iMops 2.x Class Reference (Uncompleted)

N.S.

Nov. 2016

Contents

Introduction

This is a class reference for iMops, which is a Cocoa/x86_64 port of PowerMops.

Chapter 1

Basic Data Structures

Introduction

iMops' basic data structure classes are mostly replications of the latest PowerMops' ones. However, handle-based classes were dropped because handle is no longer an efficient way to use heap memory in Mac OS X. String class is now based on a pointer to a heap block allocated through `malloc()` system call. `Object_list` class has been introduced from PowerMops 6.2 and considered as a replacement of `HandleList` class. The names of methods of these two classes are mutually similar, but `Object_list` uses the reference system, instead of handle object. The reference system is linked to the garbage collector for heap objects. Removed item object of `Object_list` will be sent to `Garbage_Pool`. At the next idle time `release:` message will be sent to the object and `Garbage_Pool` will be drained.

1.1 Elementary Data Storage

1.1.1 Object

`Object` contains behavior appropriate to all objects in the system. Every superclass chain ultimately traces back to `Object`.

Superclass	`Meta`
Source File	class-base
Status	Core
Instance variables	None
Indexed data	None
System objects	None

Methods

`class:`	`( -- token )`	Returns the token of the object's class.
`.id:`	`( -- )`	Types the object's name.
`.class:`	`( -- )`	Types the name of the object's class.
`.super:`	`( -- )`	Types the name(s) of the object's superclass(es).
`addr:`	`( -- addr )`	Returns the base address of an object's data.
`length:`	`(-- #bytes )`	Returns the length of the object's ivar data area
`copyto:`	`( ^obj -- )`	`^obj` is a pointer to another object. This method copies that object's ivar data to this object. Be careful using this method — no check is done that the objects are of the same class. However this method can be very useful in some situations.
`classinit:`	`( -- )`	This is a very special method — where an object is created Mops sends it a `classinit:` message so that it will initialize itself to reasonable values, or whatever the programmer desires all objects of that class to do when created. This method corresponds to a constructor method in `C++`. In class `Object`, it is do-nothing method, allowing any subclass to override it as appropriate. By convention, `init:` is used for explicit programatic initialization and customization thereafter, and `new:` is used to set up the framework-interface portion of Objective-C objects (such as getting a window created from Cocoa framework.)
`release:`	`( -- )`	This method does nothing in Object itself. However, in general you should send `release:` to an object before you FORGET it or deallocate its memory. `release:` will cause an object to release any heap memory it has allocated and do any other cleaning up which may be necessary. This method corresponds to a destructor method in `C++`.
`dump:`	`( -- )`	Dumps the dictionary entry for the objet in a hex format.
`print:`	`( -- )`	Dumps the dictionary entry for the object in a hex format. This provides a default `print:` method for objects that don't have a more sophisticated form or displaying their data.

persistence/serialization

`send:`	`( ^obj -- )`	Sends a `write:` message to the passed-in object, to write out this object's data as as stream of bytes.
`bring:`	`( ^obj -- )`	Sends a `read:` messsage to the passed-in object, to read in this object's data as a stream of bytes.

Error messages

None

1.1.2 ZVar

`ZVar` provides storage for 64-bit numeric quantities.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`8 bytes __1cell`
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- val )`	Returns the 8byte value in the data area.
`put:`	`( val -- )`	Stores a new value in the data area.
`+:`	`( val -- )`	Adds the `val` to the contents of the data area.
`-:`	`( val -- )`	Subtracts the `val` from the contents of the data area.
`->:`	`( ^ZVar -- )`	Copies the passed-in `ZVar`'s data to this `ZVar`
`neg:`	`( -- )`	`NEGATE`s the contents value in the data area.
`not:`	`( -- )`	Operates bit-wise `NOT` to the contents of the data area.
`and:`	`( val -- )`	Operates bit-wise `AND` with the `val` to the contents of the data area.
`or:`	`( val -- )`	Operates bit-wise `OR` with the `val` to the contents of the data area.
`xor:`	`( val -- )`	Operates bit-wise `XOR` with the `val` to the contents of the data area.
`clear:`	`( -- )`	Puts `0` to the data area.

initialization

`classinit:`	`( -- )`	`clear:` the contents.

display

`print:`	`( -- )`	Prints the data in the current number base on the screen.

Error messages

None

1.1.3 Var

`Var` provides storage for 32-bit signed numeric quantities.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`4 bytes __halfcell`
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- val )`	Returns the 4byte signed value in the data area.
`put:`	`( val -- )`	Stores a new value in the data area.
`+:`	`( val -- )`	Adds the `val` to the contents of the data area.
`-:`	`( val -- )`	Subtracts the `val` from the contents of the data area.
`->:`	`( ^var -- )`	Copies the passed-in `Var`'s data to this `VAR`
`neg:`	`( -- )`	`NEGATE`s the contents value in the data area.
`not:`	`( -- )`	Operates bit-wise `NOT` to the contents of the data area.
`and:`	`( val -- )`	Operates bit-wise `AND` with the `val` to the contents of the data area.
`or:`	`( val -- )`	Operates bit-wise `OR` with the `val` to the contents of the data area.
`xor:`	`( val -- )`	Operates bit-wise `XOR` with the `val` to the contents of the data area.
`clear:`	`( -- )`	Puts `0` to the data area.

initialization

`classinit:`	`( -- )`	`clear:` the contents.

display

`print:`	`( -- )`	Prints the data in the current number base on the screen.

Error messages

None

1.1.4 UVar

`UVar` provides storage for 32-bit unsigned numeric quantities.

Superclass	`Var` (1.1.3)
Source file	class-base
Status	Core
Instance variables	None (see `Var`)
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- uval )`	Returns 4byte value in the data area as an unsigned number.

Error messages

None

1.1.5 Int

`Int` provides storage for 16-bit signed numeric quantities.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`2 bytes __2bytes`
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- val )`	Returns the 2byte signed value in the data area.(-32768 – 32767)
`put:`	`( val -- )`	Stores a new value in the data area.
`+:`	`( val -- )`	Adds the `val` to the contents of the data area.
`-:`	`( val -- )`	Subtracts the `val` from the contents of the data area.
`->:`	`( ^INT -- )`	Copies the passed-in `Int`'s data to this `INT`
`neg:`	`( -- )`	`NEGATE`s the contents value in the data area.
`not:`	`( -- )`	Operates bit-wise `NOT` to the contents of the data area.
`and:`	`( val -- )`	Operates bit-wise `AND` with the `val` to the contents of the data area.
`or:`	`( val -- )`	Operates bit-wise `OR` with the `val` to the contents of the data area.
`xor:`	`( val -- )`	Operates bit-wise `XOR` with the `val` to the contents of the data area.
`clear:`	`( -- )`	Puts `0` to the data area.

initialization

`classinit:`	`( -- )`	`clear:` the contents.

display

`print:`	`( -- )`	Prints the data in the current number base on the screen.

Error messages

None

1.1.6 UInt

`UInt` provides storage for 16-bit unsigned numeric quantities.

Superclass	`Int` (1.1.5)
Source file	class-base
Status	Core
Instance variables	None (see `INT`)
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- uval )`	Returns 2byte value in the data area as an unsigned number. (0 – 65536)

Error messages

None

1.1.7 Byte

`Byte` provides storage for 8-bit signed numeric quantities.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`1 bytes __1byte`
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- val )`	Returns the 1byte signed value in the data area (-128 – 127).
`put:`	`( val -- )`	Stores a new value in the data area.
`+:`	`( val -- )`	Adds the `val` to the contents of the data area.
`-:`	`( val -- )`	Subtracts the `val` from the contents of the data area.
`->:`	`( ^BYTE -- )`	Copies the passed-in `Byte`'s data to this `BYTE`
`neg:`	`( -- )`	`NEGATE`s the contents value in the data area.
`not:`	`( -- )`	Operates bit-wise `NOT` to the contents of the data area.
`and:`	`( val -- )`	Operates bit-wise `AND` with the `val` to the contents of the data area.
`or:`	`( val -- )`	Operates bit-wise `OR` with the `val` to the contents of the data area.
`xor:`	`( val -- )`	Operates bit-wise `XOR` with the `val` to the contents of the data area.
`clear:`	`( -- )`	Puts `0` to the data area.

initialization

`classinit:`	`( -- )`	`clear:` the contents.

display

`print:`	`( -- )`	Prints the data in the current number base on the screen.

Error messages

None

1.1.8 UByte

`UByte` provides storage for 8-bit unsigned numeric quantities.

Superclass	`Byte` (1.1.7)
Source file	class-base
Status	Core
Instance variables	None (see `Byte`)
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- uval )`	Returns 1byte value in the data area as an unsigned number. (0 – 255)

Error messages

None

1.1.9 Bool

`Bool` provides storage for truth values.

Superclass	`Byte` (1.1.7)
Source file	class-base
Status	Core
Instance variables	None (see `Byte`)
Indexed data	None
System object	None

Methods

accessing

`set:`	`( -- )`	Sets the value in the data area TRUE.

display

`print:`	`( -- )`	Prints either "true" or "false" according to the value in the data area.

Error messages

None

1.1.10 Ptr

`Ptr` adds to `ZVar` methods that may be useful for manipulations of a non-relocatable block of heap. Memory management is wholly done by OS since Mac OS X. So, now, we need not care for memory fragmentation problems. By this change, Pointer (`Ptr`) has become the default way to manipulate heap memory area. So, in iMops, `Handle` related classes were gone.

