an internal buffer area and not only to store the pointer `aValue`.

- setIntValue: (int) aValue;

To set the object's instance data as an integer value, use `-setIntValue:`. `aValue` is an integer number representing the instance data to be.

- setLongValue: (long) aValue;

To set the object's instance data as a long integer value, use `-setLongValue:`. `aValue` is an integer number representing the instance data to be.

- setFloatValue: (long) aValue;

As opposed to `-setStringValue:` for setting the object's instance data as a string value on the one hand and to `-setIntValue:` and `-setLongValue:` using an integer representation this method is the *write-counterpart* to `-floatValue`. It is used to set the object's instance data to the floating point number `aValue`.

Part III

Database Programming Classes

Chapter 7

Database Library—Overview

This part of the reference manual describes the classes provided by the database library. Figure 7.1 shows all classes implemented by this library.

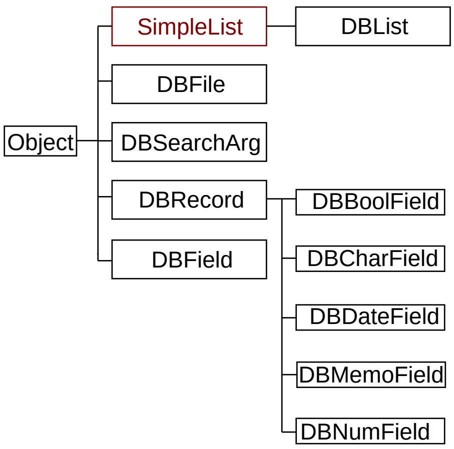

Figure 7.1: Inheritance hierarchy in Database Class library

Before using any of the classes in one of your source code files, include `<db/db.h>`. The object files must be linked with `objcdb.a` and `objcutil.a`. This can be accomplished by specifying the linker options `-lobjcdb` and `-lobjcutil` in addition to `-lobjc`.

An introduction into using this classes can be found in the tutorial.

Methods and classes not listed here should *not* be used by the application programmer.

7.1 Class "DBBoolField"

```
@interface DBBoolField : DBField
{
}

-(BOOL) booleanValue;
```

```
-setBooleanValue: (BOOL) aValue;

@end
```

7.2 Class “DBCharField”

```
@interface DBCharField : DBField
{
}

@end
```

7.3 Class “DBDateField”

```
@interface DBDateField : DBField
{
}

-(int) day;
-(int) month;
-(int) year;

-setDay: (int) aDay;
-setMonth: (int) aMonth;
-setYear: (int) aYear;

@end
```

7.4 Class “DBField”

```
@interface DBField : Object <DBProperties>
{
  id <DBEntities>  entity;
  const char      *propertyName;

  char             length,
                   decimals,
                  *stringValue;
}

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

-setStringValue: (char *) aString;
-(char *) stringValue;
-(char) length;

-(int) compareWith: (char *) aString;

@end
```

7.5 Class "DBFile"

```
@interface DBFile : Object <DBEntities>
{
   DBHEADER   *dbHeader;
   SimpleList *fieldList;

   FILE       *fileHandle;
   void       *buffer;
   long        currentRecord;

   char       *fileName;
   id          searchArguments;
}

-init: (char *) fileName;
-create: (char *) fileName withFields: (int) count list: (DBFIELD *) fields;
-free;

-field: (int) fieldNumber;
-(int) fieldCount;

-readRecord: (long) offset;
-writeRecord: (long) offset;
-(long) currentRecord;
-(BOOL) deleted;

-append;
-replace;
-delete;
-undelete;
-clear;

-(BOOL) findFirst;
-(BOOL) findNext;

-(long) recordCount;

-setSearchArguments: args;
-searchArguments;

@end
```

7.6 Class "DBList"

```
@interface DBList : SimpleList
{
  DBFile   *entity;
}

-init;
-initForEntity: (DBFile *) anEntity;

-fetchAllRecords: sender;
-fetchUsingQualifier: aQualifier;
-saveChanges: sender;
```

```
-setEntity: (DBFile *) anEntity;
-(DBFile *) entity;

@end
```

7.7 Class "DBMemoField"

```
@interface DBMemoField : DBField
{
}

@end
```

7.8 Class "DBNumField"

```
@interface DBNumField : DBField
{
}

-(int) intValue;
-(float) floatValue;
-(double) doubleValue;

-setIntValue: (int) aValue;
-setFloatValue: (float) aValue;
-setDoubleValue: (double) aValue;

@end
```

