F-Script Guide

Documentation for the F-Script language

CONTENTS

1 INTRODUCTION

1.1 What is F-Script?

F-Script is a lightweight object-oriented scripting and interactive environment.

The main goals of F-Script are:

- To provide an interactive and scripting layer for the Cocoa environment.

- To introduce and implement new and original conceptual advances at the language level. This includes a new high-level message-sending paradigm and a new set of operators for object manipulation.

- To provide an extensible object-oriented user interface architecture in which the base elements are objects that can be interactively manipulated and combined by the user to complete tasks.

You don't program new classes with the F-Script language; instead you still use Objective-C. F-Script will dynamically load these new classes and let you manipulate them.

The F-Script user interface is said to be object-oriented because the means provided for manipulating objects support polymorphism. The only generic way of manipulating an object using F-Script is by sending it messages. F-Script provides you with sophisticated tools (interpreter and object browsers) for sending messages to objects. Because objects are defined by the set of messages to which they respond, F-Script is a generic tool you can use on whatever object you want.

1.2 How to use this guide

It would be wise to read this documentation in front of an F-Script interpreter, and to interactively experiment and explore the F-Script concepts this text introduces.

Special Note from the author

The F-Script language shares some features with Smalltalk, Self and APL. Some parts of this document are directly adapted from the wonderful *APL 360 Primer* by Paul Berry, IBM Corporation. Some parts are also directly inspired by:

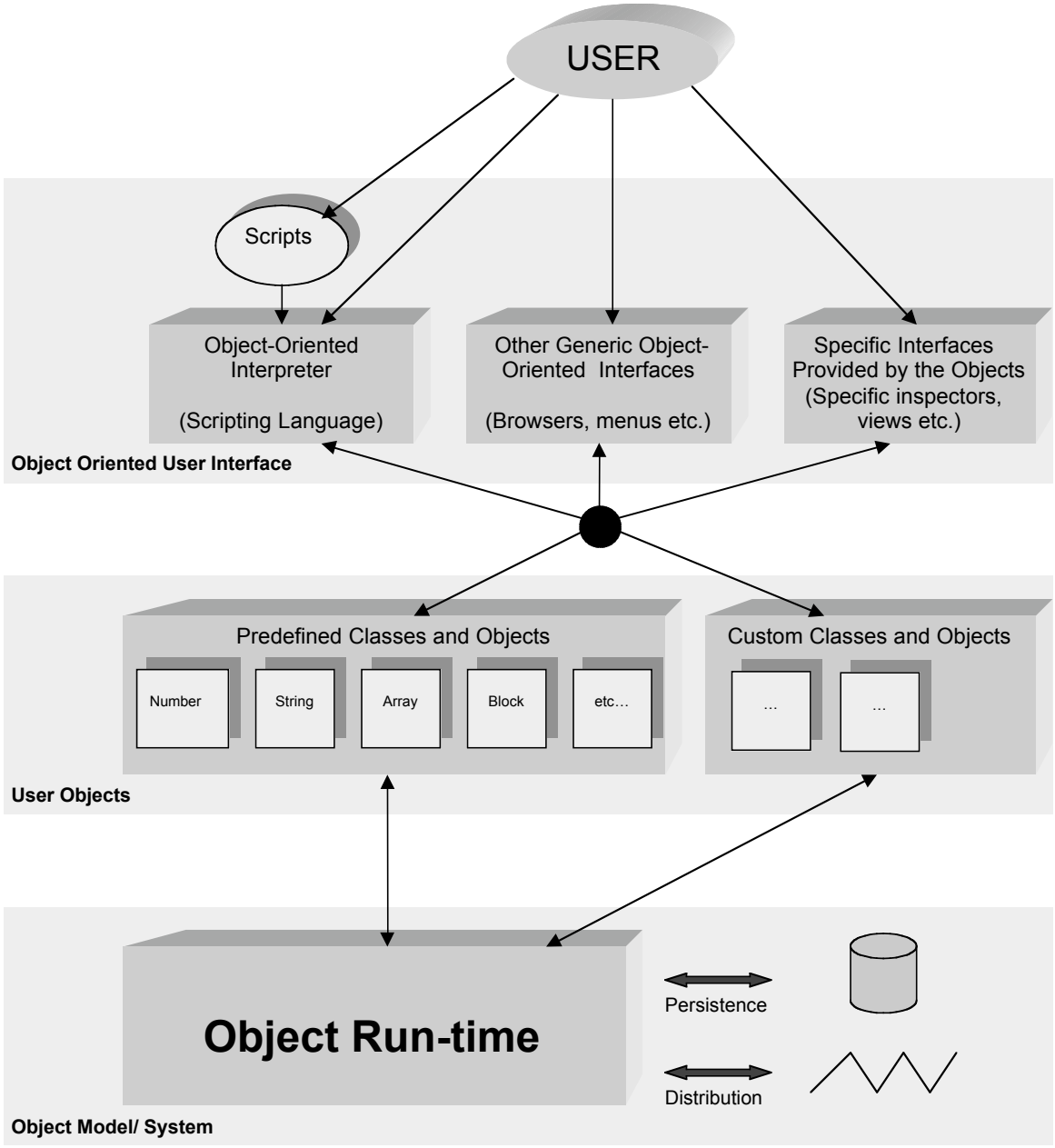
- *Inside Smalltalk*, Volume 1. Wif R. Lalonde / John R. Pugh. Prentice Hall

- *The SELF 4.0 Programmer's Reference Manual*, Agesen / Bak / Chambers / Chang / Hölzle / Maloney / Smith / Ungar / Wolczko. Sun Microsystem

- NCITS J20 DRAFT of ANSI Smalltalk Standard revision 1.9

Some examples are inspired by *Les APL étendus,* Bernard Legrand. Masson.

2 Software Architecture: The F-Script Solution



3 Using the F-Script Environment

The standard F-Script package contains an application called “F-Script.app”. This application is an interactive interface with the F-Script interpreter. It includes a command line interface and a graphical object browser.

The command line interface displays a window, which prompts you to enter commands. After you have entered a command, you hit the “Return “ key to execute it. The window then displays the result of your command. For example, try typing “3+4” and hit the Return key. You should see this:

```
> 3 + 4
7
```

The user interface keeps a history of the command you entered. In addition, some special keystrokes allow you to control the interface:

Key combination	Action
[Control +] up arrow	Get previous command in history
[Control +] down arrow	Get next command in history
Control + right arrow	Go to end of current command
Control + left arrow	Go to beginning of current command
Control + BackSpace	Delete the current command
Control + Return	New line
Control + slash	Go to next argument placeholder
F5 or option + Escape	Code completion
F7	Switch the paste parsing modes (i.e. determine if is a new line is considered as a command separator when pasting commands)
F8	Parenthesizing

The completion mechanism is aware of class names, Cocoa constants and method selectors (in a standard configuration, this represents about 40,000 unique symbols). Class names and method selectors are automatically added to this list when new bundles are loaded.

To launch the graphical object browser, select “Open Object Browser” in the Workspace menu.

4 BASIC CONCEPTS

4.1 Literal notations

A literal expression, when evaluated, produces an object.
Examples of literal expressions:

`1`	is an instance of the **NSNumber** class (or one of its subclasses)
`-3.14`	is an instance of the **NSNumber** class (or one of its subclasses)
`1.23e-2`	is an instance of the **NSNumber** class (or one of its subclasses)
`'hello'`	is an instance of the **NSString** class (or one of its subclasses)
`true`	is an instance of the **True** class
`false`	is an instance of the **False** class
`{1,2,'bill'}`	is an instance of the **Array** class (a subclass of NSMutableArray)
`[:a :b| a + b]`	is an instance of the **Block** class
`#+`	is an instance of the **Block** class in compact form
`nil`	the nil object

4.2 Sending Messages

In F-Script, you manipulate objects by sending them messages (i.e. invoking methods). Message expressions in F-Script are similar to those in Smalltalk and describe the receiver of the message, the operation being selected, and any arguments required to carry out the requested operation. The components of the message expression are called the receiver, the selector, and the arguments respectively.

Examples of message sending:

`5 sin`	The unary message "`sin`" is sent to a number object with a value of 5 (the receiver). A number object with value of -0.95892427466313845 is returned.
`2+4`	The binary message "`+`" is sent to a number object with a value of 2 with one argument, a number object with a value of 4. A number object with a value of 6 is returned.
`35 between:0 and:100`	The keyword message "`between:and:`" is sent to a number object with a value of 35, with two number objects as arguments whose values are 0 and 100. The FSBoolean object "`true`" is returned.

As shown in the above example, F-Script supports three primitive types of messages, known as **unary**, **binary** and **keyword** messages:

- Unary messages have no arguments, only a receiver and a selector.

- Binary messages, in addition to the receiver have a single argument. Their selector is formed by a list of special characters. A binary message selector is a combination of the following characters: `+ - * / = > < ~ ? % ! & | \` . Sometimes a binary selector is called an operator.

- Keyword messages contain one or more keywords, with each keyword having a single argument associated with it. The names of the keywords always end in a colon (`:`). The colon is part of the name – it is not a special terminator.

Invoking a method on an object that does not implement it will stop the execution of your instruction and yield an error. Here is an example:

```
> 3 unrealMethod

error: an instance of Number does not respond to "unrealMethod"
```

4.3 Priority and parentheses

The receiver or argument of a message expression may itself be a message expression. This gives rise to complex message expression and the need for an evaluation order. For example, the following message expression contains unary, binary and keyword messages.

```
4 sin between:2*3 and:100
```

Many languages, C included, base the evaluation of expressions on priorities assigned to different operators. For instance, multiplication (`*`) is usually assigned a higher priority than addition (`+`). F-Script's evaluation rules (which are the same in Smalltalk) however, are based on the type of message (unary, binary, and keyword) involved in the expression. In order of application, the evaluation order is as follows:

1) **Parenthesized** expressions

2) **Unary** expression (evaluated from left to right)

3) **Binary** expressions (evaluated from left to right)

4) **Keyword** expression

Note: all binary operators have the same priority level.

Fully parenthesizing a message expression removes all ambiguity about the evaluation order. Each of the following examples is shown with its fully parenthesized form to illustrate the order of evaluation.

Expression	***Fully Parenthesized Expression***
`2 sin negated`	`(2 sin) negated`
`3 + 4 * 6 + 3`	`((3 + 4) * 6) + 3`
`15 max: 32 / 3`	`15 max: (32 / 3)`
`2 sin + 4`	`(2 sin) + 4`
`5 between: 1 and: 3 sin + 4`	`5 between: 1 and: ((3sin) + 4)`
`4 sin max: 4 * 6`	`(4 sin) max: (4*6)`

4.4 Variables and a study of an expression evaluation

Variable names in F-Script are simple identifiers consisting of a sequence of letters, digits and underscores (_), beginning with a letter or an underscore. All variables are object references. The assignment expression is denoted by the "**:=**" construct. The expression "**x := 4**" binds a number object with a value of **4** to the target variable. Note that the value of an assignment expression is the value that is assigned to its target variable The subsequent expression "**x**" will evaluate to that number. The subsequent instruction "**x := x + 1**" is evaluated as follows:

1) The sub-expression "**x**" (from the right part of the assignment) evaluates and returns the number object with a value of **4** that was bound to **x**

2) The sub-expression "**1**" evaluates: this generates and returns a number object with a value of **1.**

3) The binary method "**+**" is invoked on the number object returned by step 1 with the number object returned by step 2 as the argument. The class representing numbers is an Objective-C class, so the method "**+**" is implemented in Objective-C (Objective-C does not permit symbols like "**+**" to be used as method names, so F-Script automatically performs mapping and in fact invokes the method "**operator_plus:**"). This method generates and returns a number object with a value of **5.**

4) The assignment is made: **x** is bound to the number object returned by step 3

5) The number object returned by step 3 is returned as the value of the assignment expression.