`Nil`(= 0) is never valid address in Mach, so iMops uses `0`, instead of `nilP`, to represent invalid pointer.

Superclass	`ZVar` (1.1.2)
Source file	class-base
Status	Core
Instance variables	None (see `ZVar`)
Indexed data	None
System object	None

Methods

accessing

`ptr:`	`( -- addr )`	Returns the pointer stored in the data area.

manipulation

`new:`	`( len -- )`	Allocates `len` bytes area on heap and store the address in the data area.
`release:`	`( -- )`	Releases `new:`ed heap area and clear the data area.
`setSize:`	`( len -- )`	Sets the allocated heap memory size to `len` and copies previously stored data in heap memory up to the new size. If not allocated yet, same as `new:`.
`nil?:`	`( -- b )`	Returns true or false according to whether the content pointer is `nil`.

Error messages

None

1.1.11 DicAddr

`DicAddr` is used for storing the address of a location within the dictionary. If the dictionary is saved and reloaded in a subsequent run, the address will still valid. This is accomplished by storing the address in a relocatable format. Don't depend on the details of this format, in case it changes.

Superclass	`Var` (1.1.3)
Source file	class-base
Status	Core
Instance variables	`Bool in-code-dic?`
Indexed data	None
System object	None

Methods

accessing

`get:`	`( -- addr )`	Overrides `get:` in `UVAR`. Fetch the object's data (a relocatable address), converts it to absolute and returns the result.
`put:`	`( addr -- )`	Stores the passed-in absolute address in the object's data, using our relocatable format.

display

`print:`	`( -- )`	Types the word name corresponding to the stored address, or "(no name)" if the code dictionary address isn't the address of a Mops word, or "In data dictionary" if the address is in the data dictionary.

Error messages

"Not in the dictionary!"
You attempted to `put:` an address of the location out of the dictionary.

1.1.12 X-Addr

`X-Addr` provides a storage for `xt` of a word. In contrast to Mops or PowerMops, `xt` is not the address of the word in iMops and `xt` itself is already representing the word in a relocatable format. So, `X-Addr` is less similar to `DicAddr` than to `VAR` in iMops. The difference from `VAR` is that `X-Addr` checks whether the passed-in data is really an `xt` on `put:` and has `exec:` method to execute the stored `xt`.

Superclass	`Var` (1.1.3)
Source file	class-base
Status	Core
Instance variables	None (see `Var`)
Indexed data	None
System object	None

Methods

accessing

`put:`	`( xt -- )`	Stores the `xt` in the object's data area after checking whether it is really an `xt`.
`exec:`	`( ? -- ? )`	Executes the word whose `xt` has been stored in the object.

Error messages

"Not a xt!"
You attempted to `put:` an value that is not an `xt`.

1.2 Array, List and Collection

1.2.1 Indexed-Obj

This class is the generic superclass for all standard arrays. It defines the general indexed methods, which apply regardless of indexed width.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	None
Indexed data	None (supplied by subclasses)
System object	None

Methods

accessing

`^elem:`	`( index -- addr )`	Returns the address for the element at `index`.
`limit:`	`( -- maxIndex+1 )`	Returns the allocated size of an indexed object. the maximun usable index for an indexed object is this value minus 1.
`width:`	`( -- #bytes )`	Returns the width of each indexed element.
`ixAddr:`	`( -- addr )`	Returns the address of the 0th element.

manipulation

`clearX:`	`( -- )`	Sets each element to 0.

Error messages

"The index is Out of Range!"
One of the methods taking an index found the index value to be out of range of this array.

1.2.2 Basic array classes – bArray, wArray, Array

These basic array methods are implemented for the three array classes in iMops.

Superclass	`Indexed-Obj` (1.2.1)
Source file	class-base
Status	Core
Instance variables	None
Indexed data	1, 2, 4-byte cells
System object	None

Methods

accessing

`at:`	`( index -- val )`	Returns the data at a given indexed cell.
`to:`	`( val index -- )`	Stores data at a given indexed cell.
`+to:`	`( inc index -- )`	Increments the data at a given indexed cell.
`-to:`	`( dec index -- )`	Decrements the data at a given indexed cell.
`fill:`	`( val -- )`	Stores `val` in each cell of the array

display

`print:`	`( -- )`	Types the index number and stored value of each cell with line break.

Error messages

"The index is Out of Range!"
As for `Indexed-Obj`.

1.2.3 Simplified array classes – GbArray, GwArray, GArray, GUArray, ZArray

These array classes don't have range check and are not good for subclassing. But they are simple and quick. If what you need is a bare array and if you are sure there is no need of range check, these array classes will be useful.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	None
Indexed data	1, 2-byte, 4-byte signed/unsigned, 8-byte cells
System object	None

Methods

Same as basic array classes.

Error messages

None

1.2.4 X-Array

`X-Array` is an array with the ability to execute its indexed data as `xt` of Mops word.

Superclass	`ARRAY` (1.2.2)
Source file	class-base
Status	Core
Instance variables	None
Indexed data	4-byte cells
System object	None

Methods

accessing

`exec:`	`( ind -- )`	Execute the `xt` in the indexed cell at `ind`.
`put:`	`( x0 x1 ... x(n-1) n -- )`	Stores n `xt`s into the elements of the object. xt_0 goes into element 0, xt_1 into element 1 and so on.
`actions:`	`( x0 x1 ... x(n-1) n -- )`	A synonym for `put:`.

display

`print:`	`( -- )`	Types the name of the word whose `xt` is in each element.

initialization

`classinit:`	`( -- )`	Sets all indexed elements to the `null xt`.

Error messages

"Wrong number of xts in list!"
For `put:`, the value n did not match the number of indexed elements for this object.

1.2.5 Sequence

`Sequence` is a generic superclass for classes which have multiple items which frequently looked at in sequence. At present the main function of `Sequece` is to implement the `each:` method, which makes it very simple to deal with each element. the usage is

```
BEGIN each: <obj> WHILE <do something to the element> REPEAT
```

`Sequence` can be multiply inherited with any class wihcj implements the `first?:` and `next?:` methods. The actual implementation details are quite irrelevant as long as these methods are supported.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`Bool each_started?`
Indexed data	None
System object	None

Methods

`each:`	`( ?? -- ?? b )`	Initiate processing of a sequence as in the example above. Input and output parameters depend on `first?:` and `next?:` methods of subclass.
`uneach:`	`( -- )`	Terminates processing of sequence before the normal end. Use prior to an `EXIT` out of an `each:` loop.

Error messages

None

1.2.6 Object_Array

`Object_Array` has been introduced as a replacement of `Obj_Array`. It's a generic superclass which makes it easy to generate an array of objects of a given class. Just define a new class which multiply inherits from the given class (or classes) and `Object_Array` (which must come last). This will add an indexed section to each object of the new class, with elements wide enough to contain objects of the original class. Then at run time you send `setCursor:` to the array and pass a reference which becomes the "cursor". It must have `no_subclasses`, with its class agreeing with the class of the elements.

```
:CLASS YourArrayClass super{ someclass Object_Array }
...
;CLASS

YourArrayClass theObjArray

ref someclass current-item no_subclasses
ref> current-item setCursor: theObjArray \ ok also in some definition
...
message: current-item \ send message to the current item object in the array
```

Now you can send `index:` messages to the array, and the cursor is set to point to the indexed element.

The array elements don't have an object header – they're just elements. The class info is in the reference. This is why the classes must agree exactly, and `no_subclasses` is required. This is checked in the `setCursor:` method, and gives an error if these conditions aren't met.(The description of this class is almost a copy of that in `pStruct` source code file of PowerMops.)

Superclass	`Indexed-obj` (1.2.1), `Sequence` (1.2.5)
Source file	class-base
Status	Core
Instance variables	`UINT Current` `ZVAR Cursor`
Indexed data	None
System object	None

Methods

accessing

`setCursor:`	`( ^ref -- )`	Sets the cursor reference to the object array. The cursor reference represents an object at the current index of the object array. "`ref>`" prefix is necessary just before the reference to get the address for the paramater.
`index:`	`( ind -- )`	Sets the cuttent index to `ind`.
`current:`	`( -- ind )`	Returns the current index number.
`at:`	`( ind -- ^elem )`	Returns the address of the item at `ind` in the array.
`first?:`	`( -- false | ^obj true )`	If the cursor has been set, sets the current index to 0 and returns the address of the first item object. Otherwise, returns `false`.
`next?:`	`( -- false | ^obj true )`	If there is the next item of the current, sets current index to the next and returns the address. Otherwise, returns `false`.

Error messages

"*Not a reference address. Maybe, you fogot* `ref>`."
"*Reference must use* `no_subclasses`!"
"*Referece class incompatible!*"

When `setCursor:`, the passed-in reference address was
not a valid reference address
had been declaired without `no_subclasses`
of different class from that of the item object.

1.2.7 (list)

`(list)` is a generic superclass and defines a common part of pointer based object list classes.