7.9 Class "DBRecord"

```
@interface DBRecord : Object
{
  DBFile   *entity;
  long      recNo;
  id        fieldList;
  BOOL      changed;
}

-initForEntity: (DBFile *) anEntity;
-free;

-replace;

-saveChanges: sender;

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

-field: (int) fieldNumber;
-(int) fieldCount;
```

```
-(long) recNo;
-copyToDB;
-copyFromDB;

-setFieldData: (unsigned long) field withString: (char *) aString;
-(char *) fieldData: (unsigned long) field;

@end
```

7.10 Class “DBSearchArg”

```
@interface DBSearchArg : Object
{
}

-initForField: aField operator: (char) anOperator
  withOperand: (char *) anOperand;
-free;

-add: anArg;
-next;

-field;
-(char) operator;
-(char *) operand;

-(BOOL) matchesCurrentRecordInFile: aFile;

@end
```

Chapter 8

Database Library—Classes

This chapter describes all variables and methods of the classes implemented in this library.

The description consists of three to six parts:

1. The name of the class and the precessing inheritance hierarchy
2. A list of protocols the class adopts. The methods defined in the *formal* protocols are not described here. See the protocols' descriptions for more information.
3. A short description of the class and its proposed usage
4. A list of all instance variables and their use
5. All *newly* implemented class and instance methods and their description
6. Methods of a *delegate object*—if such methods are supported—which get called at certain times

The list of instance variables is omitted if there are none of them defined but those inherited from the superclass.

If no return type of some method is specified, the return type defaults to `id`, a generic pointer to an *Objective C object*.

Methods returning an `id` value normally return `self`, which is a pointer to the object itself on successful completion, `nil` otherwise.

All methods having just one parameter called `sender` can be used as targets for command/action bindings. If not stated otherwise, `sender` is ignored. Normally, this parameter should be a pointer to the sending object which can be obtained by `self`.

8.1 DBBoolField

Inherits from: DBField : Object

Class description:

`DBBoolField` is a a special class for handling of fields storing boolean values.

For access to the value stored in a data base field of type *logic*, use you should use the methods `-booleanValue` and `-setBooleanValue:` in addition to the inherited access methods.

Methods:

- (BOOL) booleanValue;

If the string value stored in the database field equals "`t`" or "`T`" then this method will return `YES`. Otherwise `NO` is returned.

- setBooleanValue: (BOOL) aValue;

Depending on `aValue`, the string value of the database field is set to "`T`" in case `aValue` is `YES` or to "`F`" otherwise (`aValue` equals `NO`).

8.2 DBCharField

Inherits from: DBField : Object

Class description:

`DBCharField` is a a special class for handling of fields storing string values.

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

Use `-stringValue` and `.setStringValue:` for access to the data stored in the database field.

8.3 DBDateField

Inherits from: DBField : Object

Class description:

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

Instance methods for reading the different "parts" of a date, the *year*, the *month* and the *day* are provided as well as methods for setting this data.

Methods:

- (int) day;

`-day` will return the day of the date as stored in the database field. No checks on validity of the date are performed. Simply, the first two characters of the string value are interpreted as being the string interpretation of the day.

- (int) month;

`-month` will return the month of the date as stored in the database field. No checks on validity of the date are performed. Simply, the third and the forth character of the string value are interpreted as being the string interpretation of the month.

- (int) year;

`-year` will return the year of the date as stored in the database field. No checks on validity of the date are performed. Simply, the last four characters of the string value are interpreted as being the string interpretation of the year.

- setDay: (int) aDay;

This method is used to set the day of the date in the database field to `aDay`. The day should be in the range of $[1, 31]$ but is not checked for validity.

- setMonth: (int) aMonth;

This method is used to set the month of the date in the database field to `aMonth`. The month should be in the range of $[1, 12]$ but is not checked for validity.

- setYear: (int) aYear;

This method is used to set the year of the date in the database field to `aYear`. The year is not checked for validity.

You must not use negative values for `aYear` or values greater than `9999`. This will lead to unpredictable behaviour.