You can assign an object to almost any name you like. But if you attempt to display or make use of a variable before any object has been assigned to it, the interpreter will be unable to supply an associated object, and won't proceed with the execution of your instruction. It reports the trouble by yielding an error with a message in the form of "`error: undefined identifier xxx`".

To get a list of user-defined identifiers in your workspace, send the `identifiers` message to a predefined object named `sys`.

4.5 Expression list

Multiple expressions can be strung together by using the "**.**" separator. This creates an expression list. An expression list evaluates to the result of the evaluation of the last expression in the list. For example: "`x := 3. y:= 5. x+y`" evaluates to `8`. The "." symbol is also referred as the "instruction separator".

4.6 Comments

Comments are delimited by double quotes:

```
"this is a comment"
```

4.7 Cascades

A *cascade* is a sequence of message sends that are all directed to the same object. Only the first in such a sequence has an explicit receiver specified. The receiver of the subsequent messages is the same object as the receiver of the initial message in the sequence. Otherwise, each message send occurs as if it was a normal message send that was not part of a cascade. The result object of each message in the cascade except the right-most message is discarded. The result of a cascade is the value of its right-most message. Messages that form the cascade are separated by ";".

Example:

```
> myArray := {}

> myArray add:99; add:100; add:101; count
3

> myArray
{99,100,101}
```

4.8 Allocation and Deallocation of objects

F-Script follows the Cocoa reference-counting scheme. Note that when you assign an object to a variable, F-Script automatically retains it, releasing it when you re-assign another object to the variable.

5 Direct access to Cocoa

F-Script gives you direct access to Cocoa APIs.

5.1 Example: Drawing

In this example, we use the Cocoa graphic API to draw a blue circle.

Set the current color:

```
> NSColor blueColor set
```

Draw a circle:

```
> (NSBezierPath bezierPathWithOvalInRect:(500<>300 extent:100<>100)) stroke
```

5.2 Example: Playing a sound

In this example, we use the Cocoa NSSound class to play a sound:

```
> (NSSound soundNamed:'Submarine') play
```

5.3 Example: Opening a Window

Creates a NSWindow:

```
> w := NSWindow alloc initWithContentRect:(100<>100 extent:300<>300)
styleMask:NSTitledWindowMask+NSClosableWindowMask backing:NSBackingStoreBuffered  defer:false
```

Now, to put it on screen:

```
> w orderFront:nil
```

Setting its title is easy:

```
> w setTitle:'this is a nice title'
```

Playing with transparency:

```
> w setAlphaValue:0.5
```

5.4 Example: A little graphic animation

In this example, we use the Cocoa graphical API to produce a little graphical animation:

```
keyWindow := NSApplication sharedApplication keyWindow.
NSBezierPath setDefaultLineWidth:20.
keyWindow contentView lockFocus.

1 to:550 by:4 do:
[:x|
    path := NSBezierPath bezierPathWithOvalInRect:(x<>130 extent:200-(x/3)<>(x/2)).
    (NSColor colorWithDeviceRed:x/570 green:0.1 blue:1-(x/570) alpha:1) set.
    path stroke.
    keyWindow flushWindow.
    NSColor whiteColor set.
    path setLineWidth:path lineWidth + 2.
    path stroke.
].

keyWindow contentView unlockFocus.
keyWindow display.
```

6 Graphical Object Browser

F-Script provides a powerful tool to graphically interact with objects. The F-Script object browser can be opened using the `browse` and `browse:` methods of the System class (a special instance of System, representing the current F-Script interpreter, is always defined. Its name is “sys”). For instance, to open an object browser on the number 99, you will type: `sys browse:99`

From the F-Script application, you can also open the object browser by selecting “Open Object Browser” in the Workspace menu.

7 Introduction to Basic Data Types

This section introduces some features of the basic F-Script data types.

7.1 Numbers

Numbers in F-Script are standards NSNumber objects. FSNSNumber, a category of NSNumber, provides classical mathematical operations (**+**, **-**, *****, **/**) and comparison functions (**>**, **>=**, **<**, **<=**). Equality is tested with the **=** method, inequality with the **~=** method. Also provided are some mathematical functions, such as **abs**, **sin**, **cos**, **ln**, **sqrt** etc. Methods provided by F-Script for dealing with numbers work in double precision.

You can get the larger of two numbers with the `max:` method, and the smaller with the `min:` method. For example:

```
> 5 max:100
100

> 5 min:100
5
```

You can get the **floor** and **ceiling** of a number:

```
> 5.7 ceiling
6

> 5.7 floor
5
```

7.2 Strings

Strings in F-Script are standards NSString and NSMutableString objects. In addition to the Cocoa string methods, F-Script adds support for the classical comparison operators and several manipulation functions. These methods are defined in categories of NSString and NSMutableString: FSNSString and FSNSMutableString.

Example:

```
> 'aaa' < 'abc'                          “comparing“
true

> 'oliver' at:2                          “indexing“
'i'

> 'oliver' length                        “getting the length“
6

> 'oliver' reverse                       “reversing“
'revilo'

> 'Dear ' ++ 'oliver'                    “concatenating“
'Dear oliver'
```

The literal notation returns an immutable NSString.

7.3 Arrays

Arrays in F-Script are NSArray and NSMutableArray objects. Array objects are very important in F-Script because they support several advanced data manipulation features. These features are described in their own section of this guide. FSNSArray, a category of NSArray, and FSNSMutableArray, a category of NSMutableArray, add a number of methods for array manipulation.

The literal notation returns a mutable array.

Basic features of arrays include indexing (start with 0), inserting elements and removing elements. For example:

```
> {1,2,3,'oliver',5*5}              “evaluating an array literal“
{1, 2, 3, 'oliver', 25}

> {1,2,3,4} count                   “getting the size“
4

> {2,4,6,8} at:2                    “indexing“
6

> {2,4,6,8} at:4                    “invalid indexing“
error: index of an array must be a number less than the size of the array

> myArray := {1,2,3,4}
> myArray insert:'hello' at:2       “inserting“
> myArray
{1, 2, 'hello', 3, 4}

> myArray removeAt:1                “removing“
> myArray
{1, 'hello', 3, 4}

> myArray at:0 put:100              “replacing“
> myArray
{100, 'hello', 3, 4}

> {{1,2,3}, {10,11}}                “an array of array“
{{1,2,3}, {10,11}}

> {1,2,3} ++ {10,20}                “concatenating“
{1, 2, 3, 10, 20}
```

Note: Since arrays are represented by NSArray and NSMutableArray, you can also use the standard Cocoa methods for manipulating arrays in F-Script. However, as you will learn in this guide, F-Script is an array language and the methods shown in the examples above provide additional power when dealing with arrays. For instance, the `at:` method, shown above for indexing an array, lets you specify a whole array of indices instead of just one integer.

7.4 Booleans

Boolean objects (that you can enter as `true`, `false`, `YES` and `NO`) are represented by the class FSBoolean. They provides the classical Boolean operations: the AND operation is provided by the binary method `&`, the OR operation by the binary method `|`, the NOT operation by the unary method `not`. For example:

```
> true & false
false

> true | false
true

> false not
true
```

Booleans, along with Blocks also support control structure with methods `ifTrue:` and `ifTrue:ifFalse:`

"true" and "YES" both evaluate to the true value.
"false" and "NO" both evaluate to the false value.

7.5 Blocks

A Block is an object that contains some F-Script code. Generally, you create a block by bracketing a segment of code. For example: `[x := x +1]` is a block. You can execute the code inside this block by sending it a `value` message. For example:

```
> x := 1                    "assigning 1 to x"
> x                         "evaluating x"
1                           "x is 1"
> myBlock := [x := x + 1]   "assigning the block [x := x + 1] to myBlock"
> x                         "evaluating x"
1                           "x is the same"
> myBlock value             "executing the block"
> x                         "evaluating x"
2                           "x has changed"
```

A Block may have parameters. For example `[:a :b | a + b]` is a block with two parameters named `a` and `b`. When executing it you have to supply the value for the arguments, using the relevant `value…` message. For example:

```
> myBlock := [:a :b| a + b]
> myBlock value:3 value:4
7
```

Blocks may also have local variables and bindings to other objects. They can be executed recursively. Below is an example of a block with two arguments and one local variable:

```
> [:a :b| |local| local := a + b. local * 2] value:3 value:4
14
```

A block that just sends a single message to its first argument with its other arguments as arguments to the message, may be represented using the compact notation for block literal. This notation represents a block by a "`#`" immediately followed by the message selector the block has to use when it is executed. For example, the block `[:a :b| a + b]` may also be represented in a compact form with `#+` :

```
> #+ value:3 value:4
7
```

Along with Booleans, Blocks provide support for control structures with the `whileTrue .` method.

A Block can be edited in a specific inspector by sending it the `inspect` message.

7.6 Ranges, points, rectangles and sizes

A number of Cocoa APIs take or return values of types NSRange, NSPoint, NSRect and NSSize. Because these types are defined as C structs, they can't be manipulated directly from F-Script, which deals only with objects. However, Cocoa's NSValue class provides support for wrapping values of these types into objects. F-Script leverages this support by automatically wrapping and unwrapping values of these types when invoking methods. This let you invoke methods taking or returning NSRange, NSPoint, NSRect or NSSize values from F-Script.

F-Script also adds a few convenience methods to NSValue (with the FSNSValue category) and to NSNumber (with the FSNSNumber category) to deal with ranges, points, rectangles and sizes.

7.6.1 Ranges

From F-Script, you can create an NSValue containing an NSRange by invoking the class method `rangeWithLocation:length:` defined on NSValue by the FSNSValue category. You can ask a range for its `location` and `length`. For example:

```
> myRange := NSValue rangeWithLocation:10 length:5   "generating and assigning a range to
                                                      the variable myRange"

> myRange                                            "evaluating myRange "
(Range location=10 length=5)

> myRange location                                   "asking for the location"
10

> myRange length                                     "asking for the length"
5

> myRange = (NSValue rangeWithLocation:10 length:5)  "comparing myRange with another
range."
true
```

7.6.2 Points

From F-Script, you can create an NSValue containing an NSPoint by invoking the `<>` method on a number, with another number for argument. This method generates and returns a point whose x-coordinate is equal to the value of the receiver and whose y-coordinate is equal to the value of the argument. You can ask a point for its x and y coordinates with the messages `x` and `y`. For example:

```
> myPoint := 100<>150       "generating and assigning a point to the variable myPoint"

> myPoint                   "evaluating myPoint"
(100<>150)

> myPoint x                 "asking for the x coordinate"
100

> myPoint y                 "asking for the y coordinate"
150

> myPoint = (100<>150)      "comparing myPoint with another point. Parenthesis must be
                             used, otherwise the expression will be interpreted as
                            (myPoint = 100) <> 150."
true
```

Note: In Smalltalk, points are traditionally created using the `@` operator instead of the `<>` operator (that is, you type `100@150` instead of `100<>150`). However, in F-Script, the `@` symbol is used for specifying message patterns and can't be used as an operator.