Superclass	`Sequence` (1.2.5)
Source file	class-base
Status	Core

Instance variables	`Ptr theList`
	`UVAR size`
	`UVAR pos`
	`UBYTE width`
	`BOOL refs?`

Indexed data	None
System object	None

Methods

accessing

`select:`	`( ind -- )`	Makes the item at the passed-in index current.
`selectlast:`	`( -- )`	Makes the last item current.
`current?:`	`( -- b )`	If current index points an existing item in the list, returns true. Otherwise, returns false.
`current:`	`( -- n )`	Returns 1 cell value (typically an object address) at the current item pointer. If there is no current item, returns nul.
`obj:`	`( -- n )`	Synonym of `current:`.
`size:`	`( -- n )`	Returns the current number of items in the list.
`add:`	`( item -- )`	Adds a new item in the list.
`remove:`	`( -- )`	Removes current item from the list.
`clear:`	`( -- )`	Releases `theList` and clears ivars, `size` and `pos`.
`release:`	`( -- )`	Calls `clear:`.
`first?:`	`( -- ^item true | false )`	Makes the first item current and returns the address and true. If there is no item in the list , returns false.
`next?:`	`( -- ^item true | false )`	Makes the next item of the current one current and returns the address and true. If the last item is current, returns false.
`saveEach:`	`( -- in-each? pos )`	Saves `each_started?` flag and current position on the data stack. This will be usable for a nested `each:` loop combined with `restoreEach:` method.
`restoreEach:`	`( in-each? pos -- )`	Restored `each_started?` flag and previous position from the data stack. This will be usable for a nested `each:` loop combined with `saveEach:` method.

display

`dumpList:`	`( -- )`	Types the size, current position and item width of the list and the contents of `theList` in hexadecimal format.
`dumpAll:`	`( -- )`	Same as `dumpList:` except that if the list has no item, types "(not open)". This method may be overridden in subclasses.

Error messages

"*The index is Out of Range!*"
When `select:`, parameter `ind` was out of the range of the list object.

1.2.8 Ptr_List

`Ptr_List` provides a list of pointers. Pointers in the list can be any memory addresses pointing to object base or allocated heap area.

Note that `release:` methods in this class is inherited from `(list)`, which will not send `release:` message to each item object. So, if you add heap object address to `Ptr_List`, you should take care of `release:` of the item object (or `free()` if it is a mere heap block).

Superclass	`(list)` (1.2.7)
Source file	class-base
Status	Core
Instance variables	None (see `(list)`)
Indexed data	None
System object	None

Methods

accessing

`new:`	`( size -- )`	Allocates heap memory of `size` bytes and adds the pointer into the list.

initialization

`classinit:`	`( -- )`	Sets item `width` 8 (1 cell bytes).

Error messages

None

1.2.9 Object_List

`Object_List` provides a list of heap-based objects of arbitrary classes. It uses the heap object reference feature, which includes garbage collecting system. We expect that elements will be added to the end, and probably not removed at all, or not very often.

Superclass	`(list)` (1.2.7)
Source file	class-base
Status	Core
Instance variables	None (see `(list)`)
Indexed data	None
System object	None

Methods

accessing

`add:`	`( ^obj -- )`	Adds the passed-in object to the list.
`new:`	`( [#elem] ^class -- )`	Creates a heap-based object of the passed-in class and adds it to the list.
`newObj:`	`( [#elem] ^class -- )`	Synonym of `new:`.
`top:`	`( -- )`	Makes the last item object in the list current. It takes a list like a stack.
`drop:`	`( -- )`	Removes the last item object from the list, treating a list like a stack.
`remove:`	`( -- )`	Removes the current item object from the list.
`obj:`	`( -- ^obj )`	Returns the base address of the currently selected item object.
`release:`	`( -- )`	Discounts reference numbers of all heap object items in the list, and `release:` the list.
`clear:`	`( -- )`	Synonym of `release:`.
`currentIndex#:`	`( -- idx )`	Returns currently selected item's index number.

initialization

`classinit:`	`( -- )`	Sets `width` to 16 (reference data size).

display

`dumpAll:`	`( -- )`	Types the contents of the data area of the list, then that of each item in hexadecimal format.

persistence/serialization

`send:`	`( ^obj -- )`	Serializes the whole list, by sending `send:` to each object in the list.
`bring:`	`( ^obj -- )`	Recreate each object in the list and send `bring:` to each one, so that the entire list is reconsituted.

1.2.10 (Col),ordered-Col, wordCol, byteCol

Collections are ordered list with a current size, that can also behave like a stack. We implement them by multiply inheriting the generic `(Col)` class with an array class of the appropriate width. `(Col)` adds the concept of a current size to the array methods.

Note: `ordered-Col`, `wordCol`, `byteCol` are 32, 16, 8 bit collections respectively. All methods are identical to `(Col)` with relevant array classes.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`INT Size`
Indexed data	None (supplied by the array class)
System object	None

Methods

accessing

`size:`	`( -- #elements )`	Returns the number of elements currently held in the list. This must always be less than or equal to `limit:`.
`add:`	`( val -- )`	Appends value in the next available cell, and increment `Size` by 1. An error occurs if size=limit before the operation (list full).
`last:`	`( -- val )`	Fetches the contents of the cell last added to the list. Error if the list is empty.
`remove:`	`( ind -- )`	Deletes the element at `ind` from the list, and reduce the `Size` by 1. Error if the list is empty.
`clear:`	`( -- )`	Sets list to empty.
`indexOf:`	`( val -- ind T | F )`	Searches for `val` within the current list, and returns the index and a `true` boolean if it was found, and `false` boolean if not found.

Error messages

"*The list is empty!*"
`remove:` or `last:` was attempted on an empty list.
"*The list is full!*"
`add:` was attempted with size=limit.

1.2.11 X-Col

The class is a collection of execution tokens. It adds one new method, and overrides one method of `X-Array`.

Methods

`removeXT:`	`( xt -- )`	Removes the earliest `xt` equal to the passed-in `xt`. Does nothing if no match found.
`print:`	`( -- )`	As for print of `X-Array`, but only types the `xt` names that are actually in the collection.

Error message

As for `(Col)`

1.3 Utility

1.3.1 Dic-Mark

`Dic-Mark` markes a dictionary position, and includes methods for traversing the dictionary.

Superclass	`Object` (1.1.1)
Source file	class-base
Status	Core
Instance variables	`ZARRAY LINKS` `UVAR CURRENT`
Indexed data	None (supplied by the array class)
System object	`theMark` (defined in "inspectors" initially not loaded)

Methods

accessing

`current:`	`( -- addr )`	Returns the current position.

manipulation

`set:`	`( addr -- )`	Sets the current position to `addr` (setting the array `Links` appropriately.
`setToTop:`	`( -- )`	Sets the current position to the top of the dictionary.
`next:`	`( -- addr )`	Moves the current position to the preceding dictionary word, and returns the address of the link field of the current word. Returnds zero if we were already at the base of the dictionary.

Error messages

None

1.4 FP data class

1.4.1 FVAR

`FVAR` supports elementaly FP calculations on double (8 bytes) FP number.

Superclass	`Object` (1.1.1)
Source file	fpelements
Status	CORE
Instance variables	`8 BYTES __FPCELL`
Indexed data	None
System object	None

Methods

accessing

`put:`	`(f: r -- )`	Stores the passed-in FP number in the data area.
`get:`	`(f: -- r )`	Returns the FP number stored in the data area.
`+:`	`(f: r -- )`	Adds `r` to the content of the data area.
`-:`	`(f: r -- )`	Subtracts `r` from the content of the data area.
`*:`	`(f: r -- )`	Multiplies the content of the data area by `r`.
`div:`	`(f: r -- )`	Divides the content of the data area by `r`.
`clear:`	`( -- )`	Stores FP `0.0` to the data area.

initialization

`classinit:`	`( -- )`	Clears the data area.

Error message

None

1.4.2 CGRect

`CGRect` provides some elementary functionalities to use Core Graphics Rectangle structure in Mac OS X. `CGRect` and `NSRect` have the same structure whose contents are 4 double(8byte) floating point numbers (totally 32 bytes length). The first 2 form the Origin Point structure and the last 2 the Size.

Note that the origin of the coordinate system is at Left-Bottom and that Y-axis grows upper.

A global `value __CGContext` is defined. If you get CGContextRef of a window, put it to `__CGContext`, then you can `draw:` a CGRect instance on the window. But note, CGContextRef returned from Core Graphics system calls is a temporary object. That means such a CGContextRef will be released and become invalid just after the event handler is returned. So, when you draw a CGRect on a window, Getting CGContextRef and doing `draw:` should be within one event-handler.

Superclass	`Object` (1.1.1)
Source file	WindowClass
Status	Core

Instance variables	`FVAR XO`
	`FVAR YO`
	`FVAR WIDTH`
	`FVAR HIGH`

Indexed data	None
System object	FrameRect, TempRect

Methods

accessing

`put:`	`( x0 y0 wid hi -- )`	Sets the rectangle from integers.
`setOrigin:`	`( x0 y0 -- )`	Sets the Left-bottom of the rectangle from integers.
`setSize:`	`( wid hi -- )`	Sets the size of the rectangle from integers.
`putX0:`	`( f: x0 -- )`	Sets the X-coordinate of the origin with the passed-in FP number.
`putY0:`	`( f: y0 -- )`	Sets the Y-coordinate of the origin with the passed-in FP number.
`getOrigin:`	`( f: -- x0 y0 )`	Returns the origin coordinates in FP.
`getSize:`	`( f: -- wid hi )`	Returns Size (width, height) in FP.
`getX0:`	`( f: -- x0 )`	Returns X-coordinate of the origin in FP.
`getY0:`	`( f: -- y0 )`	Returns Y-coordinate of the origin in FP.
`getWid:`	`( f: -- wid )`	Returns the width of the rectangle in FP.
`getHi:`	`( f: -- wid )`	Returns the height of the rectangle in FP.

manipulation

`addToX0:`	`( f: r -- )`	Adds the passed-in FP number to the X-cordinate of the origin.
`addToY0:`	`( f: r -- )`	Adds the passed-in FP number to the Y-cordinate of the origin.

display

`draw:`	`( -- )`	Draws the rectangle on the content of `__CGContext`.
`paint:`	`( -- )`	Same as `draw:` but with the content of the rectangle filled.