8.4 DBField

Inherits from: OBJECT

Adopted Protocols: DBPROPERTIES

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 the global data is provided using the methods of the `DBProperties` protocol.

Instance Variables:

id <DBEntities> entity;

`entity` holds a pointer to an object representing the entity (the database file or table) this field belongs to.

const char * propertyName;

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

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 * stringValue;

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 `-stringValue` and `-setStringValue:` methods. This guarantees that the internal record buffer is always kept up to date.

Methods:

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

`-initWithName: andLength: andDecimals` is used to initialize a `DBField` object. The first parameter, `aName`, should be a `NULL`-terminated string and represents the name of the database field.

`aLength` specifies the total length of the data stored in the field in bytes. `someDecimals` represents the number of decimals stored.

If `someDecimals` is greater than `0`, the total length of the numeric value stored is `aLength - 1 - someDecimals` digits before the comma, and `someDecimals` decimals.

- free;

Free the storage allocated for this object.

- setStringValue: (char *) aString;

This method is used to modify the data in the internal record buffer.

`aString` is a `NULL`-terminated string representing the data to be stored. The data is copied into the record buffer.

- (char *) stringValue;

`-stringValue` returns the data currently stored in the internal record buffer for this field as a `NULL`-terminated string.

- (char) length;

`-length` returns the total length of the database field in bytes. Simply, the value stored in the instance variable `length` is returned.

- (int) compareWith: (char *) aString;

This method was provided to compare the string stored in the database field with `aString`. The method returns an integer less than `0` if the stored string is lexically less than `aString`, `0` if the string equals `aString` and an integer greater `0` otherwise.

8.5 DBFile

Inherits from: Object

Adopted Protocols: DBEntities

Class description:

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.

Using the methods provided by the protocol `DBEntities` is preferred over using e.g. `-field:`.

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.

SimpleList * fieldList;

`fieldList` is a pointer to an instance of the `SimpleList` class provided by the utility library. In this list an instance of `DBField` or one of its subclasses is stored for every database field.

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.

char * fileName;

The name of the database file is stored in this instance variable. More exactly, a pointer to a `NULL`-terminated string is stored where the filename resides.

id searchArguments;

This variable stores a pointer to the first *search argument* which will be used when searching the database file using the methods `-findFirst` and `-findNext`.

If no search arguments should be used, i.e., every record matches the current search arguments, this variable is `nil`.

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.

- create: (char *) fileName withFields: (int) count list: (DBFIELD *) fields;

The method previously described (`-init:`) can only be used to open an existing database file. When you want to create a new database file, you have to specify all necessary header information to write a new template database.

`fileName` is the name of the database file, which shall be created. If this file already exists, it is overwritten.

`count` specifies the number of fields the database shall contain.

The information what fields shall be stored is passed in the `fields` parameter. `fields` is an array of structures of type `DBFIELD`. This type is defined in `<db/dbtypes.h>`.

You must fill in the following fields

- `name` ··· must be a `NULL`-terminated string of uppercase letters (for compatibility reasons) with at most 11 characters (including the terminating `NULL` character). This is the name of the field.

  Use `toupper()` to convert the name to uppercase letter.
- `type` ··· specifies the type of data stored in this field. This can be `DB_FLD_CHAR` for character fields (strings), `DB_FLD_NUM` for numeric fields (with and without decimals), `DB_FLD_LOGIC` for fields storing logic values (boolean values), or `DB_FLD_DATE` for fields storing dates.

  `DB_FLD_MEMO` should *not* be used because handling of memo fields is not implemented now.
- `data_ptr` ··· is the pointer to the data for this field in the record buffer. This is calculated at creation of the database. You must initialize this variable to `NULL`.
- `length` ··· is the length of the field data in bytes. The datatype used for this variable is `unsigned char`.

  Remember to count all characters to be stored. When using numeric data with decimals, `length` is `decimals + 1 +` *number of digits before comma.*

  When using a date field, `length` *must* be `8`.

  Character data is stored without a terminating `NULL` character.
- `dec_point` ··· is an `unsigned int` designed to hold the number of decimals for numeric fields. For all other field types this must be set to `0`.