7.6.3 Rectangles

From F-Script, you can create an NSValue containing an NSRect by invoking the `extent:` or the `corner:` method on an NSValue containing an NSPoint, with another point as argument. The `extent:` method generates and returns a rectangle whose origin is the receiver and whose width and height are provided by the argument. The `corner:` method generates and returns a rectangle whose origin is the receiver and whose opposite corner is the argument. You can ask a rectangle for its `extent` (a point defined by the width and the height of the rectangle), its `origin` (a point representing the origin of the rectangle) and its `corner` (a point representing the corner at the opposite of the origin). For example:

```
> myRect := 100<>150 extent:20<>20          "generating and assigning a rectangle to the
                                            variable myRect"

> myRect2 := 100<>150 corner:120<>170       "generating and assigning a rectangle to the
                                            variable myRect2"

> myRect                                    "evaluating myRect"
(100<>150 extent:20<>20)

> myRect extent                             "asking for the extent"
(20<>20)

> myRect corner                             "asking for the corner"
(120<>170)

> myRect origin                             "asking for the origin"
(100<>150)

> myRect = myRect2                          "comparing myRect with myRect2"
true
```

7.6.4 Sizes

From F-Script, you can create an NSValue containing an NSSize by invoking the class method `sizeWithWidth:height:` defined on NSValue by the FSNSValue category. You can ask a size for its `width` and `height`. For example:

```
> mySize := NSValue sizeWithWidth:20 height:30      "generating and assigning a size to
                                                    the variable mySize"

> mySize                                            "evaluating mySize"
(Size width=20 height=30)

> mySize width                                      "asking for the width"
20

> mySize height                                     "asking for the height"
30

> mySize = (NSValue sizeWithWidth:20 height:30)     "comparing mySize with another size"
true
```

8 Control Structures

Unlike many languages, no additional syntactic structures need to be added to describe the conventional conditional and repetitive control structures. In F-Script, these control structures are implemented in terms of objects and message passing. In particular, block and Boolean objects provide the required support. Note that F-Script also provides some higher-level control structures than those presented in this section; in fact, explicit *if* and *while* type structures are much less used in F-Script than in other languages.

8.1 Conditional

The conditional selection is expressed by sending the message `ifTrue:ifFalse:` to a Boolean with, typically, two blocks as argument. The Boolean will respond to the message by evaluating the appropriate argument block. If the Boolean is `true`, it will evaluate the first block and ignore the second. On the other hand if the Boolean is `false`, it will evaluate the second block and ignore the first. Finally, the result of the evaluation of the selected block is returned.

Below is a comparison of F-Script code and C code for the conditional:

F-Script

```
number1 < number2 ifTrue:
[
    maximum := number2.
    minimum := number1.
]
ifFalse:
[
    maximum := number1.
    minimum := number2.
]
```

C

```
if (number1 < number2)
{
    maximum = number2;
    minimum = number1;
}
else
{
    maximum = number1;
    minimum = number2;
}
```

Boolean objects `true` and `false` also accept the single keyword message `ifTrue:`. The Boolean `true` responds to this message by returning the value of the block argument; `false` responds by returning the special object `nil`. There are also methods `ifFalse:` and `ifFalse:ifTrue:`

8.2 Loop

F-Script provides a conditional repetition equivalent to the C **while** statement. It is again based on blocks and makes use of the fact that blocks are objects and thus can support their own message protocol. Consider the following program fragments to compute the sum of the first 100 integers:

F-Script

```
sum := 0.
number := 1.
[number <= 100] whileTrue:
[
    sum := sum + number.
    number := number + 1.
]
```

C

```
sum = 0;
number = 1;
while (number <= 100)
{
    sum = sum + number;
    number = number + 1;
}
```

The block that receives the `whileTrue:` message repeatedly evaluates itself and, if the termination condition is not yet met, evaluates the argument block. The `whileTrue` (without argument), `whileFalse` and `whileFalse:` methods also exist.

F-Script also provide something like the **for** statement in the form of the “to:do:” and “to:by:do:” methods provided by the FSNSNumber category (a category of NSNumber).

F-Script

```
sum := 0.
1 to:100 do:
[:i|
    sum := sum + i.
]
```

C

```
sum = 0;
for (i = 1; i <= 100; i++)
{
    sum = sum + i;
}
```

9 Persistence

Objects can be saved in files using the `save:` or `save` messages:

```
> myObject save:'myFile'        “save myObject to the specified file“

> myObject save                 “open a file browser and save to the chosen file“
```

You load an object by sending the `load:` or `load` message to a predefined object named `sys`:

```
> myObject := sys load:'myFile'   "load the object stored in the file"

> myObject := sys load            "open a file browser and load the chosen file"
```

You can save and load your entire workspace by sending `saveSpace:`, `saveSpace`, `loadSpace:` or `loadSpace` to `sys`.

F-Script uses NSArchiver to save objects. So, objects to be saved must conform to the standard Cocoa NSCoding protocol. Files generated are standard NSArchiver files; you can use them with other applications to load archived objects using the Cocoa NSUnarchiver API.

10 Distributed Objects

Cocoa integrates a distributed object system. As with any other Cocoa APIs, you can use it directly from F-Script. In addition, F-Script provides two convenient methods to access this system. The first method, `vend:`, registers its receiver in the distributed object system using the name passed as argument. The second method, `connect`, which should be invoked on a string, returns a proxy to a registered distributed object.

Vending an object:

```
> myObject vend:'foo'          "vend myObject, using 'foo' as a public name“
```

Connecting to a distributed object:

```
> myProxy = 'foo' connect      "return a proxy to the distributed object registered
                                under the name 'foo'"
```

Note that `connect` only searches for distributed objects on the local host. Use the standard Cocoa API if you want to connect to a distributed object on a different host.

Using distributed objects, applications can provide scripting capabilities by vending objects. By connecting to such distributed objects from F-Script you can remotely send commands to these applications from F-Script.

11 Custom Classes Integration

F-Script can automatically integrate your custom classes. All you have to do is to put a bundle containing your classes in the F-Script repository. This repository is created by the F-Script.app application when you launch it for the first time. By default it will be "~/Library/Application Support/F-Script" (where ~ stands for your home directory). You put your bundles (which can be whole frameworks) in the "~/Library/Application Support/F-Script/classes" directory.

At launch time, an F-Script interpreter automatically links up to these bundles. You can put as many bundle as you want in the class repository.

Of course, can also use the standard Cocoa methods to dynamically load classes. For instance, the following F-Script instruction will dynamically load the GLUT framework (an OpenGL utility toolkit) provided by Apple:

```
(NSBundle bundleWithPath:'/System/Library/Frameworks/GLUT.framework') load
```

12 Object equality and identity

F-Script uses the Cocoa concept for equality and identity. Equality is defined in Cocoa by the `isEqual:` method. For convenience, F-Script provides the `=` and `~=` methods to test the objects for equality and inequality. These methods use the standard Cocoa `isEqual:` method to determine equality.

Identity can be tested with the `==` and `~~` methods. The `==` method only responds true if the receiver and the argument are the same object in memory.

Since these methods are defined in FSNSObject, a category of NSObject, they can be used with all objects. Note, however, that FSNSArray redefines `=` and `~=` to give them a different meaning.

13 Object duplication

Some objects support a duplication service.

The `clone` message returns an autoreleased copy of the receiver. In general, this is just a cover method that calls `copy` or `mutableCopy`, auto-releases the copy and returns it.

The `setValue:` message sets the value of the receiver to that of the argument. Usually, the receiver and the argument are to be of the same class.

Most of the built-in F-Script classes implement `clone` and `setValue:`.

14 Exceptions

F-Script uses the Objective-C exception model, and let you throw and handle exceptions. To define an F-Script exception handler, you use the `onException:` method of class Block. This method takes a block as argument (the exception handler). The method evaluates the receiver and, if an exception is raised across the receiver, the handler is evaluated. If the handler declares that is has an argument, then it is given the actual exception as the argument.

General notation:

```
[block that may raise an exception ] onException: [:exception| exception handler code ]
```

Example:

```
[1/0 ] onException:[:e| sys beep. sys log:e ] " Will play a beep and log the exception
                                                thrown by the division by zero "
```

To explicitly throw an exception, you can use the NSException API. Example:

```
(NSException exceptionWithName:'MyException' reason:'testing' userInfo:nil) raise
```

Since Mac OS X 10.3, it is possible to throw any Objective-C object as an exception object. In Objective-C the syntax is:

```
@throw object
```

In F-Script, the same is done by sending the "throw" message to the object you want to throw:

```
object throw
```

Note: the "throw" method is implemented in FSNSObject, a category of NSObject.

15 Objective-C mapping

F-Script is based on the Objective-C object model. However, in some situations, mapping must occur between F-Script and Objective-C.

15.1 Operators

Objective-C does not support non-alphabetical symbols as method selectors. When such a selector is used in F-Script, mapping takes place in order to produce an alphabetical selector. Each non-alphabetical symbol that can be used to form a selector name has an alphabetical name, given in the following table. This alphabetical name is used to produce the alphabetical Objective-C selector.

Symbol	Alphabetical Name
+	plus
-	hyphen
<	less
>	greater
=	equal
*	asterisk
/	slash
?	question
~	tilde
!	exclam
%	percent
&	ampersand
|	bar
\	backslash

The mapping works by appending the names of the symbols, using "_" as a separator. The Objective-C selector name begins with "`operator_`" and ends with "`:`".

Examples:

The operator	+	is mapped to	`operator_plus:`
The operator	-	is mapped to	`operator_hyphen:`
The operator	++	is mapped to	`operator_plus_plus:`

15.2 Non Object types

In F-Script, all arguments and return values of messages are objects. Objective-C, however, supports non-object types in addition to object types. To enable an Objective-C method involving non-object types to be invoked from F-Script, object arguments may be mapped to non-object arguments, and non-object return values may be mapped to object return values. This mapping occurs automatically.

Expected argument type is	You must provide a
char (this includes BOOL)	NSNumber or FSBoolean
unsigned char	NSNumber
short	NSNumber
unsigned short (this includes unichar)	NSNumber or NSString with one character
int	NSNumber
unsigned int	NSNumber
long	NSNumber
unsigned long	NSNumber
long long	NSNumber
unsigned long long	NSNumber
float	NSNumber
double	NSNumber
_Bool	FSBoolean
SEL	Block in compact form
NSPoint or CGPoint	NSValue containing an NSPoint
NSSize or CGSize	NSValue containing an NSSize
NSRect or CGRect	NSValue containing an NSRect
NSRange	NSValue containing an NSRange
pointer (e.g. int *)	FSPointer or nil (nil is mapped to NULL)

Return type is	In F-Script, you get a
char (this includes BOOL)	FSBoolean
unsigned char	NSNumber
short	NSNumber
unsigned short	NSNumber
int	NSNumber
unsigned int	NSNumber
long	NSNumber
unsigned long	NSNumber
long long	NSNumber
unsigned long long	NSNumber
float	NSNumber
double	NSNumber
_Bool	FSBoolean
SEL	Block in compact form
NSPoint or CGPoint	NSValue containing an NSPoint
NSSize or CGSize	NSValue containing an NSSize
NSRect or CGRect	NSValue containing an NSRect
NSRange	NSValue containing an NSRange
pointer (e.g. int *)	FSGenericPointer or nil (NULL is mapped to nil)
void	FSVoid

This mapping specification does not cover all Objective-C types. The Objective-C methods that use these non-covered types for an argument or their return type are not directly callable from F-Script.

15.3 The nil object

The special Objective-C value `nil` is supported in F-Script. Sending a message to `nil` from F-Script will always return `nil` except for the `==` message, which returns `true` if its argument is `nil`, and otherwise returns `false`.