Error messages

None

1.5 Optional data structure

1.5.1 Complex

`Complex` supports a complex number data structure with fp number coefficients.

Superclass	`Object` (1.1.1)
Source file	mathnums
Status	optional
Instance variables	`FVAR REAL`
	`FVAR IMAGINARY`
Indexed data	None
System object	None

Method

accessing

`>R:`	`( f: r -- )`	Store `r` to the `REAL` part of the complex number.
`>I:`	`( f: r -- )`	Store `r` to the `IMAGINARY` part.
`getR:`	`( f: -- r )`	Returns the `REAL` part of the complex number.
`getI:`	`( f: -- r )`	Returns the `IMAGINARY` coefficient part of the complex number.
`+R:`	`( f: r -- )`	Adds `r` to the `REAL` part.
`+I:`	`( f: r -- )`	Adds `r` to the `IMAGINARY` part.
`-R:`	`( f: r -- )`	Subtracts `r` from the `REAL` part.
`-I:`	`( f: r -- )`	Subtracts `r` from the `IMAGINARY` part.

`set:`	`( f: r i -- )`	Sets `r` to the `REAL` part and `i` the `IMAGINARY` part.
`conj:`	`( -- )`	Converts to the conjugate complex number.
`->:`	`( ^complex -- )`	Initialize by the passed-in complex munber object. The input patrameter should be the base address of an instance of `Complex` class.
`+:`	`( ^complex -- )`	Adds the passed-in complex number to this one.
`-:`	`( ^complex -- )`	Subtracts the passed-in complex number from this one.
`*:`	`( ^complex -- )`	Multiplies this complex number by the passed-in one.
`div:`	`( ^complex -- )`	Divides this complex number by the passed-in one.
`length:`	`( f: -- r )`	Returns the distance from the origin to the complex number in the Gaussian Plane.
`arg:`	`( f: -- r )`	Returns the argument of the complex number. ($-\pi < \theta \leq \pi$)

display

`print:`	`( -- )`	Prints the real and imaginary coefficients in scientific format on the screen.

Error messages

None

1.5.2 fpMatrix

`fpMatrix` supports elementary matrix calculations with FP elements. `fpMatrix` is heap based. So it is different from `fMatrix` in PowerMops. `fpMatrix` needs `init:` with row-column parameters to allocate the heap block before use.

Superclass	`Ptr` (1.1.10)
Source file	mathnums
Status	optional
Instance variables	`UVAR ROW#` `UVAR COL#`
Indexed data	None
System object	None

Method

initialization

`init:`	`( row col -- )`	Set rows and colums, allocates heap block for the matrix data area, then clears the heap block.

accessing

`getSize:`	`( -- row col )`	Returns the numbers of rows and columns.
`row#:`	`( -- n )`	Returns the number of the rows.
`col#:`	`( -- n )`	Returns the number of the columns.
`set:`	`( f:` $r_{11}\ r_{12}\ \ldots r_{1m}\ r_{21}\ \ldots r_{nm}$ `-- )`	Sets the contents of the matrix from FP stack.
`->:`	`( ^matrix -- )`	Initializes and sets the contents from the passed-in matrix object. The parameter must be the base address of an instance of `fpMatrix` class.
`^elem:`	`( r c --` $\hat{}elem_{rc}$ `)`	Returns the address of (r,c)-element of the matrix in heap block. Both `r` and `c` are 1-base (no 0-th).
`to:`	`( r c -- ) (f: r' -- )`	Store `r'` to this matrix as the (`r`,`c`)-element.
`at:`	`( r c -- ) (f: -- r' )`	Returns (`r`,`c`)-element of this matrix.
`=size?:`	`( ^matrix -- b )`	Compares the size with that of the passed-in matrix, then returns `true` if the sizes are identical or `false` otherwise.
`row:`	`( n -- ) ( f: --` $r_{n1}\ r_{n2}\ldots$ `)`	Returns `n`-th row.
`col:`	`( m -- ) ( f: --` $r_{1m}\ r_{2m}\ldots$ `)`	Returns `m`-th column.
`tr:`	`( f: -- r )`	Returns the trace (diagonal sum) of this matrix.

manipulation

`+:`	`( ^matrix -- )`	Adds the passed-in fpMatrix to this one.
`-:`	`( ^matrix -- )`	Subtract the passed-in fpMatrix from this one.
`scprod:`	`( f: r -- )`	Scalar-multiplies this matrix by `r`.
`l-acts:`	`( ^matrix -- )`	Transforms this matrix by the action of the passed-in fpMatrix from LEFT.
`r-acts:`	`( ^matrix -- )`	Transforms this matrix by the action of the passed-in fpMatrix from RIGHT.
`*=>:`	`( ^matrix1 ^matrix2 -- )`	Creates this matrix by the matrix product of the two passed-in fpMatrices.

display

`print:`	`( -- )`	Prints the contents of this matrix in scientific format on the screen.
`dump:`	`( -- )`	Prints the data area in hex format, then `print:`.

Error messages

"*column# out of limit!*"
When accessing an element or `^elem:`, the column parameter was out of the limit.

"*row# out of limit!*"

When accessing an element or `^elem:`, the row parameter was out of the limit.

"*Matrices size collision!*"

When matrix addition or subtraction, operands matrices have mutually different types so that the operation couldn't be completed.

"*column-row mismatch!*"

When matrix multiplication, the column number of the left and the row number of the right are different so that the operation couldn't be completed.

"*The result matrix would be different type!*"

When left or right action, the target matrix shouldn't change the type. But after the action you tried, the type would change so that the operation couldn't be completed.

Chapter 2

String

introduction

Mops strings are implemented as blocks of heap that can expand and contract as their contents change. A string object itself contains a pointer to the heap block that contains the string's data. It also contains three other ivars which we will describe below.

Strings can be useful for a wide variety of programming needs. They can serve as file buffers, staging area for text to be printed on the screen, dictionaries, or vehicles for parsing user input. You should consider using strings for any run of bytes whose length and/or contents are likely to change in the course of your program's execution. Strings are not restricted to ASCII text, although that will probably be their most common use. Note, however, that text constants can more efficiently be implemented as SCONs, string literals or CString.

When a string is no longer needed, you should send it `release:` to release the heap block. In iMops, a string object doesn't use handle but pointer system, so that `new:` method to allocate the base pointer in advance lost its meaning. `new:` in iMops is a method for allocating heap area as a buffer. So it needs `size` parameter. `put:` methods tries to release the heap block at first, and create wholly new string object. So you don't need `release:` before `put:`.

iMops' string class supports only elementary features. It has search feature, but always case-sensitive. Case-insensitive seach or other advanced features will be added as an optional library in the future. However, string objects are used for various purposes internally in iMops.

As an interface with Mac OS X library (Objective C), a word `copy2CFStr` is defined (in "inuc2").

```
copy2CFStr ( addr len -- CFStringRef )
```

The word gets the address and length of an ordinaly character string and create CFString object the contents of which is a copy of the passed-in character string. CFString (Core Foundation class) and NSString (Objectve C Foundation class) are one same thing. So you can pass the string `copy2CFStr` returns to Objective C method as a NSString parameter. CF/NSString created by `copy2CFStr` is owned by iMops, so you should release the object when it becomes no longer necessary. When you pass the CF/NSString to some objecive C method, it is usually ok to release it immediately after the passing. Just after creating CF/NSString by `copy2CFStr`, the object reference is stored in a zvalue `tmpStr`. So, you can release the string by

```
tmpstr CFRelease \ as a CFString
```

or

```
tmpstr ReleaseObjC \ as a NSString
```

`copy2CFStr` is supposing temporary and one-at-once use of the string to be returned. For a true support of CF/NSString, Mops class for that should be defined (when needed in the future).

2.1 Fundamental string class

2.1.1 String

String defines a variable-length string with basic access methods whose data exists as a block of heap. Size is limited within 4GB (by 4byte UVar size).