So creating a simple database with two fields, where the first is designed to store a string with at most 10 characters and a numeric field with 5 digits before and 2 after the comma would look like this:

```
.
.
DBFile  *newDatabase = [DBFile alloc];
DBFIELD  fieldinfo[2];
```

```
/* set information for first field */
strcpy (fieldinfo[0].name,"FIELD 1");
fieldinfo[0].type = DB_FLD_CHAR;
fieldinfo[0].data_ptr = NULL;
fieldinfo[0].length = 10;
fieldinfo[0].dec_point = 0;

/* set information for second field */
strcpy (fieldinfo[1].name,"FIELD 2");
fieldinfo[1].type = DB_FLD_NUM;
fieldinfo[1].data_ptr = NULL;
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.

Using the methods to search for a specific field as provided by the protocol `DBEntities` is preferred. This method is subject to change.

- (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;

Write the record in the internal record buffer to the database file. A new record is appended.

This method is equal to using `[database writeRecord: [database recordCount]]`.

- replace;

Replace the current record in the database file with the data in the internal record buffer.

This is equal to using `[database writeRecord: [database currentRecord]]`.

- delete;

Mark the current record as deleted. The information is *not* written to the database automatically. The changes are automatically saved to disk by calling `-replace`.

- undelete;

Mark the current record as not deleted. The information is *not* written to the database automatically. The changes are automatically saved to disk using `-replace`.

- clear;

`-clear` clears the record buffer. All values are reset to their defaults. You should always call this method before setting up the record buffer to append a new record.

- (BOOL) findFirst;

To access all records in the database, you should use `findFirst` and `findNext`.

`-findFirst` searches for the first not deleted record in the database matching the current *search arguments* and returns `YES` on success.

If no active record could be found, `NO` is returned.

Visiting all records in a database can be accomplished using the following piece of source code