15.4 Exceptions

An Objective-C exception is reported by the interpreter as an F-Script execution error (see the FSInterpreter and FSInterpreterResult classes) unless it is handled by an F-Script exception handler (see section 14).

If the interpreter is used from F-Script.app and an exception is raised and not handled, the execution of the current command is aborted, and the user is shown a message describing the exception.

15.5 Symbolic Constants

Cocoa defines a number of symbolic constants (e.g. NSNotFound). Most of them are pre-defined in F-Script and can be used directly.

16 Advanced Messaging : Message Pattern

16.1 Introduction to Message Patterns

Calculations frequently involve not just one object but a whole array of them. F-Script gets much of its power and simplicity from its approach to the processing of arrays.

With F-Script, the operations that apply to single objects can be applied with equal ease to the processing of entire arrays.

For instance, if `A` is an array of four numbers, and `B` is another array which also consists of four numbers, then the instruction `A+B` causes the computer to add the first number in `A` to the first number in `B`, and the second number in `A` to the second number in `B`, and so on. Four separate additions are performed, and so the result is also an array of four numbers.

```
> A := {1, 2.5, 7, 11}
> B := {10, 20, 30, 40}
> A+B
{11, 22.5, 37, 51}
```

What is fun is that the same sort of element-by-element parallel processing can be obtained with any kind of message!

```
> {1,2,3,4} * {1,2,3,4}
{1, 4, 9, 16}

> {1,2,3,4} max:{-10,20,0,2}
{1, 20, 3, 4}

> {1,2,3,4, 'bar'} > {-10,20,0,2,'foo'}
{true, false, true, true, false}

> {1,2,3,4,5,6} = {-1,2,-3,4,-5,6}
{false, true, false, true, false, true}

> {1,2,3,4} between:{0,1,-5,3} and:{2,2,-2,10}
{true, true, false, true}
```

What about unary messages?

```
> {1.2, 1.8, 3.2, 4} floor
{1, 1, 3, 4}

> {'oliver', 'henry', 'bertram'} uppercaseString
{'OLIVER', 'HENRY', 'BERTRAM'}
```

Not all arguments have to be arrays:

```
> {1,2,3,4} * 2
{2, 4, 6, 8}

> {1,2,3,4} between:3 and:10
{false, false, true, true}
```

How it works

This special "do-it-to-all-my-array-elements" process occurs when, from F-Script, an NSArray object is sent an unrecognized message (i.e. a message not implemented by the class of the object). In this case, the F-Script interpreter generates messages and sends them to each element of the array. If some arguments are themselves arrays, their elements are also taken one after the other.

For example, when F-Script executes `{1,2,3,4} * 2`, it generates and sends four messages: `1 * 2`, `2 * 2`, `3 * 2` and `4 * 2`.

Now, what if you want to send to each element of an array a message recognized by the array? For example, say you have an array of arrays and you want the size of each sub-array. Just sending the `count` message does not work:

```
> {{1,2,3,4}, {'oliver', 'henry'}, {10,100}} count
3
```

Here you get the size of the enclosing array, not the size of each element. To force the message to be dispatched to each element, you have to use special notation that involves typing the "`@`" symbol before the message selector:

```
> {{1,2,3,4}, {'oliver', 'henry'}, {10,100}} @ count
{4, 2, 2}
```

The "`@`" notation means that a loop is performed on the array.

But what if you have an array of arrays of arrays and you want to send the message to each sub-sub-array? In this case you just add one "`@`" to your message:

```
> { { {1,2,3,4}, {'oliver', 'henry'}, {10,100} } , { {1945,1968}, {20002} } } @@ count
{{4, 2, 2}, {2, 1}}
```

You can string as many `@` as you want. The leftmost `@` is said to be at level 1, the next `@` is said to be at level 2 and so on.

What about arguments? Is there an equivalent special notation to force loops on arguments? Yes –in the case of arguments, you have to put an `@` before the argument. For example, supposing you have two arrays of strings and you want to concatenate the first string of one array with the first of the other, the second with the second and so on. You have to explicitly loop on your two arrays or else you will just get their concatenation because the concatenation operator is the same for arrays and strings:

```
> {'General ', 'Mr. ', 'Miss ' } @ ++ @ {'Grant', 'Smith', 'Robinson'}
{'General Grant', 'Mr. Smith', 'Miss Robinson'}
```

Another example where you must use `@` on the argument:

```
> 2 max: @ {0,1,10,20}
{2, 2, 10, 20}
```

Note that when there is an `@` at a given level in a message pattern expression, any other loop *for the same level* must be explicitly stated. For example:

```
> {1,2,3} + {10,20,30}       "implicit loop on the receiver and the argument
                             (because arrays do not respond to +)"
{11, 22, 33}

> {1,2,3} @ + {10,20,30}     "explicit loop on the receiver -> no more implicit loop on
                              the argument."
error: argument of method "+" must be a number or a FSBoolean

> {1,2,3} @ + @ {10,20,30}   "explicit loop on the receiver and the argument"
{11, 22, 33}
```

Message patterns

The base element in an object-oriented program is message sending. An object-oriented program can be thought as a way to express what messages are to be sent to what objects. The `@` notation gives you the possibility to specify not just one particular sent message, but an entire set of sent messages. It defines what is called a "multi-message pattern" or simply a "message pattern". The classic message-sending paradigm then becomes a special case in the Message Pattern world of F-Script.

We can extract the structural information of a particular message pattern expression and represent it:

The pattern of the message pattern expression `{1,2,3} @ + @ {10,20,30}` is `@:@`

The pattern of the message pattern expression `2 max:@ {0,1,10,20}` is `:@`

The pattern of the message pattern expression `{{1,2,3,4},{'oliver', 'henry' },{10,100}} @ count` is `@`

As you will see in the following chapters, some patterns are frequently used and are even given standard well-known names.

Note that patterns may be implicit:

In `{1,2,3} + {10,20,30}` we have the implicit pattern `@:@`

In `{1,2,3} between:3 and:10` we have the implicit pattern `@::`

Explicit loop control structures (*while, loop…until, etc)* found in classic languages are often replaced with message patterns in F-Script.

In the following example, you can use a message pattern to apply a block to each element of an array:

```
> sum := 0
> [:elem| sum := sum + elem] value: @ {1,2,3}
> sum
6
```

Note that this is just to give an example of message pattern. There is a much better way to sum the elements of an array, as you will see in the section on the reduction operator.

In this following example, you can now use a message pattern to index an array of array. You want to get the element at index 1 of each sub-array (recall that "`at:`" is the indexing method):

```
> {{1,5,3,8}, {'Lisa', 'Omere', 'Bart', 'Marge'}, {{1,4}, {6,7,8}}} @ at:1
{5, 'Omere', {6, 7, 8}}
```

16.2 Advanced message patterns

Up until now, we have seen that by using patterns on some arrays, you can generate messages that use the first element of each array, then the second and so on. But can you combine an array element in another way? For example, supposing you have two arrays (`A` and `B`) of numbers and you want to multiply the first element of `A` with each element of `B`, then the second element of `A` with each of the elements of `B` and so on. To do this, you have to specify that you want a loop on `A` and an **inner** loop on `B`. You have to use a number after the `@` symbol to state the inner level of each loop:

```
> {1,2,3} @1 * @2 {10,100,1000,10000}
{{10,100,1000,10000}, {20,200,2000,20000}, {3,300,3000,30000}}
```

The outer-most loop specifications can be abbreviated by omitting the `1` after `@`. The above command and the following are thus equivalent:

```
> {1,2,3} @ * @2 {10,100,1000,10000}
{{10,100,1000,10000}, {20,200,2000,20000}, {3,300,3000,30000}}
```

The `@1:@2` pattern has a standard name; it is called the **outer-product**.

Some examples (recall that ++ is the concatenation operator; it works for arrays as well as for strings). The last example introduces and comments on another capability of the message pattern syntax: the multi-level message pattern (a basic example was seen in the previous section).

```
> {'a1', 'a2', 'a3' } ++ {'b1', 'b2', 'b3', 'b4' }  "No pattern"

{'a1','a2','a3','b1','b2','b3','b4'}

> {'a1','a2','a3'} @ ++ @ {'b1','b2','b3','b4'}

{'a1b1','a2b2','a3b3'}

> {'a1','a2','a3'} @1 ++ @2 {'b1','b2','b3','b4'}

{{'a1b1','a1b2','a1b3','a1b4'},
 {'a2b1','a2b2','a2b3','a2b4'},
 {'a3b1','a3b2','a3b3','a3b4'}}

> {'a1','a2','a3'} @2 ++ @1 {'b1','b2','b3','b4'}

{{'a1b1','a2b1','a3b1'},
 {'a1b2','a2b2','a3b2'},
 {'a1b3','a2b3','a3b3'},
 {'a1b4','a2b4','a3b4'}}

>  [:a :b| a ++ b] value:{'a1','a2','a3'} value:{'b1','b2','b3','b4'}  "No pattern"

{'a1','a2','a3','b1','b2','b3','b4'}
```

```
>  [:a :b| a ++ b] value:@1{'a1','a2','a3'} value:@2{'b1','b2','b3','b4'}

{{'a1b1','a1b2','a1b3','a1b4'},
 {'a2b1','a2b2','a2b3','a2b4'},
 {'a3b1','a3b2','a3b3','a3b4'}}

> {{'a11','a12','a13'},{'a21','a22','a23'}} ++ {{'b11','b12'},{'b21','b22'}}  "No pattern"

{{'a11','a12','a13'}, {'a21','a22','a23'}, {'b11','b12'}, {'b21','b22'}}

> {{'a11','a12','a13'},{'a21','a22','a23'}} @ ++ {{'b11','b12'},{'b21','b22'}}

{{'a11', 'a12', 'a13', {'b11', 'b12'}, {'b21', 'b22'}},
 {'a21', 'a22', 'a23', {'b11', 'b12'}, {'b21', 'b22'}}}

> {{'a11','a12','a13'},{'a21','a22','a23'}} @@ ++ @@ {{'b11','b12'},{'b21','b22'}}

{{'a11b11', 'a12b12'}, {'a21b21', 'a22b22'}}

> {{'a11','a12','a13'},{'a21','a22','a23'}} @1@ ++ @2@ {{'b11','b12'},{'b21','b22'}}

{{{'a11b11', 'a12b12'}, {'a11b21', 'a12b22'}},
 {{'a21b11', 'a22b12'}, {'a21b21', 'a22b22'}}}

> {{'a11','a12','a13'},{'a21','a22','a23'}} @1@2 ++ @2@1 {{'b11','b12'},{'b21','b22'}}

{{{{'a11b11', 'a12b11', 'a13b11'}, {'a11b12', 'a12b12', 'a13b12'}},
  {{'a11b21', 'a12b21', 'a13b21'}, {'a11b22', 'a12b22', 'a13b22'}}},
 {{{'a21b11', 'a22b11', 'a23b11'}, {'a21b12', 'a22b12', 'a23b12'}},
  {{'a21b21', 'a22b21', 'a23b21'}, {'a21b22', 'a22b22', 'a23b22'}}}}
```

Let's analyze the last example:

The pattern here is **`@1@2:@2@1`**.

This pattern has the form **ij:kl** where **i** is **`@1`**,**j** is **`@2`**,**k** is **`@2`** and **l** is **`@1`**.