Superclass	`Ptr` (1.1.10)
Source file	class-base
Status	Core

Instance variables	`UVar Pos`
	`UVar Lim`
	`UVar Size`
	`int flags`

Indexed data	None
System object	execNameStr, Cont-Plist, Prop-Base, Prop-str (Installer) File-Contents, Src-Paths, __loc-file, __BareFileName, locate-name (For Locate) $MWord (for cascading), $MethName (for error description)

Methods

accessing

`pos:`	`( -- u )`	Returns the value of `Pos`.
`>pos:`	`( u -- )`	Stores `u` in `Pos`.
`lim:`	`( -- u )`	Returns the value of `Lim`.
`>lim:`	`( u -- )`	Stores `u` in `Lim`.
`len:`	`( -- n )`	Returns the value of `Lim` − `Pos`.
`>len:`	`( n -- )`	Adds `n` and the value of `Pos`, then stores the result to `Lim`.
`skip:`	`( n -- )`	Adds `n` to `Pos`.
`more:`	`( n -- )`	Adds `n` to `Lim`.
`start:`	`( -- )`	Clears `Pos`, so that the active part now starts at the "real" start of the string.
`begin:`	`( -- )`	Clears both `Pos` and `Lim`. Useful for setting up for an iterative operation on the string.
`end:`	`( -- )`	Sets both `Pos` and `Lim` to `Size` (i.e. the end) of the string. Useful for setting up for an iterative operation which has to go backwards through the string.
`nolim:`	`( -- )`	Sets `Lim` to the end of the string.
`reset:`	`( -- )`	Clears `Pos` and sets `Lim` to the end of the string. The active part will now be the whole string.
`step:`	`( -- )`	Steps forward in the string, setting `Pos` to `Lim` and then setting `Lim` to the end of the string.
`<step:`	`( -- )`	Steps backward in the string, setting `Lim` to `Pos` and then clearing `Pos`.

manipulation

`new:`	`( size -- )`	Creates a heap block for the string's data whose length is equal to the passed-in `size`, and sets the pointer.
`?new:`	`( -- )`	Ensure a heap block is allocated. If a block is already allocated, does nothing.
`size:`	`( -- n )`	Returns the size of the (whole) string.
`setSize:`	`( n -- )`	Sets the size of (whole) string to `n`, then does a `reset:`.
`clear:`	`( -- )`	Ensure a heap block is allocated, and sets its `Size` 0.
`get:`	`( -- addr len )`	Returns the addresss and length of the active part of the string.
`all:`	`( -- addr len )`	Returns the address and lenght of the entire string (not just active part).
`1st:`	`( -- c )`	Returns the character at `Pos`.
`^1st:`	`( -- ^c )`	Returns the address at `Pos`.
`>uc:`	`( -- )`	Converts the active part to upper cases.
`uc:`	`( -- addr len )`	Converts the active part to upper cases and do `get:`.
`put:`	`( addr len -- )`	Ensure a heap block is allocated, then replace it with passed in string, and does `reset:` as well.
`->:`	`( ^str -- )`	Replaces the whole of the string (as in `put:`) with the active part of `^str`, which may belong to string class or the subclass.
`dup:`	`( ^str -- )`	Creates the string whose contents, `Pos` and `Lim` are same as `^str`, which may belong to string class or the subclass.
`insert:`	`( addr len -- )`	Ensures a heap block is allocated, then inserts the string given by `addr len` at `Pos`. Increments both `Pos` and `Lim` by `len` (thus the bytes at the `Pos` and `Lim` position will be the same as before, and the byte immediately preceding the `Pos` position will be the last of the inserted bytes).
`$insert:`	`( ^str -- )`	Insert the active part of `^str`, as for `insert:`.
`chinsert:`	`( c -- )`	Insert the character `c`, as for `insert:`.
`add:`	`( addr len -- )`	Inserts the `addr len` string at the end of this string. `Pos` and `Lim` are then set to the (updated) end position.
`+:`	`( c -- )`	Appends the character `c` to the end of the string, and sets `Pos` and `Lim` to the (updated) end position.
`fill:`	`( c -- )`	Overwrite each characters in the active part of the string with the character `c`.
`search:`	`( addr len -- b )`	Searches the active part of the string, starting from the left (i.e. the `Pos` position), for the string (`addr len`). If a match is found, `Lim` is set to indicate the first of the matching characters and true is returned. If no match is found, `Lim` is unchanged and false is returned.
`chsearch:`	`( c -- b )`	Searches the active part of the string for the character `c`. If it is found, `Lim` is set there and true is returned. If it isn't found, `Lim` is unchanged and false is returned.
`<chsearch:`	`( c -- b )`	Backward search for the character `c`. If found, sets `Pos`.
`delete:`	`( -- )`	Deletes the active part, then `Lim` is set to `Pos`.
`append0:`	`( -- )`	Append 0 at the end of the string to make C-format string.
`clearAll:`	`( -- )`	Clears whole data area. Useful to clear littered data area.

display

`print:`	`( -- )`	Types the active part of the string, as a UTF-8 character string.
`dump:`	`( -- )`	Dumps all data in hex, and string contents in hex numbers and characters.
`rd:`	`( -- )`	`reset:` and `dump:`. An abbreviation.

stream interface

`read:`	`( addr len -- code )`	Copies the active part of the string to the memory area given by `addr len`. Updates `Pos` by the number of bytes transferred. Returns zero if all the active part is transferred, -1 if not (i.e. the length of the active part was greater than `len`).
`write:`	`( addr len -- 0 )`	Similar to `add:`. Always returns zero, indicating success.

persistence/serialization

`send:`	`( ^obj -- )`	Serialize the string , by first sending the ivars, then the string itself.
`bring:`	`( ^obj -- )`	Reconstitutes the string as serialized by `send:`.

Error messages

"*String pointer(s) out of bounds!*"
`Pos` was found to be greater than `Lim`, or either was negative or greater than the size of the string. `Pos` and `Lim` are also displayed when this mesage is given. We check for this error whenever we access the actual characters of the string. Operations such as `>pos:` don't perform the check — this is for speed, and also because when we are doing manipulations on `Pos` and `Lim` we don't want to put any restriction on intermediate values.

"*Can't do that on a string copy!*"
You attempted to insert, delete, or change the size of the string object which was flagged as a "copy". See above under `copyTo:`

2.2 Optional string classes

2.2.1 TrTbl

`TrTbl` (Translate Table) class provides a functionality to search a specified set of characters in a string. The searching is very fast compared to a normal character search. The instance is used in `STRING+` class, which allows uncluttered and extremely fast search operation in `scan:` and `<scan:` methods.

Superclass	`Object` (1.1.1)
Source file	String+
Status	Optional
Instance variables	`UInt count` `256 bytes theTbl`
Indexed data	None
System object	`ucTbl`

Methods

accessing

`tbl:`	`( -- addr )`	Returns the Table address.
`put:`	`( addr len -- )`	Puts the passed-in string on the table for translate.
`selChar:`	`( c -- )`	Selects the given character.
`selCharNC:`	`( c -- )`	Selects a character, and if it is a letter, enters the same value in the LC (lower case) and UC (upper case) positions of the table, so that case will in effect be ignored when the table is used.
`selChars:`	`( addr len -- )`	Selects table bytes according to the characters in the passed-in string. The number of the order of a character in the selected characters is put into the table byte corresponding to the character. When two or more characters in the string are same, the first happening is taken as the order number.
`selRange:`	`( lo hi -- )`	Selects the range from the `lo` character to the `hi` character of the table (inclusive).
`invert:`	`( -- )`	Changes the selected characters into the not-selected, the not-selected into the selected. The table bytes corresponding to the newly selected characters are set to be -1.
`>uc:`	`( -- )`	Copies the 26 bytes corresponding to A-Z into the a-z positions. So that the translation operation using this table object will give identical results for upper and lower letters.
`transc:`	`( c -- c' )`	Translates one character using the table. Returns the corresponding byte `c'` from the table.

initialization

`clear:`	`( -- )`	Clears the whole contents of the object.

Error messages

None.

2.2.2 String+

`String+` class adds case-insensitive search, non-character string and other useful features to `String` class.

Byte string is **Big Endian** by default. Declaring the constant `BIG-ENDIAN?` as `false` at the start of the source file will change the mode.

Superclass	`String` (2.1.1)
Source file	String+
Status	Optional
Instance variables	None (See `String` class(2.1.1))
Indexed data	None
System object	None

Methods

accessing

`last:`	`( -- c )`	Returns the last character in the string.
`Line>:`	`( -- )`	Sets the `Lim` to the end of the current line (i.e. the next return character, or the end of the string). `Pos` isn't moved.
`nextLine?:`	`( -- b )`	Sets the `Pos` and `Lim` to delimit the next line. If `Lim` initially doesn't point to a new line character, the "next" line will actually be the rest of the current line. Returns `true` when there is the next line (meaning unless the `Lim` is already at the end of the string). `false` otherwise.
`<nextLine?:`	`( -- b )`	Backward `nextLine?:`.
`readN:`	`( ^file n -- )`	Reads `n` bytes from the given file. `^file` should be the base address of a file class instance or other stream-type object which has file-type methods. The file must have been opened before.
`readRest:`	`( ^file -- )`	Reads all the remainder of the given file into `self`.
`readAll:`	`( ^file -- )`	Reads all the contents of the given file into `self`.
`compare:`	`( addr len -- n )`	Compares the `addr len` string with the active part of `self`. The comparison is CASE-SENSITIVE when the global flag `CASE?` (a `VALUE`) is `true`, CASE-INSENSITIVE otherwise. Returns 0 when the strings are same, -1 when the passed-in string precedes in dictionary order (provided, the char order is UTF-8), 1 when the passed-in string follows in dictionary order.
`?:`	`( addr len -- n )`	Same as `compare:`, except that if the `addr len` string is shorter than the active part of `self`, only `len` characters from the current position of `self` are used.
`=?:`	`( addr len -- b )`	A compare for equal/not equal only.
`ch=?:`	`( c -- b )`	Compare the given character against the character at `Pos`.
`scan:`	`( ^trtbl -- n )`	Scans the active part from the beginning for any character selected in the passed-in `TrTbl`. If found, `Lim` is set just before the character and the content of the table byte corresponding to the character is returned. If not found, returns 0.
`<scan:`	`( ^trtbl -- n )`	Backward `scan:`
`scax:`	`( ^trtbl -- n )`	`scan:` for NOT-selected characters in the TrTbl.
`<scax:`	`( ^trtbl -- n )`	Backward `scax:`

manipulations

`ovwr:`	`( addr len -- )`	Overwrites the active part of `self` with the string (`addr len`). Copying stops at the end of the active part, or when `len` characters have been transferred. `Pos` is incremented by the number of characters transferred.
`chovwr:`	`( c -- )`	Overwrites the first character of the active part of the string (if any) by the char `c`.