```
.
.
if ([database findFirst])
  do {
    .
    . /* do modifications to records */
    .
  } while (![database findNext]);
.
.
```

Here, `database` is assumed to be a pointer to a successfully initialized `DBFile` object.

- (BOOL) findNext;

`-findNext` searches for the next record in the database file which is not marked as deleted and matches the search arguments.

The search *must* have been initially started using `findFirst`.

Because at the moment indexing is not supported exhaustive searches can cause a severe delay in your application's response time.

- (long) recordCount;

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

- setSearchArguments: args;

`-setSearchArguments:` is used to set the search arguments to `args`. The search arguments in effect up to now are removed and the first of them is returned by this method.

To clear the search arguments, simply pass `nil` in the `args` parameter.

Search arguments should be instances of `DBSearchArg`.

- searchArguments;

`-searchArguments` will return a pointer to the first search argument currently in use with the database file. If no search arguments have been set previously, `nil` is returned.

8.6 DBList

Inherits from: SimpleList

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.

As the class is derived from `SimpleList` access to the single records which are stored as Objective C objects is provided using the familiar methods (see Section 3 on Page 16).

Instance Variables:

DBFile * entity;

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

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 `-setEntity:`.

- initForEntity: (DBFile *) anEntity;

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 `anEntity`.

- fetchAllRecords: sender;

Using the entity' s `-findFirst` and `-findNext` methods all active records are retrieved into this object.

This method is an *action method* and can be directly bound to a command sent by a button or a menu item.

- fetchUsingQualifier: aQualifier;

In addition to `-fetchAllRecords:` this method allows a so-called *qualifier* to be specified for the fetch. This qualifier is merely an instance of `DBSearchArg` used to filter the records and only fetch those matching the criteria as defined in the search arguments.

This method is *no* action method! `aQualifier` is a pointer to an instance of `DBSearchArg` which should be the first search argument to use.

- setEntity: (DBFile *) anEntity;

Associate the database object `anEntity` with the record list.

- (DBFile *) entity;

This returns the associated instance of `DBFile`.

8.7 DBMemoField

Inherits from: DBField : Object

Class description:

`DBMemoField` is a a special class for handling of fields storing multiple lines of text.

Memo fields are currently not supported.

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

8.8 DBNumField

Inherits from: DBField : Object

Class description:

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

Methods for reading and writing numerical data in various formats are provided.

The access methods themselves are a subset of the ones described in the protocol `Value`. Only the access functions used for numerical data are implemented.

Methods:

- (int) intValue;

`-intValue` returns the value stored in the database field as an integer value.

If the exact numeric representation is a floating point number, the value returned by this method will be rounded.

- (float) floatValue;

For field data represented as a floating point value, this method should be used for read-access. `-floatValue` will return a floating point number representing the instance data of the object.

If you need a higher-precision floating point number, use `-doubleValue` and `-setDoubleValue:` for access to the database field.

- (double) doubleValue;

Just as `-floatValue`, this method is used to query the instance data in a numeric representation. The return type of this method is `double`.

- setIntValue: (int) aValue;

To set the object's field data as an integer value, use `-setIntValue:`. `aValue` is an integer number representing the instance data to be.

- setFloatValue: (long) aValue;

As opposed to `-setStringValue:` for setting the object's instance data as a string value on the one hand and to `-setIntValue:` and `-setLongValue:` using an integer representation this method is the *write-counterpart* to `-floatValue`. It is used to set the object's instance data to the floating point number `aValue`.

- setDoubleValue: (double) aValue;

To set the object's instance data as a double precision floating point number, use the classes' instance method `-setDoubleValue:`. `aValue` is a double precision floating point number representing the new value stored in the database field.

8.9 DBRecord

Inherits from: Object

Class description:

The previously decribed class `DBList` is used to store many records in memory at once. For every single record an instance of `DBRecord` is allocated and initialized. These objects are used as the the internal storage area of the records and to modify the data stored there.

To simplify usage of a `DBList` object and the associated instances of `DBRecord` in a container object (detail's view) the methods `-setFieldData: withString:` and `-fieldData:` are provided. As a consequence of this, you can simply insert `DBRecord` objects into a `Container` object and the single fields will be displayed automatically in the container's detail's view.

Instance Variables:

DBFile * entity;

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.

id fieldList;