To execute this pattern, F-Script first looks at the first level of the pattern, so it considers **i:k**, which is **`@1:@2`**. It then applies this first level pattern: it generates the "messages" between the combinations (described by this pattern) of the elements of the arrays in the message expression.
These combinations, described by the semantics of the **`@1:@2`** pattern, are:

```
{'a11','a12','a13'} <----message----> {'b11','b12'}        (1)
{'a11','a12','a13'} <----message----> {'b21','b22'}        (2)

{'a21','a22','a23'} <----message----> {'b11','b12'}        (3)
{'a21','a22','a23'} <----message----> {'b21','b22'}        (4)
```

But what exactly is "`<----message---->`"?
`<----message---->` is, in fact, the original message pattern expression (more precisely: the part of the original message pattern expression composed by the pattern specification and the selector) minus the first level pattern.

So `<----message---->` is **`@2++@1`** (i.e. **j:l**).

Hence (1) is in fact: `{'a11','a12','a13'} @2++@1 {'b11','b12'}`
(2) is in fact: `{'a11','a12','a13'} @2++@1 {'b21','b22'}`
(3) is in fact: `{'a21','a22','a23'} @2++@1 {'b11','b12'}`
(4) is in fact: `{'a21','a22','a23'} @2++@1 {'b21','b22'}`

F-Script now executes (1), (2), (3) and (4) and returns an array with the four results of these four executions. The final result is: `{{result of the execution of (1), result of the execution of (2)}, {result of the execution of (3), result of the execution of (4)}}`.

17 Reduction

17.1 Introduction

Arrays implement a very useful message, the \ operator. This operator takes one argument, a block, and carries out what is called a **reduction**. Reducing an array consists in cumulatively evaluating a block on the elements of an array. For example, you add up the elements of an array with this command:

```
> {1,2,3,4} \  [:a :b| a+b]
10
```

In this case, you can do even better by using the compact form for the block:

```
> {1,2,3,4} \ #+
10
```

The result is computed as if you had entered: `1 + 2 + 3 + 4`

17.2 Examples of use

17.2.1 Maximum Reduction: Looking for the Largest

To select the single largest element of an array of numbers, `A`, you reduce the array by the maximum operator, as shown below:

```
A \ #max:
```

If `BALDUE` is the array of the balances due for all of the customers of a store,

```
BALDUE := {62.15, 127, 4.42, 18.65, 814.5, 76.42, 118.50, 6.01}
```

`then BALDUE \ #max:` gives the amount owed by the customer who has the biggest bill:

```
> BALDUE \ #max:
814.5
```

17.2.2 Minimum Reduction: Looking for the Smallest

In similar fashion, `A \ #min:` selects the (algebraically) smallest element of an array. For instance, if `ROOT1` contains the vector of all the first roots of a set of equations,

```
ROOT1 := {0.4815, -0.085236, 16.442, 0.000625, -4, 3.17215}
```

then `ROOT1 \ #min:` selects whichever value is the smallest.

```
> ROOT1 \ #min:
-4
```

17.2.3 OR Reduction: Looking for "Any"

Suppose you need to know whether a particular value exists anywhere in a long array. Let's say you want to know if any element of the array `A` is equal to the single number `Q`. If you type:

```
A = Q
```

then you will have a vector of Booleans indicating for each element of `A` whether or not it is equal to `Q`. You don't want to examine all these Booleans - you want to reduce them to a single result, either `true` or `false`, by applying the logical OR operation (implemented in F-Script by the | operator) so that it puts an OR between each of the elements:

```
false|false|false|false|false|true|false| ... |false|false|true|false|false
```

Thus, the instruction you need is typed like this:

```
A = Q \ #|
```

The result is `true` if there is a `true` anywhere in that array; it will be `false` if – and only if – every element is `false`.

Suppose `N` is an array of integers. You want to know if any of them is a perfect square. If an element of `N` is a perfect square, then its square root is an integer. In this case, sending the `fractionPart` message to the root will return zero. The following expression tests to see whether that condition is met by any elements of `N` :

```
> N := {103, 117, 142, 121, 135, 176, 149, 169, 128, 156, 118, 124, 133}
> (N raisedTo:0.5) fractionPart = 0 \ #|
true
```

And if you need to know not just whether any of them are perfect squares, but how many, you can find out by reducing the expression `(N raisedTo:0.5) fractionPart = 0` by `+` instead of `|` (this works because the FSBoolean class implements the `+` method, so a Boolean can be added as if true were 1 and false 0):

```
> (N raisedTo:0.5) fractionPart = 0 \ #+
2
```

17.2.4 AND Reduction: Looking for "All"

By using the AND reduction you can test whether **all** elements of an array satisfy a certain condition.

Suppose you want to know if every one of a set of equations has real roots. The vector of discriminants for these equations has been stored as the variable `DISC` (an array). Then

```
DISC >= 0
```

is an array of Booleans, indicating for each element of `DISC` whether it is true that the element is equal to or greater than 0. The operation AND placed between every element of this Boolean array will return the result "true" if every element is true, and otherwise "false". Thus, to find out if the test is true for every element of `DISC`, you enter:

```
DISC >= 0 \ #&
```

Suppose you have an array, `KEY`, and another array called `LOCK`. Both arrays are the same length. You need to know whether every element of `KEY` is equal to the corresponding `LOCK` element:

```
> KEY  := {1.01, 1.763, 1.808, 1.2346, 1.2272, 1.8095, 1.1}
> LOCK := {1.01, 1.763, 1.898, 1.2346, 1.2272, 1.8095, 1.1}
> KEY = LOCK \ #&
false
```

Evidently at least one of the elements of `KEY` does not match an element of `LOCK`.

17.2.5 Example Using the Sum of Products: Prices Times Quantity Ordered

Suppose that `PRICE` is a variable which contains the price list for various items sold by a store, and `C1` and `C2` are vectors indicating the quantities of the various items ordered by Customer 1 and Customer 2. Then the total bill for Customer 1 is the sum of the product of `PRICE` and `C1`, while the total bill for Customer 2 is the sum of the product of `PRICE` and `C2`.

```
> PRICE := {0.66, 1.4, 27.1, 2.39, 14, 7.6, 8.45, 2.8}

> C1    := {0   , 0  ,  2  , 1   ,  0, 0  , 0   , 0  }

> C2    := {12  , 7  ,  0  , 5   ,  0, 0  , 0   , 10 }

> C1 * PRICE \ #+
56.59

> C2 * PRICE \ #+
57.67
```

17.2.6 Example: The Area Under a Curve

One simple approach to finding the area under a curve is to divide it into a great many small trapezoids and then find the sum of the areas of all of them. Suppose you want to find the area under the curve produced by some function F of X for all the values of X between 0 and 1. You might get a suitably fine division by splitting that interval into 100 parts. Counting both end points, that makes 101 values. Suppose now that you have stored under the name FX the vector of the 101 values of F of X as X varies from 0.00 up to 1.00 in steps of 0.01. The area of any one of the trapezoids is the average of the two values of FX that bound it, times the width of the interval, which is 0.01. You don't actually have to average all those adjacent pairs; you can get the same effect by simply using FX times the width, provided that you first divide the first and last elements of FX by 2. Suppose that D is a vector whose first and last elements are 2, with ninety-nine 1 in between. Then you get the area under the curve by the instruction:

```
area := FX * width / D \ #+
```

17.2.7 Array Analysis

Suppose you have an array that represents the ages of a group of people:

```
AGES := {27, 51, 44, 62, 53, 19, 23, 52, 21, 53, 35, 51, 41}
```

Now you can ask some questions about these people:

Are they all older than 20?	`AGES > 20 \ #&`	returns	`false`
Is one of them older than 60?	`AGES > 60 \ #|`	returns	`true`
How old is the youngest person?	`AGES \ #min:`	returns	`19`
How many people are 35 or under?	`AGES <= 35 \ #+`	returns	`5`
Are they all over 25 but under 60?	`AGES > 25 & (AGES < 60) \ #&`	returns	`false`
How many are over 25 but under 60?	`AGES > 25 & (AGES < 60) \ #+`	returns	`9`
What is the average age?	`AGES \ #+ / AGES count`	returns	`40.769230769`
What is the percentage of people who are over 30?	`100*(AGES>30 \ #+) / AGES count`	returns	`69.23`

17.2.8 Example Using an array of arrays

Suppose M is a matrix with 3 rows and 4 columns. With F-Script, M is represented by an array of arrays.

```
M := {{1,  2,  3,  4},
      {5,  6,  7,  8},
      {9, 10, 11, 12}}
```

If you want to get the sum of the elements for each column, you simply execute the following reduction:

```
> M \ #+
{15, 18, 21, 24}
```

This reduction sums together each elements of M (i.e. each sub-array), and is thus equivalent to this explicit array operation

```
{1,  2,  3,  4}
       +
{5,  6,  7,  8}
       +
{9, 10, 11, 12}
```

which, thanks to the implicit message pattern rule, translate itself to:

```
{1+5+9, 2+6+10, 3+7+11, 4+8+12}
```

If you want to get the sum of the element for each row, you simply execute the following reduction:

```
> M @ \ #+
{10, 26, 42}
```

This time, we needed to apply the reduction one level deeper in M, in order to get to the rows' elements level. We did this using an explicit message pattern, which apply the reduction to each sub-array. Our instruction is thus equivalent to

```
{{1 ,  2,  3,  4} \ #+ ,
 {5 ,  6,  7,  8} \ #+ ,
 {9 , 10, 11, 12} \ #+ }
```

which is itself equivalent to:

```
{1 +  2 +  3 +  4,
 5 +  6 +  7 +  8,
 9 + 10 + 11 + 12}
```

Now, let's see an illustration on a concrete case. A company uses three database servers. Each time a server is down (due to a problem or for maintenance reasons), the duration of this downtime is noted. We have an array `A` that, for each server, list the duration (in min.) of each downtime incident for the current year:

```
A := {{49,18,123,3,87,21,7,24,11,19,243},
      {22,5,1,188,2,67},
      {13,15,7,22,55,81,3,19}}
```

We compute the total downtime of each server using:

```
> A @ \ #+
{605, 285, 215}
```

Although all the examples presented in this section use only arrays of numbers and arrays of booleans, we should stress that the reduction can be used for any kind of arrays, and with any kind of argument block (compact or not).

18 Advanced Indexing

As we have seen in the previous sections, you can select a particular element of an array by using the **at:** method. For example:

```
> {4,6,8,10,12,16} at:1
6
```

F-Script has two other very useful ways of indexing, which we introduce below.

18.1 Compression: selecting some elements from an array and omitting others

Suppose you have an array named A. You would like to generate a new array that contains some of the elements from A, but omits other. For instance, suppose A contains numbers and you want to keep all those that are greater than zero, while omitting those that aren't. The operation that does this is called *compression*. To perform compression, you index your array by an array of the same size, made of booleans. This operation returns an array. An element of the array you are indexing will be in the resulting array if its corresponding element in the index array is `true`.