`$ovwr:`	`( ^str -- )`	Overwrites the active part of `self` with the active part of the passed-in string object.
`addLine:`	`( addr len -- )`	Adds the `addr len` string as a new line at the end of the `self`.
`$addLine:`	`( ^str -- )`	Adds the active part of the passed-in string object as a new line at the end of `self`.
`translate:`	`( ^trtbl -- )`	Translates the characters of the active part of `self` using the passed-in `TrTbl` object.
`trans1st:`	`( ^trtbl -- c )`	Returns the result of a translation of the first character of the active part of `self` using the passed-in `TrTbl` object. Returns 0 if `self` is empty.

non-character string

`1stW:`	`( -- u )`	Returns the first 2byte number at `Pos`.
`1stL:`	`( -- u )`	Returns the first 4byte number at `Pos`.
`1stZ:`	`( -- n )`	Returns the first 8byte number at `Pos`.
`>1st:`	`( c -- )`	Stores the passed-in 1byte number at `Pos`.
`>1stW:`	`( n -- )`	Stores the passed-in 2byte number at `Pos`.
`>1stL:`	`( n -- )`	Stores the passed-in 4byte number ar `Pos`.
`>1stZ:`	`( n -- )`	Stores the passed-in 8byte number ar `Pos`.
`nxtC:`	`( -- c )`	Sends `1st:` to `self`, then sets `Pos` at the next byte. (Note that methods of this type DON'T check the `Lim` nor `Size`.)
`nxtW:`	`( -- u )`	Sends `1stW:` to `self`, then sets `Pos` at the next 2byte.
`nxtL:`	`( -- u )`	Sends `1stL:` to `self`, then sets `Pos` at the next 4byte.
`nxtZ:`	`( -- n )`	Sends `1stZ:` to `self`, then sets `Pos` at the next 8byte.
`nxtN:`	`( n -- n' )`	Returns the first `n`-byte number in the active part and sets `Pos` to the next `n`-byte. Returns 0 if `n` is larger than the length of the active part. If `n` is larger than 8, the higher bytes will be cut off.
`>nxtC:`	`( c -- )`	Sends `>1st:` to `self`, then sets `Pos` at the next byte.
`>nxtW:`	`( n -- )`	Sends `>1stW:` to `self`, then sets `Pos` at the next 2byte.
`>nxtL:`	`( n -- )`	Sends `>1stL:` to `self`, then sets `Pos` at the next 4byte.
`>nxtZ:`	`( n -- )`	Sends `>1stZ:` to `self`, then sets `Pos` at the next 8byte.
`>nxtN:`	`( val n -- )`	Stores the passed-in value `val` as a `n`-byte number at `Pos`, then advances `Pos` by `n`.
`+C:`	`( c -- )`	Synonym of `+:`.
`+W:`	`( n -- )`	Appends the passed-in 2byte number to the contents of `self`.
`+L:`	`( n -- )`	Appends the passed-in 4byte number to the contents of `self`.
`+Z:`	`( n -- )`	Appends the passed-in 8byte number to the contents of `self`.
`+N`	`( val n -- )`	Appends the passed-in value `val` as a `n`-byte number to the contents of `self`.
`count:`	`( -- )`	Assuming the substring starting at `Pos` is a counted string, sets `Pos` and `Lim` to delimit it.
`wCount:`	`( -- )`	Similar to `count:` except for assuming the substring, maybe non-alined, has a 2byte length, big-endian.

initialization

`new:`	`( -- )`	`new: self` with default string size.

Error messages

None.

Chapter 3

File

introduction

General use

File class defines elementary methods to manage a file on the disk. File object is also a typical container object for a serializations of iMops object.

In iMops, file navigation panel window is not a stand-alone class, but a mere instance variable of a file object. So, methods to manipulate a navigation dialog belong to the file class.

iMops file object gets a URL objective-C object from the file open navigation panel. URL object is a recommended file representative in Cocoa/Objective-C, so you can pass the object as a parameter on calling Objective-C method to manipulate a file.

Parallel to the URL object, iMops' File object keeps the path string. Ordinaly file manipulation methods, `open:` `create:` etc., uses the path string with a standatd C library call `fopen()`. You can set file path and/or name by `name:` method.

Before `read:` or `write:` from/to a file, the file should be opened by methods like `open:`, `create:` or `createNew:`.

`open:` method will seach the file in "Project paths". But project paths are supposed to be under iMops/source/ folder(directory), so that `open:` method will not seach the other places. If you need to open a file in the other location than iMops/source or under, do `setfullpath:` (not `name:`) to set full file path name, then `open:`. Or you can use file navigation dialog, too.

When you finished `read:` or `write:`, the file needs to be closed by `close:`.

iMops' file class supports only elementary part of file manupulations. Iterative operations using File Manager framework or other advanced features will be added as an optional library class in the future.

Navigation Panels

Application Modal Panel

`navGet:` is a method to get a file location information through an OpenFile navigation panel, while `navPut:` is to set the file location through a SaveFile navigation panel. Both methods themselves don't open or save the file. They simply set a file location data of the file object for open/saveing the file. `navGet:` and `navPut:` methods can be used in same way as those in PowerMops, except that the file type parameter for `navGet:` should be set as a constant, not resource ID. Those constants are `Plain-Text-file`, `All-Text-file` and `Image-File`. The other values will be taken as NO-FILTERING.

(However, the file type filtering by this method is not functioning on Mac OS X 10.6. Regardless of the file type constants, all files including packages are selectable in the navigation panel. A known bug)

Window Modal Panel

`navGetWinModal:` and `navPutWinModal:` get additional input parameters, an window object and Open or Save handler's xt, and open a window modal file navigation panel with regard to the passed-in window, then return immediately. When a button on the opened navigation panel is clicked, the passed-in handler `xt` will be executed with the stack parameter 1 (when Open/Save) or 0 (when cancel). The `xt` is executed as a callback in a different context from the main thread, so that the word represented by the `xt` cannot leave data on the stack. That is, the stack effect of the word should be `( 1 | 0 -- )`.

The simplest code for an Open file panel may be like following:

```
WINDOW+ WW
...
File myfile

: openHandler  handleOpen: myfile IF setok: myfile THEN ;
...
... new: WW .. show: WW

0 WW ' openHandler navGetWinModal: myfile
```

The word `openHandler` above gets the file location data and sets ok flag if Open button is clicked. The kernel is `handleOpen:` method of file class, which is called also from `navGet:` method. Which button was clicked can be checked by sending `ok?:` messeage to `myfile`. If Open button was clicked, the method will return true, otherwise false. Instead of "`setOK: myfile`", you can put there a code to really open `myfile`.

As for Save file panel:

```
Window+ WW
....

File myFile

: saveHandler  handleSave: myFile IF setok: myFile THEN ;

... new: WW  show: WW

: SAVENAV " filename" WW ['] saveHandler  navPutWinModal: myFile ...
```

The string "`filename`" will be inserted into the textinput pane of the Save file panel as the default initial file name.

Window Modal Panel with Context

`navGetCWinModal:` and `navPutCWinModal` methods take still one more parameter, the context object. These methods are defined only for convenience. But it is surely useful in some cases. The passed-in

object will become the context of the execution of the handler `xt`. The sample code from iBucket source (BktEDocuments):

```
:CLASS Maindow super{ Window+ }
 record
  {
   String tmpAlertText
   ZVAR FINDWIN
  }
  FILE theFile
  SCROLLVIEW myScroll
  sourceVIEW MainTV
...

: readFileHandler  handleOpen: theFile IF  theFile readfile: mainTV THEN ;

:m OpenFile:
  PLAIN-TEXT-FILE ^base dup [’] readFileHandler navGetCWinModal: thefile
;m

: saveFileHandler  handleSave: theFile IF theFile saveText: mainTV THEN  ;

:m saveAs:
  getfilename: theFile dup 0<= IF 2drop " untitled" THEN
  ^base dup [’] saveFileHandler navPutCWinModal: theFile
;m
..
```

Words `readFileHandler` and `saveFileHandler` are defined in a class definition context, so that the instance variables are accessible in the definitions. However, these words must be executed in context of some object of the class, say, in a method definition, because they need their object context (= the current object in their execution) to get to the instance variable. Setting the object context is incorporated in method calls but not in normal word calls.

However, unfortunately, `xt` passed to a method to open a window modal navigation window will be executed as a callback, not in some method. In order to solve the problem, `navGetCWinModal:` and `navPutCWinModal:` get as an additional parameter an object base address, which will be made the object context of the execution of the handler `xt`, so that `readFileHandler` or `saveFileHandler` can be executed without crash even though they are callbacks.

3.1 Fundamental file class

3.1.1 File

`File` class defines elementary methods to manipulate a file on Disk. A file navigation panel window (Open/Savepanel) is also supported.