For every single record, the field list as defined for the instance of `DBFile` is copied to its private storage area. Access to the fields is performed using `-field:`. The data can be modified directly using the methods `-setFieldData: withString:` and `-fieldData:`.

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:

- initForEntity: (DBFile *) anEntity;

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

- free;

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

- replace;

This method copies the record buffer to the database object and replaces the record associated with this object on disk.

- setChanged: (BOOL) value;

Using `-setChanged:` you can modify the value of the instance variable `changed` by hand. `changed` is set to `value`.

- (BOOL) changed;

The method `-changed` returns the value stored in the instance variable `changed`.

Normally, `YES` is returned, if the record got modified, otherwise `NO` is returned.

- field: (int) fieldNumber;

This method will return a pointer to the instance of `DBField` or one of its subclasses associated with the database field `fieldNumber`. Reading and modifying the data stored in the field object can be done using the appropriate object's instance methods, e.g. `-stringValue` or `-setStringValue:`.

- (ing) fieldCount;

`-fieldCount` returns the number of fields the record currently stores.

- (long) recNo;

`-recNo` returns the number of the record in the database file, which is associated with this object.

- copyToDB;

`-copyToDB` copies the record buffer of this object to the internal record buffer of the database object.

The number of the record in the database file is *not* set. So don't try saving the data using `[database replace]`.

- copyFromDB;

`-copyFromDB` retrieves the data stored in the internal record buffer of the database object to the record buffer of this object.

- setFieldData: (unsigned long) field withString: (char *) aString;

This method was only implemented for convenience reasons. It is automatically called when a `DBRecord` object in a container control gets modified by the user. You should normally not use the method by yourself.

The field number is specified by `field`, `aString` is the new string value the field will be set to.

- (char *) fieldData: (unsigned long) field;

This method was only implemented for convenience reasons. It is automatically called when a `DBRecord` object in a container control is displayed in detail's view. You should normally not use the method by yourself.

`field` specified the field number. The string value of the record's field is returned.

8.10 DBSearchArg

Inherits from:

Class description:

`DBFile` itself as well as `DBList` do support fetching records matching some application-defined criteria. These criteria—or search arguments—are specified by using instances of the class `DBSearchArg`.

Search arguments are organized in a linked list. The first search argument must be passed to the appropriate `DBFile` or `DBList` methods to be in effect.

Freeing the first search argument will cause all other objects in the list to be freed, too.

Single search arguments will always work on a single field of a record. This field is compared using an *operator* (see Table 8.1) with an *operand* given as a `NULL`-terminated string.

The constants shown in Table 8.1 are defined in `<util/DBSearchArg.h>`.

The single search arguments in the list will form a single query determining if the current record matches the search criteria or if it does not. A record will only pass the comparison if it passes *every* single comparison. If only one search argument evalueates to *false* the whole comparison will evaluate to *false*.

Methods:

- initForField: aField operator: (char) anOperator withOperand: (char *) anOperand;

This initializer method initializes the search argument object. It is automatically assumed to be the last search argument in the linked list.

`aField` is a pointer to a `DBField` object representing the first operand. This object must not be taken from a `DBRecord` object but from the entity instance `DBFile` itself. `anOperator` is one of the constants described in Table 8.1 defining the operator to be used for the comparison. `anOperand` is the second operand. It is a `NULL`-terminated string value which is automatically

Operand	**Description**
CMP_EQUAL	The comparison evaluates to *true*, if both operands (the string value stored in the specified field of the record and the `NULL`-terminated string specified as the second operator) are equal. Otherwise the comparison results in *false*.
CMP_NOTEQUAL	*false* is returned if both operands are equal, *true* otherwise.
CMP_LESS	If the first operand—the one stored in the database field—is (lexicographically) less than the second operand, the comparison results in *true*, otherwise in *false*.
CMP_GREATER	If the first operand—the one stored in the database field—is (lexicographically) greater than the second operand, the comparison results in *true*, otherwise in *false*.
CMP_LESSEQUAL	If the first operand—the one stored in the database field—is (lexicographically) less than or equal to the second operand, the comparison results in *true*, otherwise in *false*.
CMP_GREATEREQUAL	If the first operand—the one stored in the database field—is (lexicographically) greater than or equals the second operand, the comparison results in *true*, otherwise in *false*.

Table 8.1: Constants usable as operators for search arguments and a short description of each of them.

cut if it is longer than the maximum length a string value in the field specified by `aField` is allowed to be.

To illustrate the use of search arguments together with a `DBList` a simple piece of source code will be shown:

```
.
.
.
/*
 * declare and initialize the variables
 */
DBFile *dbfile = [[DBFile alloc] init: "test.dbf"];
DBList *dblist = [[DBList alloc] initForEntity: dbfile];
id      searchArg;
.
.
/*
 * now initialize the search arguments.
 * only the records having the string value "r1" in field "comment"
 * and having a value greater than or equal to "P" and less than
 * "Q" in field "name" shall match the criteria.
 */

/* first search argument: field "comment" == "r1" */
searchArg = [[DBSearchArg alloc] initForField: [dbfile propertyNamed: "comment"]
                                     operator: CMP_EQUAL
                                  withOperand: "r1"];


/* add second search argument: field "name" >= "P" to the list */
[searchArg add: [[DBSearchArg alloc] initForField: [dbfile propertyNamed: "name"]
                                         operator: CMP_GREATEREQUAL
                                      withOperand: "P"]];

/* add the third and last search argument: field "name" < "Q" to the list */
[searchArg add: [[DBSearchArg alloc] initForField: [dbfile propertyNamed: "name"]
                                         operator: CMP_LESS
                                      withOperand: "Q"]];

/*
 * now fetch the records into the DBList instance
 */
[dblist fetchUsingQualifier: searchArg];

/*
 * and delete the search arguments
 */
[searchArg free];
.
.
.
```

As the single search arguments are connected by a logical *and*, this query can be read as:

> Fetch all records from the database `dbfile` into the record list `dblist` where the value stored in the field "`comment`" equals "`r1`" and whose "`name`" field starts with the letter "`P`".

As indexing is not supported at the moment, queries will always have to search the whole data file. This can be very time-consuming if your application will make queries to a large data file frequently.

- free;

`-free` will free the whole list of search arguments starting with this object.

- add: anArg;

This method adds `anArg` to the end of the list of search arguments starting with this object.

- next;

`-next` returns the next search argument in the list. If this one is the last argument, `nil` is returned.

- field;

Using `-field` a pointer to the field object representing the first operator of the search argument is returned.

- (char) operator;

This method returns the operator used for the search argument.

- (char *) operand;

`-operand` returns the second operand of the search argument.

- (BOOL) matchesCurrentRecordInFile: aFile;

This method is called by the `DBFile` or `DBList` instance used for fetching the records for every single record. It has to check the current record and returns `YES` if it matches the search arguments or `NO` if it does not do so.

Chapter 9

Database Library—Protocols

This chapter describes all protocols available in the Database library. These descriptions consist of two parts,

1. The name of the protocol and a list of all classes which adopt it
2. A list of all methods declared and a short description of these

9.1 DBEntities

Adopted by: DBFile

Protocol description:

The `DBEntities` protocol defines various methods useful with all classes directly providing access to a database file or a table in a database.

At the moment only `DBFile` adopts this protocol, in future versions I plan to support various database systems, e.g. SQL servers, for database access. Then some more classes will be provided conforming to this protocol and providing the same consistent interface to your application program.

Methods:

- database;

In case the entity object does represent a single table or relation from a database, this method will return a pointer to the database object managing the entity.

In case there is no database class managing access to the table this method just returns `nil`.

- getProperties: (SimpleList *) aList;

`-getProperties:` will fill the list object `aList` with all property objects the entity consists of. In case of `DBFile`, these property objects will be instances of `DBField` or one of its subclasses.

- (BOOL) matchesEntity: (id <DBEntities>) anEntity;

The instance method `-matchesEntity:` will check if the entity object manages the same data (database file, table or relation) than `anEntity`. In case this is true, `YES` is returned, `NO` otherwise.

- (const char *) entityName;

`-entityName` returns the name of the entity as a `NULL`-terminated string. You are not allowed to modify the string you receive by calling this method.

- propertyNamed: (const char *) aName;

The method `-propertyNamed:` will search for a property having the name `aName` in the list of properties the entity manages. The property object will be returned in case it was found, `nil` otherwise.

Depending on the entity object `aName` is a case-sensitive or case-insensitive `NULL`-terminated string. For `DBFile` objects, the names of the properties are treated all as upper-case letters.

For compatibility reasons, you should use this method to retrieve a pointer to a specific database field instead of the `-field:` method of `DBFile`.

9.2 DBProperties

Adopted by: DBField

Protocol description:

To provide a general interface for every property object (e.g. instances of the class `DBField`) the `DBProperties` protocol was defined.

In this version of the database library many of the methods will just return default values. These methods are:

- `-isKey` will always return `NO`.
- `-isReadOnly` will always return `NO`.
- `-isSingular` will always return `YES`.
- `-propertyType` will return `nil`.
- `-setName:` will not change the name of the property at the moment and always return `NO`.

The protocol and the methods were designed for use with some more complex classes for managing databases as will provided in one of the next versions of this library package.

Methods:

- (id <DBEntities> entity;

`-entity` returns the entity object associated with the property object.

- (BOOL) isKey;

If the property is a key property or part of a key property, this method will return `YES`. Otherwsie `NO` is returned.

- (BOOL) isReadOnly;

Some properties will be of a read-only type. In this case, `-isReadOnly` will return `YES`, `NO` otherwise.

- (BOOL) isSingular;

If the property object represents a single field in a database file or table/relation, this method will return `YES`. In case the property is a compound property, i.e., its contents are computed from some other fields in the entity, `NO` is returned.

- (BOOL) matchesProperty: (id <DBProperties>) aProperty;

The instance method `-matchesProperty:` will check, if `aProperty` and the property object itself do manage the same database field or column in a database table or relation. If this is the case, `YES` is returned, otherwise this method results in `NO`.

- (const char *) propertyName;

`-propertyName` returns the name of the property as a constant `NULL`-terminated string. Depending on the implementation of the used database engine (server or library class) the name is converted to lower-case or upper-case letters only.

- propertyType;

This method is reserved to check the type of a property. At the moment this method is not implemented in any of the classes adopting this protocol. It will always return `nil` by now.

- (BOOL) setName: (const char *) aName;

To eventually change the name of the property, you can use this method. It will set the name of the property to `aName` if this is possible and will return a boolean value determining if the name could be changed or not.

In case of using the internal library classes `DBFile` and `DBField` for access to a database, changing a property's name is not possible and this method will always return `NO`.

Bibliography

[1] *Introduction to PM Programming.* Salomon, Larry Jr. Article series starting in EDM/2 Vol. 1, issue 7.

[2] *OS/2 Version 2.0, Volume 4: Application Development.* IBM International Technical Support Center, Boca Raton. IBM Document Number GG24-3774-00.

[3] *Programming the Container Control.* Salomon, Larry Jr. Article series in EDM/2 Vol. 1, issues 3–5.

Index