Suppose that A is constructed like this:

```
A := {1, -2, 3, 4, -5, 6, 7, 8}
```

You want to keep all the elements from V except the second and fifth. So as an index you need an array that has the same length as V, and all of whose elements have the value `true` except the second and the fifth, which must be `false`.

```
> A at:{true, false, true, true, false, true, true, true}
{1, 3, 4, 6, 7, 8}
```

If all the elements of the selection vector (i.e. the array of booleans used as index) are `true`, then all the elements from the indexed array are preserved:

```
> A at:{true, true, true, true, true, true, true, true}
{1, -2, 3, 4, -5, 6, 7, 8}
```

Conversely, if all the elements of the selection vector are `false`, then none of the elements from the indexed array are selected, and so the result is an empty vector:

```
> A at:{false, false, false, false, false, false, false, false}
{}
```

You will recall that when F-Script tests whether a relationship is true, it responds with boolean objects. These booleans are just what is needed for the selection vector during compression. For instance, suppose you would like to keep from A only those elements that are greater than some constant X. The expression `A > X` generates a response for each element in A. That response is `true` for each element of A that is greater than X, and `false` for each that is not:

```
> A > X
{false, false, false, true, false, true, true, true}
```

This expression can be used directly in the compression, like this:

```
> A at: A > X
{4, 6, 7, 8}
```

(Evidently, X was something smaller than 4, but greater than 3)

```
> A at: A <= 0
{-2, -5}
```

Suppose you manage a car store. You have an array named `cars` that contains objects of class “Car”. Each one represents a particular car sold by you. The Car class has a method `owner`, which returns an object of class “Customer”. This Customer class has a `mail:` method with which you can send an e-mail to a customer. The Car class has also a `type` method that returns the type of the car, and a `groupId` method that returns a group identification for the car. Say you want to send a message to all the owners of a car of type "BMW A" of the group 544. You type:

```
(cars at:(cars groupId = 544 & (cars type = 'BMW A'))) owner distinct mail:'Dear customer
blah blah blah'
```

The `distinct` method returns only the different objects of an array. It is used in the example above because we want to send only one message to a customer, even if it has purchased several cars of the selected group and type.

18.2 Indexing by an Array of Integers

Once an array exists, you may want to refer to the elements in certain positions within it. You can select several elements at once by indexing with an array of integer.

```
> {4,6,8,10,12,16} at:{1,2,5,2}  "This will select the elements at index 1, 2, 5 and 2"
{6, 8, 16, 8}
```

If you use an index which refer to an element which doesn't exists in the array, F-Script is unable to execute the instruction and reports an error.

A vector of 0 index number (i.e. an empty vector used as an index) refers to none of the elements of an array, and therefore it produces an empty vector of results.

```
> {4,6,8,10,12,16} at:{}
{}
```

18.3 Indexing arrays of arrays

Suppose M is a matrix with 3 rows and 4 columns. With F-Script, M is represented by an array of arrays.

```
M := {{1,  2,  3,  4},
      {5,  6,  7,  8},
      {9, 10, 11, 12}}
```

If you want to refer to row two, you simply select the second element of M:

```
> M at:1
{5, 6, 7, 8}
```

Note that since F-Script indexing start at 0, index of row two is 1.

If you want to refer column two, you enter:

```
> M @at:1
{2, 6, 10}
```

Here, note the use of a message pattern. This expresses the fact that we want to index each element of M (i.e. each rows) and not M itself.

If you want to refer to the element in row two, column three, you enter:

```
> (M at:1) at:2
7
```

If you would like the third and fourth elements in that row you enter:

```
> (M at:1) at:{2,3}
{7, 8}
```

If you would like the elements in column four, rows one and two and one, you enter:

```
> (M at:{0,1,0}) @at:3  "Note the use of a message pattern to get the fourth column"
{4, 8, 4}
```

You use the same procedure to select a sub-matrix from within M. If you want the matrix of those elements which are on row two and three and columns one, two and one of M, you enter:

```
> (M at:{1,2}) @at:{0,1,0}
{{5,  6, 5},
 {9, 10, 9}}
```

The result is a two by three matrix.

If you would like to select the rows for which the sum of elements are greater than 15, you enter:

```
 > M at: M @ \ #+ > 15
{{5,  6,  7,  8},
 {9, 10, 11, 12}}
```

Rows two and three are selected.

19 Other Interesting Messages

This section introduces some other interesting messages, but it is not exhaustive. For example, see >< and `transposedBy:` in the FSNSArray category documentation.

19.1 iota

The `iota` method, sent to a number, generates an array of consecutive integers in the range [0..receiver-1]. For example:

```
> 12 iota
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

One way to think of `iota` is to say that it generates all the index numbers for an array of a given size. It is used frequently, in many different situations.

For example, an array, `A`, contains 80 elements. If we want to get the first 15 we could write `A at:{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14}`, but `A at: 15 iota` is shorter.

The `iota` method is very handy when you want to refer to a consecutive block of numbers. For instance, you can get the first 35 powers of 2 simply by typing this instruction:

```
> 2 raisedTo:@ 35 iota
{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536,
131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864,
134217728, 268435456, 536870912, 1073741824, 2147483648, 4294967296, 8589934592,
17179869184}
```

You can create a suite of N numbers, starting at ORIGIN, separated by a constant STEP, with the following formula: *N iota * STEP + (ORIGIN)*. For example:

```
> 10 iota * 5 + 100
{100, 105, 110, 115, 120, 125, 130, 135, 140, 145}
```

19.2 Set Manipulation

The FSNSArray category provides standard Set manipulation messages:

- The `union:` message returns the union of the receiver and the argument, considered as sets:

```
> {1,2,3,4,5,6} union:{4,5,6,7,8,9,10}
{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

- The `intersection:` message returns the intersection of the receiver and the argument, considered as sets:

```
> {1,2,3,4,5,6} intersection:{4,5,6,7,8,9,10}
{4, 5, 6}
```

- The `difference:` message returns the receiver minus the argument, considered as sets:

```
> {1,2,3,4,5,6} difference:{4,5,6,7,8,9,10}
{2, 3, 1}
```

Elements are compared using the equality notion (i.e. **isEqual:**).

19.3 Finding the index of an object within an array

Suppose that A is an array which has the following values:

```
A := {1.2916, 1.3184, 1.2196, 1.1629, 1.2619, 1.2961, 1.1326}
```

and B is a single number:

```
B := 1.2619
```

Then the expression

```
A ! B
```

means "Where in A can you find an object equal to B?" Note that object equality is determined using the "isEqual:" method. The expression is read as "the A-index of B".

F-Script responds with the index number that shows which element of A is equal to B:

```
> A ! B
4
```

As you may have noticed, this method is very similar to the method "indexOfObject:" defined by NSArray.

If you would like to know where in A its largest value is located, that can be found using:

```
> A ! (A \ #max:)
1
```

And the smallest value likewise:

```
> A ! (A \ #min:)
6
```

19.3.1 Finding Several Indices at Once

Suppose instead of being a single number, B is itself an array. In that case, it is easy to ask for the A-index of each of the elements of B in turn. You just have to use a message pattern:

```
> B := {1.2619, 1.2916, 1.2961}

> A !@ B
{4, 0, 5}
```

Note that with this message pattern, the result always has in it one element for each element in the argument of `!`.

19.3.2 Looking for the Index Number of an Object that isn't there

Suppose that the object passed as argument to `!` simply isn't represented anywhere in the receiver. What number does F-Script returns as the index of this nonexistent element? For an object that isn't represented anywhere in the receiver, F-Script responds with the first illegal index for the receiver. For instance, suppose that A is a vector of seven elements with the following values:

```
> A := {11, 12, 13, 14, 22, 77, 18}
```

Then the possible index numbers for this vector are the integers 0, 1, 2, 3, 4, 5, 6. The first "illegal" index for this array is 7. If you ask for the index of a value that isn't anywhere in the vector A, F-Script responds by saying it is at location 8. For instance:

```
> A ! @{77, 15}
{5,7}

> A ! 'hello'
7
```

19.4 Sorting

The result of the `sort` message sent to an array is an array of integers containing the indices that will arrange the receiver of `sort` in ascending order. For example:

```
> A := {5,2,1,3,6,4}
> A sort
{2, 1, 3, 5, 0, 4}
```

You can then get `A` in ascending order by indexing it with the result of `sort`:

```
> A at:(A sort)
{1, 2, 3, 4, 5, 6}
```

The advantage of doing it this way is that, once you have the ordered index numbers, you can then apply them not only to the original scrambled array, but to any other array of the same size. For example, suppose you have an array `E` of object of class `Employee`. An Employee object responds to the `salary` message. To get the employees ordered by salary, you type:

```
E at:(E salary sort)
```

The sort method is stable, that is, if two members compare as equal, the order of their indices in the returned array is preserved.

When sorting, the < message is sent in order to compare the elements of the array. So, note that sorting will only work if all the elements of the array properly implement this comparison message.

20 Smalltalk collection protocols & F-Script

F-Script differs significantly from standard Smalltalk when it comes to collection support. F-Script is based on the Cocoa collection framework and provides an original high-level object-oriented programming model, based on established array-programming techniques.

Smalltalk provides a set of powerful and handy control structures for dealing with collections of objects (e.g. applying an operation to each element of a collection, selecting elements verifying a given condition etc.). However, the F-Script array programming model subsumes this approach, for most purposes. This is why the traditional Smalltalk collection methods such as do:, collect:, etc. are not present in F-Script.

In this section we show how to achieve, in F-Script, the effect of some of the methods provided by standard Smalltalk. In the following examples we assume that C is a collection of Employee instances (in F-Script, C is represented by an array). Objects of class Employee respond to the salary and raiseSalary: messages.

20.1 do:

The `do:` method is the fundamental collection iteration method in Smalltalk. It is used to evaluate a block with each element of a collection. On the whole, in F-Script, this is expressed by an array expression. In this example, we want to add 1000 to the salary of every employee in our collection C:

Smalltalk collection protocols	`C do:[:e| e raiseSalary:1000]`
F-Script	`C raiseSalary:1000`

Occasionally, you may still want to use a block-based approach in F-Script (for instance, it may not be possible to express your instructions as an array expression). In this case, you can use the `:@` pattern with the `value` method:

```
[:e| e raiseSalary:1000] value:@C
```

Note that this remark also applies to the examples examined in the following sections.

20.2 with:do:

The `with:do:` method allows you to iterate over two collections at the same time, evaluating a block for each element of the first collection and each corresponding element of the second collection. On the whole, in F-Script, this is expressed by an array expression. In this example, we suppose that A is a collection of numbers representing the amount to add to the salary for each employee.

Smalltalk collection protocols	`C with:A do:[:e :amount| e raiseSalary:amount]`
F-Script	`C raiseSalary:A`

Of course, it is also possible (as always) to use a block-based approach with an explicit message pattern (in this case the `:@:@` pattern is used):

```
[:e :amount| e raiseSalary:amount] value:@C value:@A
```

20.3 collect:

The `collect:` method answers a collection constructed by gathering the result of evaluating a block with each element of the receiver. With F-Script we generally use an array expression to do this. In this example, we want to generate a collection containing the salary of each employee in C:

Smalltalk collection protocols	`C collect:[:e| e salary]`
F-Script	`C salary`

20.4 select:

The `select:` method answers a collection which contains only the element in the receiver which cause the block to evaluate to true. With F-Script, we combine a boolean array expression with a compression. In this example, we want to select the employees whose salary is lesser than 5000:

Smalltalk collection protocols	`C select:[:e| e salary < 5000]`
F-Script	`C at: C salary < 5000`

20.5 reject:

The `reject:` method answers a collection which contains only the element in the receiver which causes the block to evaluate to false. With F-Script, we combine a Boolean array expression with a compression. In this example, we want to "reject" the employees whose salary is less than 5000, thus selecting the employee whose salary is greater or equal to 5000:

Smalltalk collection protocols	`C reject:[:e| e salary < 5000]`
F-Script	`C at: C salary >= 5000`

20.6 allSatisfy:

The `allSatisfy:` method is used to test whether all the elements of the receiver fulfill a certain condition. With F-Script we combine a Boolean array expression with an AND reduction. In this example, we want to know if all the employees have a salary greater than 1000:

Smalltalk collection protocols	`C allSatisfy:[:e| e salary > 1000]`
F-Script	`C salary > 1000 \ #&`

20.7 anySatisfy:

The `anySatisfy:` method tests whether an element of the receiver fulfills a certain condition. With F-Script we combine a Boolean array expression with an OR reduction. In this example, we want to know if there are any employees with a salary greater than 1000:

Smalltalk collection protocols	`C anySatisfy:[:e| e salary > 1000]`
F-Script	`C salary > 1000 \ #|`

Note that `anySatisfy:` may sometimes perform better than the F-Script approach. This is because `anySatisfy:` stops iterating the collection as soon as an element is found that meets the condition. If you are in a situation where this is a problem, you can choose to explicitly iterate using the `whileTrue:` method.