Superclass	`Object` (1.1.1)
Source file	FileClass
Status	Core

Instance variables	`ZVar descriptor`
	`ZVar PanelObj`
	`ZVar myURL`
	`ZVar TempWindow`
	`ZVar Size`
	`ZVar tmpPos`
	`String fpathName`
	`bool completepath?`
	`bool ok?`

Indexed data	None
System object	PList, Instld-Exec (Installer) ProjPaths (for serializing project paths data)

Methods

initialization

`name:`	`( addr len -- )`	Sets the file name from the passed-in `addr len` string. `completepath?` flag is cleared.
`setfullpath:`	`( addr len -- )`	Sets the absolute full file path name from the passed-in string. `completepath?` flag is set.
`pathNSStr:`	`( ^NSStr -- )`	Sets the file object data based on the passed-in NSString object whose content is a file path.
`clearName:`	`( -- )`	Clears the inner string for the file name.

open-file panel

`navGet:`	`( ftype -- b )`	Opens a default OpenFile Naviagion panel as an **application modal** window, gets file location and the name through the panel. Valid file types (`ftype`) are Plain-Text-file, All-Text-File, Image-File and 0 (all).**(However, the filtering of the file type won't work on Mac OS X 10.6, at present.)** If a file is selected on the panel and the Open button is clicked, then the file path and URL are set and returns true. Otherwise, nothing is set and returns false.

`navGetWinModal:`	`( ftype ^win xt -- )`	Opens a default OpenFile naveigation panel as a window modal sheet window with regard to the passed-in window object (`^win`). Window object to be passed in is the base address of an iMops window class instance, which should have gotten `new:`. This method returns immediately. If a file is selected and Open button is clicked, the passed-in `xt` is executed with the stack parameter 1. If cancel button is clicked, the passed-in `xt` is executed with the stack parameter 0.
`navGetCWinModal:`	`( ftype ^win ^obj xt -- )`	Opens a default OpenFile naveigation panel as a window modal sheet window with regard to the passed-in window object (`^win`). Window object to be passed in is the base address of an iMops window class instance, which should have gotten `new:`. This method returns immediately. If a file is selected and Open button is clicked, the passed-in `xt` is executed with the stack parameter 1 where the current object is set "`^obj`". If cancel button is clicked, the passed-in `xt` is executed with the stack parameter 0.
`navGetModeless:`	`( ftype xt -- )`	Opens a default OpenFile naveigation panel as a modeless window. This method returns immediately. If a file is selected and Open button is clicked, the passed-in `xt` is executed with the stack parameter 1. If cancel button is clicked, the passed-in `xt` is executed with the stack parameter 0.

file operations

`open:`	`( -- rc )`	Searches the file within project paths if the full path name is not set. Then opens the file for read, write or accessing and returns 0 if the file found, returns -1 if not found.
`openReadonly:`	`( -- rc )`	Same as `open:` except for opening the file for read only.

`read:`	`( addr len -- rc )`	Reads `len` bytes into the buffer starting at `addr`. Returns 1 if succeeded but the end of file is not reached. 0 if the end of file is reached or if some error happened. Send `Error?:` or `EOF?:` to see whether some error happened or merely the end of the file is reached.
`write:`	`( addr len -- rc )`	Writes `len` bytes from the buffer starting at `addr`. Returns 1 if succeeded but the end of file is not reached. 0 if the end of file is reached or if some error happened. Send `Error?:` or `EOF?:` to see whether some error happened or merely the end of the file is reached.
`readLine:`	`( addr len -- rc )`	Reads `len` bytes into the buffer starting at `addr`. The read will terminate if a CR ($0D) or LF ($0A) is received. Returns 1 if succeeded, 0 if some error happened.
`moveTo:`	`( pos -- rc )`	Sets the file position indicator to `pos` relative to the beginning of the file. Returns 1 if succeeded, 0 if some error happened.
`move:`	`( offs -- rc )`	Sets the file position indicator to `pos` relative to the current position of the file. Returns 1 if succeeded, 0 if some error happened.
`rewind:`	`( -- )`	Sets the file position indicator to the beginning of the file.
`last:`	`( -- )`	Sets the file position indicator to the end of the file.
`close:`	`( -- rc )`	Closes the currently open file.
`delete:`	`( -- rc )`	Deletes the file (so, you should be careful). The file must not be open, or an error will result.

accessing

`pos:`	`( -- pos )`	Returns the current file position.
`bytesRead:`	`( -- #bytes )`	Returns the length of the read contents in bytes.
`restBytes:`	`( -- #bytes )`	Returns the length of the rest (=not read yet) contents in bytes.
`size:`	`( -- fsize )`	Returns the size in bytes of the file. Returns 0 when file is not opened.
`EOF?:`	`( -- b )`	Returns `true` if `read:` or `write:` has reached the end of the file. Returns `false` otherwise.
`Error?:`	`( -- rc )`	Returns error code if some error happened in file operation. Returns 0 (no error) otherwise.
`getFileName:`	`( -- addr len )`	Retuns `addr len` string of the file name.
`getWholePath:`	`( -- addr len )`	Retuns `addr len` string of the whole path string of the file.

`getURL:`	`( -- ^URL | 0 )`	Returns the objective-C URL object that represents the file for some objective-C process. Returns 0 if it is not set yet.
`getfpathPtr:`	`( -- ^str )`	Returns the pointer of the whole path string in C string format.
`setOK:`	`( -- )`	Sets `ok?` flag.
`ok?:`	`( -- b )`	Returns the state of `ok?` flag.
`resetURL:`	`( ^URL -- )`	Resets the file objects based on the passed-in objective-C URL object.
`release:`	`( -- )`	Releases path string and URL object if created, and clears all object data area.
`clear:`	`( -- )`	Simply clears the wole data area of the file object.

display

`printPath:`	`( -- )`	Types the full path string of the file on the screen.

save-file panel

`navPut:`	`( addr len -- b )`	Opens a default SaveFile Naviagion panel as an **application modal** window, gets file location and the name through the panel. The passed-in `addr len` string is inserted in the file name pane as a default file name. If a location is selected and the Save button is clicked, then the file path and URL are set and returns true. Otherwise, nothing is set and returns false.
`navPutWinModal:`	`( addr len ^win xt -- )`	Opens a default SaveFile naveigation panel as a window modal sheet window with regard to the passed-in window object (`^win`). Window object to be passed in is the base address of an iMops window class instance, which should have gotten `new:`. This method returns immediately. If Save button is clicked, the passed-in `xt` is executed with the stack parameter 1. If cancel button is clicked, the passed-in `xt` is executed with the stack parameter 0.

`navPutCWinModal:`	`( addr len ^win ^obj xt -- )`	Opens a default SaveFile naveigation panel as a window modal sheet window with regard to the passed-in window object (`^win`). Window object to be passed in is the base address of an iMops window class instance, which should have gotten `new:`. This method returns immediately. If Save button is clicked, the passed-in `xt` is executed with the stack parameter 1 where the current object is set "`^obj`". If cancel button is clicked, the passed-in `xt` is executed with the stack parameter 0.
`navPutModeless:`	`( addr len xt -- )`	Opens a default SaveFile naveigation panel as a modeless window. This method returns immediately. If Save button is clicked, the passed-in `xt` is executed with the stack parameter 1. If cancel button is clicked, the passed-in `xt` is executed with the stack parameter 0.

File Navigation panel

`createOpenPanel:`	`( -- )`	Creates a default OpenFile navigation panel for manipulations.
`navfType:`	`( ftype -- )`	Filters selectable file type on the OpenFile panel (Known bug: This method won't work.)
`handleOpen:`	`( 1 | 0 -- T | F )`	If the input is 1 (not 0), gets data from the OpenFile navigation panel, sets the file object and return `true`. If input is 0, do nothing and returns `false`. This method is supporsed to be used in combination with `navGet...` methods.
`createSavePanel:`	`( -- )`	Creates a default SaveFile navigation panel for manipulations.
`setPanelNameField:`	`( addr len -- )`	Sets `addr len` character string to the default file name for saving.
`NSPanelNameField:`	`( ^nsstr -- )`	Sets the passed-in CF/NSString to the default file name for saving.
`handleSave:`	`( 1 | 0 -- T | F )`	If the input is 1 (not 0), gets data from the SaveFile navigation panel, sets the file object and return `true`. If input is 0, do nothing and returns `false`. This method is supporsed to be used in combination with `navPut...` methods.

`setPanelPrompt:`	`( addr len -- )`	Sets `addr len` character string to the prompt text of the default button of the navigation panel.
`setPanelmessage:`	`( addr len -- )`	Types `addr len` character string on the navigation panel as the message text.
`setPanelTitle:`	`( addr len -- )`	Sets `addr len` character string to the title of the navigation panel.
`setWindowForNav:`	`( ^win -- )`	Sets the passed-in window object to the window for which a window modal navigation panel will be modal.
`navRunModal:`	`( -- )`	Opens the navigation panel created before, application modal.
`winModalPanel:`	`( xt -- )`	Opens the navigation panel having been created, modal for the window already set by `setWindowForNav:`, sets the passed-in `xt` to be the button click handler. The navigation panel should have been created before calling this method.
`navRunWinModal:`	`( ^win xt -- )`	Calls `setWindowForNav:` with the parameter `^win`, then calls `winModalPanel:` with the parameter `xt`.
`MWinModalPanel:`	`( ^obj xt -- )`	Opens the navigation panel having been created, modal for the window already set by `setWindowForNav:`, sets the passed-in `xt` to be the button click handler run with `^obj` being the current object.
`navRunModeless:`	`( xt -- )`	Opens the navigation panel having been created modeless, sets the passed-in `xt` to be the button click handler.

Error messages

None

3.2 File extension classes

None yet.