20.8 inject:into:

The `inject:into:` method applies an operation cumulatively to all the receiver elements. With F-Script we use reduction, which is very similar. In this example, we want to get the sum of all the salaries:

Smalltalk collection protocols	`C inject:0 into:[:sum :e| sum + e salary]`
F-Script	`C salary \ #+`

20.9 asSortedCollection:

The `asSortedCollection:` method is used to sort a collection using a sort order specified by a given sort block. With F-Script we combine the `sort` method with indexing. In this example, we want to sort the employees by salary in ascending order:

Smalltalk collection protocols	`C asSortedCollection:[:e1 :e2| e1 salary < e2 salary]`
F-Script	`C at: C salary sort`

If we want to sort by salary in descending order instead:

Smalltalk collection protocols	`C asSortedCollection:[:e1 :e2| e1 salary > e2 salary]`
F-Script	`C at: C salary sort reverse`

21 Other Notes

21.1 The latent object

When F-Script is launched, a file named **fs_latent** is searched for at the top level of the F-Script repository. If this file exists it must contain the textual representation of a block literal. This block is then instantiated and sent the "`value`" message. The idea is that you can use this file to do whatever you want before the interactive F-Script session begins. This is very similar to the ".cshrc" file on Unix.

In addition, when a space is loaded (cf. `loadSpace:` in class System), an object named **fs_latent** is searched for in this space. If such an object exists, it is sent the "`value`" message.

22 Guidelines for Implementing User Objects

Basically, a user object must be user-friendly and designed to be used interactively. Methods designed to be used interactively should:

- React in a user-friendly way in the event of an error (bad arguments, problems using some resources, etc.). A user method should generate a message informing the user of the problem and, whenever possible, should help the user to find a solution or more information.

- Shield the user from memory management tasks as much as possible: never transfer the ownership of a new object to the user (in many cases, this means: return auto-released objects).

When it makes sense, a user object should also:

- Conform to the Cocoa NSCoding Protocol (this will primarily be used by F-Script for saving objects to disk).

- Implement `clone` and `setValue:` methods (cf. section 13)

If the user object provides a graphical interface for interaction with the user, it may be a good idea to implement the `-(void)inspect` method on the object. This method should open the graphical user interface. Some built-in F-Script objects already implement this method. In addition, if implemented, it will be invoked by the F-Script object browser when the user clicks on the inspect button.

Here is how the division operator of the NSNumber class would deal with the requirements on user methods: Invoked from F-Script, the "`operator_slash:`" method takes an NSNumber as argument and returns an auto-released NSNumber. As shown in our example, we can also use the “double” native Objective-C type since the mapping will be done automatically. The argument must not be zero, so the method tests this condition before processing:

```
-(double)operator_slash:(double)operand
{
  if (operand == 0) FSExecError(@"division by  zero");

  return ([self doubleValue]/operand);
}
```

The FSExecError() function is provided by the F-Script framework. FSExecError() raises an exception, so the execution of `operator_slash:` ends if this function is executed.

This is all that is required!

Another possibility is to require an object of class NSNumber for the argument. We will explore this alternative below, so as to illustrate how to handle arguments that are objects:

```
-(double)operator_slash:(NSNumber *)operand
{
  double operandValue = [operand doubleValue];

  if (operandValue == 0) FSExecError(@"division by  zero");

  return [self doubleValue]/operandValue;
}
```

In the current implementation of F-Script, the classes of arguments that are objects are not tested when a message is sent (as with Smalltalk). For the "`operator_slash:`" method, this means that if a non-NSNumber object is passed as argument, the method may fail. Therefore, you might want to check the classes of the arguments:

```
-(double)operator_slash:(NSNumber *)operand
{
  double operandValue;

  if (![operand isKindOfClass:[NSNumber class]]) FSExecError(@"argument 1 of method \"/\"
must be a number");

  operandValue = [operand doubleValue];

  if (operandValue == 0) FSExecError(@"division by  zero");

  return [self doubleValue]/operandValue;
}
```

A more convenient way to test an argument class is by using a function called `FSVerifClassArgs()` or a variation called `FSVerifClassArgsNoNil()`:

```
-(double)operator_slash:(NSNumber *)operand
{
  double operandValue;

  FSVerifClassArgsNoNil(@"/",1,operand,[NSNumber class]);

  operandValue; = [operand doubleValue];

  if (operandValue == 0) FSExecError(@"division by  zero");

  return [self doubleValue]/operandValue;
}
```

These functions are provided by the F-Script framework, and are explained in more detail in section 52.

23 F-Script in Action: the Flying Tutorial

To activate this tutorial into your F-Script session just type:

```
> sys installFlightTutorial
```

This will put the required objects in your workspace.

23.1 Object Model

In this tutorial, you manage an airplane company. Three classes have been designed for you: **Flight**, **Airplane** and **Pilot**.

- An airplane is attributed a number called its **ident**, and has a **model**, a **capacity** and is located at a particular **location**.
- A pilot has a **name**, an **address**, an age and a **salary**.
- A flight is attributed a number called its **ident**, and has a **departureDate**, an **arrivalDate**, a **departureLocation**, an **arrivalLocation**. In addition to this, a Flight is under the responsibility of a **pilot** and is executed with a particular **airplane**.

Here are extracts of the Objective-C headers for these classes:

Airplane.h

```
////////////////////////////////// Methods For Airplane ////////////////////////////////

+ (id) aeroplaneWithIdent:(id)theIdent model:(id)theModel capacity:(id)theCapacity
location:(id)theLocation;

- (id)   capacity;
- (void) setCapacity:(id)theCapacity;
- (id)   ident;
- (void) setIdent:(id)theIdent;
- (id)   location;
- (void) setLocation:(id)theLocation;
- (id)   model;
- (void) setModel:(id)theModel;
```

Pilot.h

```
////////////////////////////////// Methods For Pilot ///////////////////////////////////

+ (id) pilotWithName:(id)theName address:(id)theAddress salary:(id)theSalary age:(id)theAge;

- (id)   age;
- (void) setAge:(id)theAge;
- (id)   address;
- (void) setAddress:(id)theAddress;
- (id)   name;
- (void) setName:(id)theName;
- (id)   salary;
- (void) setSalary:(id)theSalary;
```

Flight.h

```
////////////////////////////////// Methods For Flight ///////////////////////////////////

+ (id) flightWithIdent:(id)theIdent pilot:(id)thePilot airplane:(id)theAirplane
departureDate:(id)theDepartureDate arrivalDate:(id)theArrivalDate
departureLocation:(id)theDepartureLocation arrivalLocation:(id)theArrivalLocation;

- (id)   airplane;
```

```
- (void) setAirplane:(id)theAirplane;
- (id)   arrivalDate;
- (void) setArrivalDate:(id)theArrivalDate;
- (id)   arrivalLocation;
- (void) setArrivalLocation:(id)theArrivalLocation;
- (id)   departureDate;
- (void) setDepartureDate:(id)theDepartureDate;
- (id)   departureLocation;
- (void) setDepartureLocation:(id)theDepartureLocation;
- (id)   ident;
- (void) setIdent:(id)theIdent;
- (id)   pilot;
- (void) setPilot:(id)thePilot;
```

Three arrays are defined: **F** contains all the Flight objects, **A** contains all the Airplane objects and **P** all the Pilot objects.

23.2 Visualization

You can see what is in your three arrays by simply evaluating them. For example, enter:

```
> P
```

To get a better view of the array, try this:

```
 > P inspectWithSystem:sys blocks:{#name, #address, #salary, #age}
```

You should see a new window with a tabular representation of P.

23.3 Querying

Give all the salaries for each pilot.

```
> P salary
```

Give the sum of all salaries.

```
> P salary \ #+
```

Give the average salary.

```
> P salary \ #+ / P count
```

Give the pilots who live in Paris.

```
> P at:P address = 'PARIS'
```

Give the pilots with a salary greater than the average salary.

```
> P at:P salary > (P salary \ #+ / P count)
```

Rank the pilots by salary in increasing order.

```
> P at:P salary sort
```

Rank the pilots by salary in decreasing order.

```
> P at:P salary sort reverse
```

Give the number of pilots living in Paris with a salary greater than or equal to 200 000.

```
> (P at:P address = 'PARIS' & (P salary >= 200000)) count
```

Give the airplanes whose locations are in the list {'PARIS', 'NEW YORK', 'BOSTON'}

```
> A at:({'PARIS', 'NEW YORK', 'BOSTON'} containsObject:@ A location)
```

or

```
> A at:A location =@ {'PARIS', 'NEW YORK', 'BOSTON'} \ #|
```

or

```
> A at:A location @1=@2 {'PARIS', 'NEW YORK', 'BOSTON'} @\ #|
```

or

```
> A at:(A location >< {'PARIS', 'NEW YORK', 'BOSTON'}) @count > 0
```

or

```
> A at:{'PARIS', 'NEW YORK', 'BOSTON'} !@ A location < 3
```

Give the airplanes whose locations are NOT in the list {'PARIS', 'NEW YORK', 'BOSTON'}

```
> A at:(A location =@ {'PARIS', 'NEW YORK', 'BOSTON'} \ #|) not "among other possibilities"
```

For each flight, give the pilot.

```
> F pilot          " Returns an array, of the same size than F, of Pilot instances"
```

For each flight, give the name of the pilot.

```
> F pilot name
```

For each flight with a pilot living in Paris, give the airplane model.

```
> (F at:F pilot address =  'PARIS') airplane model
```

Give a list of the different airplane models of the flights with a pilot who lives in Paris.

```
> (F at:F pilot address = 'PARIS') airplane model distinct
```

For each pilot, give all the flights that the pilot is responsible for.

```
> F at:@ P >< F pilot  "Returns an array of the same size than P. Each element
                        is an array of flights"
```

For each pilot, give the number of flights that the pilot is responsible for.

```
> (F at:@ P >< F pilot) @ count
```

Are all pilots responsible for at least two flights?

```
> (F at:@ P >< F pilot) @ count > 2 \ #&
```

Give the airplanes that are in the same location as the airplane number 1207.

```
> A at:A location = ((A at:A ident = 1207) at:0) location
```

or

```
> A at:A location = (A at:A ident ! 1207) location
```

For each airplane, give all the airplanes that are in the same location.

```
> A at:@ A location >< A location
```

For each airplane, give all the OTHER airplanes that are in the same location.

```
> A at:@(A location >< A location @difference:@ A index @enlist)
```

For each airplane, give all the airplanes that are NOT in the same location.

```
> A difference:@ (A at:@ A location >< A location)
```

Give the pilots that are responsible for at least one flight in each airplane.

```
> P at:(F at:@ P >< F pilot) airplane @distinct @count = A count
```

How many cities are the destination of fewer than five flights?

```
> ((F arrivalLocation union:F departureLocation) >< F arrivalLocation) @count < 5 \ #+
```

Give the number of flights for each pilot, for each airplane. Put the result in a variable named N.

```
> N := (A >< @ (F at:@ P >< F pilot) airplane) @@count
> N
```

Using N, give the number of flights for each pilot.

```
> N @\ #+
```

Using N, give the number of flights for each airplane.

```
> N \ #+
```

Using N, give the number of flights for each airplane, for each pilot.

```
> N transposedBy:{1,0}
```

24 GUI Tutorial

The following instructions open a window with two text fields and a button.
Enter these instructions in your interpreter.

```
w := NSWindow alloc initWithContentRect:(100<>100 extent:300<>200)
styleMask:NSTitledWindowMask+NSClosableWindowMask backing:NSBackingStoreBuffered
defer:false.
w setTitle:'My calculator'.
w orderFront:nil.
button := (NSButton alloc initWithFrame:(100<>20 extent:100<>50)).
button setBezelStyle:NSRoundedBezelStyle.
w contentView addSubview:button.
t1 := (NSTextField alloc initWithFrame:(60<>120 extent:50<>25)).
t2 := (NSTextField alloc initWithFrame:(200<>120 extent:50<>25)).
w contentView addSubview:t1.
w contentView addSubview:t2.
bl :=  [t2 setDoubleValue:t1 doubleValue * 2].
button setTarget:bl.
button setAction:#value:.
```

Enter a numerical value in the left field, then click the button. The right field shows the left field's value multiplied by 2.