Chapter 4

Window

introduction

Window is one of the most fundamental GUI elements. `Window` and `Window+` classes are Mops classes to create and manipulate a window. `Window` class is for a bare window without any view system on it. `Window+` class is for a normal window with the view system on it. All GUI objects in Mops could be seen as kinds of proxies of corresponding system objects. So you need to send '`new:`' to a Mops GUI object at appropriate time to create a system object for it.

iMops uses Cocoa/Appkit framework, so a window in iMops is a Cocoa window, whose default appearance is a bit different from a Carbon window, that is, the default background color is gray, not white.

Cocoa window object is an encapsulated data structure like carbon window. iMops create a window object in system through a generic class method `alloc` of Cocoa window class (`NSWindow`), and keep the reference pointer in the iMops window object's data area.

`new:` of iMops `Window` class requires 3 (4 as stack items) parameters to initialize the windows attributes — the frame rectangle (the pointer of a rectangle data structure), title string (addr-len string) and window attributes flag (a predefined constant). You can use a system object `FrameRect` or `TempRect` for the rectangle parameter.

```
window ww
50 50 800 600 put: FrameRect
FrameRect " mywindow" docWindow new: ww
show: ww
```

`new:` method itself doesn't open the window. In order to open the newly created window, you need to explicitly send `show:` to the window object after `new:`.

You can draw texts or graphics directly on the window using CoreGraphics functions (a low level graphics word set to simulate QuickDraw by CoreGraphics are in preparation). But you should normally use view system on a window for high functionalities. `Window+` class is for a window with the content view. `new:` method of `Window+` takes one more paramater which is a Mops view object to be the window's first content view.

```
window WW+
View VV

50 50 800 600 put: FrameRect
FrameRect " mywindow+" docWindow VV new: WW+
show: WW+
```

As for a view structure, see the description in the Chapter for `View` class.

4.0.1 Local Coordinate and Global Coordinate

As you may know, in the coordinate system of Cocoa, the origin is set at the LEFT-BOTTOM corner, and X-axis grows rightward, Y-axis upward. Local coordinate, that is, the coordinate within a window, in iMops is Cocoa like one. So (0,0) corresponds to the LEFT-BOTTOM corner of the window's content area, and Y-axis grows upward in iMops.

However, as far as the global coordinate for positioning of a window on the main screen concerned, iMops internally recalculates them to be a QuickDraw like coordinate system. That is, in the global coordinate, (0,0) corresponds to the LEFT-TOP of the main screen, and Y-axis grows downward. This is because we normally see the LEFT-TOP corner to be the base for global positioning since a window grows RIGHT-BOTTOM-ward. The origin of the window frame rectangle corresponds to the position of the LEFT-TOP corner of the window's content area (NOT including the title bar) in the global coordinate system.

4.0.2 Window Close Handling

Before closing a window, some special actions may be needed in some situations. For example, you may want to open a dialog box for an alert when a user are closing the window on which (s)he is editing a document without save.

For that purpose, `Window` class has methods `?close:` and `close?:` . When you send `?close:` to a window object, the method calls a `FORWARD`-defined word, `WindowCloseHandler`. Then, the word, `WindowCloseHandler`, sends a `close?:` message through dynamic binding to the window object that got the `?close:` message. The method `close?:` should return a boolean value `true/false`. If `close?:` returns `true`, `WindowCloseHandler` disposes the Cocoa window object and clear all data in the window. If `close?:` returns `false`, `WindowCloseHandler` does nothing so that the window is kept opened.

`close?:` method of `Window` class simply returns `true`, so `?close:` is equivalent to `close:` in `Window` class. But you can override `close?:` method in a subclass. `close?:` method in a subclass would typically display a dialog box if in need, and returns `true` (when when the window should be closed) or `false` (when the window closing process should be aborted) corresponding to the user action.

`?close:` will be ordinarily called from a menu item. While, `WindowCloseHandler` is called also when the (red) close button on the window is clicked, so that coherent behaviors can be implemented.

4.1 Fundamental Window Classes

4.1.1 Window

`Window` defines methods for basic window creation/manipulations. It is subclassed by `Window+` class, which includes view structures on a window.

Superclass	`Object` (1.1.1)
Source file	WindowClass
Status	Core

Instance variables	`ZVar WinObj`
	`ZVar ObjCClass`
	`ZVar WINDELEGATE`
	`CGRect MYFRAME`
	`ZVAR TITLESTR`
	`X-ADDR DRAWHANDLER`
	`Bool alive`
	`UByte myStyle`

Indexed data	None
System object	None

Methods

accessing

`getSize:`	`( f: -- wid hi )`	Returns the window content size in **Floating Point** numbers.
`getFrame:`	`( f: -- x0 y0 wid hi )`	Returns the window's frame rectangle in floating point numbers.
`alive?:`	`( -- b )`	Returns `true` when the window already got `new:` and keeps a Cocoa system window object, `false` otherwise.
`getWinObj:`	`( -- ^win )`	Returns the base address of the Cocoa window object created for the iMops window object. Useful to send an objective-C message to the system window object.
`getTitleNSString:`	`( -- ^str )`	Returns the title string of the window as an `NSString` object. NSString is equivalent to CFString.
`setMinSize:`	`( wid hi -- )`	Sets the minimum size of the window.
`setMaxSize:`	`( wid hi -- )`	Sets the maximum size of the window.

manipulation

`setSize:`	`( wid hi -- )`	Sets the size of the window's content rectangle to (wid,hi).
`move:`	`( x y -- )`	Moves the left top corner of the content rect of the window to (x, y) in global coordinate.
`show:`	`( -- )`	Makes a `new:`ed window appear and moves it to front.
`setTitle:`	`( addr len -- )`	Sets the passed-in counted string to be the title of the window.
`setTitleNSString:`	`( ^str -- )`	Sets the passed-in NSString to be the title of the window. NSString is equivalent to CFString.
`setTitleFPath:`	`( ^str -- )`	Sets the passed in NSString whose contents is the file path to be the title of the window. The title will be the file name with the small icon.

`setAlpha:`	`( f: a -- )`	Sets the window's alpha value (transparency) from the passed-in FP value whose range is from 0.0 (transparent) to 1.0 (default).

creation and disposition

`new:`	`( ^rect tadd tlen styl -- )`	Creates a Cocoa Window for this window object. The window size is set from the passed in rectangle structure. `taddr tlen` string will be the window title. `styl` is the window's style flag. Constants, `docWindow`, `NoCloseStyle` and `notResizable` are defined for that.
`close:`	`( -- )`	Closes the window and disposes the window object.
`release:`	`( -- )`	Synonym of `close:`.
`clear:`	`( -- )`	Clears the object data for re-`new:` after closing the window without sending `close:` message.
`?close:`	`( -- )`	Calls `WindowCloseHandler`. Then, releases the Cocoa window object and its delegate object if `WindowCloseHandler` returns non-zero. Does nothing otherwise.
`close?:`	`( -- true )`	Called from `WindowCloseHandler`. This method will be typically overridden in a subclass. When this method returns `true`, `WindowCloseHandler` sends `clear:` to the window object, then returns 1, which will cause the disposition of the Cocoa window object. When this method returns `false`, `WindowCloseHandler` simply returns 0, which will cause to keep the window object alive.

testing

`test:`	`( -- )`	Creates and Opens a small (200×100) test window.

4.1.2 Window+

`Window+` class supports a window with a view structure on it. Since most window will have text views, image views, buttons, sliders or other GUI elements on it, `Window+` class will be the standard class for a normal window.

Superclass	`Window` (4.1.1)
Source file	Window+
Status	Core
Instance variables	`ZVar ContView`
Indexed data	None
System object	LocateWindow, __Install-Dialog

Methods

creation and disposition

`new:`	`( ^rect taddr tlen styl ^view -- )`	Creates a Cocoa Window for this window object. The top (the last) parameter is the object base address of a view object which will be the content view of the window. As for a view object, see the description of `View` class.
`close:`	`( -- )`	Closes the window and disposes the window object and view objects on it.
`release:`	`( -- )`	Synonym of `close:`.
`clear:`	`( -- )`	Releases the view system, if any, on the window and clears the window's object data area.

drawing

`draw:`	`( -- )`	Sends `draw:` to views and flushes the window buffer.

testing

`textViewTest:`	`( -- )`	Opens a window with a text view on it. This method is only for a test.

Chapter 5

View

Introduction

The concept `View` includes many GUI elements.

In iMops, the base class is `View`, which is the super class of classes `TextView`, `ImageView`, `ScrollView` and all controller classes. While controller classes in Cocoa are not subclasses of the View class, they are proper subclasses in iMops class system, because controllers occupy the definite areas of their own on some view and, in that meaning, they are quite similar to subviews.

You can set subviews to the parent view by `addview:` method. Relative positioning is not supported yet. The initial position of a subview is set by sending `setframe:` message to the subview. The origin of the local coordinate of a view is the **left-bottom** corner. When you set the origin of the frame rectangle of a subview, for example, to be (20,30), the subview's left-bottom corner will be put at (20,30) in the next parent view's local coordinate. Subviews are resizable according to the resizing of the parent view by default.

Concerning the structure of Rectangle, see the description of `CGRect` Class(1.4.2).

5.1 View

5.2 TextView

5.3 ScrollView

5.4 ImageView

Chapter 6

Control

6.1 TextField

6.2 Button

6.3 IconButton

6.4 CheckBox

6.5 Slider

Chapter 7

Menu

7.1 Menu