25 Puzzle

Question: what does the `puzzle` block do when executed?

```
puzzle :=  [:v |
    |u w s|
    u := v dup.
    s := {}.
    [u count > 0] whileTrue:
    [
        w := u = (u \ #min:).
        s := s ++ (u at:w).
        u := u at:w not.
    ].
    s.
]
```

Turn the page to find out…

Answer:

The puzzle block example is a sorting script (note, however, that a `sort` method is already provided by arrays; this script is just a puzzle for fun).

```
> puzzle := [:v |
    |unsorted which sorted|
    unsorted := v clone.
    sorted := {}.
    [unsorted count > 0] whileTrue:
    [
        which := unsorted = (unsorted \ #min:).
        sorted := sorted ++ (unsorted at:which).
        unsorted := unsorted at:which not.
    ].
    sorted.
]

> puzzle value:{2,56,1,3,2,5,2,1,-123,0,67}
{-123, 0, 1, 1, 2, 2, 2, 3, 5, 56, 67}
```

This sorting script arranges the elements of an array in ascending order, using compression to find which elements should go first, and catenation to reassemble them into a new, ordered array. The procedure is as follows:

1. Given an array `v`, create a working copy of it named `unsorted`.

2. Call the sorted array that results `sorted`. Start with `sorted` being an empty array.

3. Test to see whether any elements remain in `unsorted`. If there are none, go to step 8.

4. Set up the logical array `which`, with a `true` corresponding to each element of `unsorted` that is equal to the minimum of `unsorted`.

5. Compress `unsorted` by `which`. That is, pick out from `unsorted` those elements that are equal to its minimum. Catenate them to those already found in `sorted`.

6. Compress `unsorted` by the negation of `which`. That is, respecify `unsorted` to be all those elements that were not selected.

7. Return to step 2.

8. All elements are ordered in `sorted`. Return it.

Note that this script works only if the objects to be sorted respond to the `min:` method.

26 Questions & Answers

Q 1: Some symbolic constants names standard to Cocoa (for example `NSBackingStoreBuffered`) are pre-defined in the namespace of the F-Script Interpreter, but not all constants are. What am I supposed to do?

A: If a symbolic constant name is not defined you have to directly use the value associated with this symbolic constant. To find this value, go to the header file where this constant is defined.

Q 2: Why is array indexing now based at 0? In the first version of F-Script it was based at 1, like Smalltalk and APL.

A: Things have been unified with Cocoa Indexing, which starts at 0. This did not just have an impact on the indexing method, but also on several other methods including: index, iota, !, !!, transposedBy:, sort, at:put:, insert:at:, ><.

Q 4: I read somewhere that there is an F-Script Interface Builder palette but found no documentation about it.

Yes, there is a palette named FScriptPalette.palette in the distribution. This feature is still at an experimental stage. Some guidelines are given below:

You can do two things very different with this palette:

- Provide your user with an F-Script command line view embedded within your application (this is not one of the experimental parts and it is very effective, but note that you can also do this programmatically).

- Use F-Script from Interface Builder for implementing parts of your application logic (this is the experimental part).

This palette lets you drag and drop a FSInterpreterView into your application. This subclass of View is a command line interface to an F-Script interpreter. The interpreter is bundled into the FSInterpreterView (see the Reference Manual for a description of the FSInterpreterView class). This is the same view used by the F-Script.app application. By using this palette you get all the functionality of F-Script in your application.

Please note that you can drop this view into a non-visible window. By doing this you can use F-Script from Interface Builder for implementing your application logic.

To get the full power of this palette you have to know how to enable the "live" mode. This mode lets you use F-Script to code your application logic directly from Interface Builder. To enable the live mode, proceed as follows :

1) Drop a FSInterpreterView into a window.

2) Double-click in this view.

3) A window opens: this is an F-Script session. You are now in live mode!

When you are in live mode, the connections you make using the FSInterpreterView's outlets *object1,object2…object9* are immediately active (in the normal use of IB, the connections are only made when you go in test mode). You can then start to use the objects you have connected. You can use them from the active F-Script session by using their names: *object1…object9*.

The possibilities are then boundless, since F-Script gives you total control over all objects that can be manipulated with IB. For example, you can set the target of a button to be an F-Script block provided by you, and the action to be "**#value:**"
This block will then be executed when the button is pressed.

This way, Interface Builder becomes the GUI Builder for the F-Script programmer.

Note that when you use this palette, you must link your application with the F-Script framework.

Q 5: Any support for AppleScript?

Yes, through the "eval" verb. Example:

```
tell application "F-Script"
	eval "10 iota"
end tell
```

User Classes and Categories

27 Array *(F-Script User Object)*

Inherits From:	NSMutableArray
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	FScript/Array.h

Class Description

Array is a concrete subclass of NSMutableArray. It provides an optimized implementation of F-Script's array programming model.

Inserting a number in an instance of this class is a special case; for efficiency purposes, the number you insert may not be really referenced by the array, instead the array may just store its value. This means that, when indexing the array, you may not get back your original number object, but another number object with the same value.

NSArray and NSMutableArray, which are Array superclasses, do not allow the **nil** value to be referenced in an array. This is not an acceptable limitation for an array language such as F-Script. To avoid this problem, the Array class supports the insertion of **nil**. However, you should be careful when using an array that actually contains **nil,** because that breaks an important NSArray class invariant. Generic code dealing with NSArrays is usually not prepared to handle an array that contains **nil**, and, consequently, may fail in this situation. When using such an array, you should only pass it to code that has been specifically designed to handle **nil** in arrays, or is known to be safe in this case. All the methods declared in the Array class header (i.e., *Array.h*) as well as in the FSNSArray and FSNSMutableArray category headers (i.e., *FSNSArray.h, FSNSMutableArray.h*) have been designed to handle **nil** in arrays correctly. This includes all the methods described in this manual.

28 Block *(F-Script User Object)*

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	FScript/Block.h

Class Description

Note: *Extensive documentation about blocks can be found in the Smalltalk literature.*

The Block class provides the concept of script, or procedure, or function. A block is an object containing a sequence of F-Script statements. When a block is evaluated using an acceptable **value** message, its statements are executed. A block can take an arbitrary number of arguments and can have its own temporary variables, as well as having access to variables of its enclosing environment. The statements in the block are executed when the block is sent a message in the form "**value**","**value:**", "**value:value:**" etc. where the number of colons in the message is at least the same as the number of arguments the block takes (extra arguments are ignored, but it is an error not to provide enough). When a block takes more than twelve arguments you cannot use the "**value[:]**" form, but instead you have to use the **valueWithArguments:**, a method which works for any number of arguments**.**

Among other things, Blocks are used to implement several control structures in F-Script and allow the programmer to easily extend the system using customized control structures.

Here are some examples of blocks:

[5 * 2] is a block which evaluates to **10** when executed.

[:a :b | a+b] is a block with two arguments, **a** and **b**, which, when executed, send the **+** message to its first argument using its second argument as a parameter.

[:a :b| |loc | loc := a clone. a setValue:b. b setValue:loc] is a block with two arguments, **a** and **b**, and a temporary, **loc**. When executed, this block switches the values of its two arguments (in this example, they must support the setValue: method).

The general form of block literal is given by this grammar

```
<block>              ::= <compact block> | '[' <block body> ']'
<compact block>      ::= '#' <selector>
<block body>         ::= <declarations> <statements>
<declarations>       ::= <block argument>* '|'] [<temporaries>]
<block binding>      ::= '*' identifier
<block argument>     ::= ':' identifier
<temporaries>        ::= '|' identifier* '|'
```

The <statements> part is some regular F-Script code.

The <declaration> part is optional as are each of its subparts. The *argument* subpart is a list of argument declarations. An argument declaration is made up of the sign **:** followed by an identifier. The *temporaries* subpart is a list of identifiers inside a couple of |

Like other literal expressions in F-Script, the evaluation of a block literal in an F-Script expression creates and returns a block instance; an object of the Block class.

To execute a block object, send it a **value...** message. For example:

[5 * 2] value evaluates to **10.**

[:arg | arg sqrt] value:16 evaluates to **4**.

[:a :b | a+b] value:2 value:3 evaluates to **5**.

[:a :b | a+b] valueWithArguments:{2,3} evaluates to **5**.

Expressions within a block can, of course, reference temporary variables and arguments of the block. They may also reference “externals” identifiers defined in the environment where the block was syntactically defined (these may be global variables as well as arguments or temporary variables of a syntactically enclosing block). Each block object is an independent *closure* that captures the current bindings for any of these identifiers which are referenced from within the block code. Any such captured bindings and their associated discrete variable or objects are preserved as long as the block object continues to exist and is available for evaluation. Note that the value of any such captured discrete variables and the state of any object captured by an argument binding remain subject to possible modification.

There is another form of Block literal called the compact form. This form may be used when all the block does is send a message. Here are some examples:

#sqrt	is functionally equivalent to	**[:a | a sqrt]**
#max:	is functionally equivalent to	**[:a :b | a max:b]**
#+	is functionally equivalent to	**[:a :b | a+b]**
#between:and:	is functionally equivalent to	**[:a :b :c | a between:b and:c]**

A compact literal creates what we call a “compact block.”

Blocks are powerful tools. Because they are objects, they can be used as arguments to methods; they can be saved and loaded, inserted into collection etc. In addition, Blocks support the control structure style of F-Script. In F-Script there are no special instructions for control structures such as *while...* or *if...else*. These control structures are realized by using the

general messaging mechanism. (See the method **ifTrue:** and **ifTrue:ifFalse:** in the FSBoolean class and **whileTrue:** in this class.)

Once a block has been created, it can be interactively edited by sending it the **inspect** message.

Several methods of this class will cause blocks to be compiled or executed. If this leads to an error (i.e. syntax error during compilation or a run-time error during execution) then the method will be interrupted and an exception will be raised. In some cases this exception-raising behavior may not be what you want. For instance, if your block is the target of a button and the action method is **value:**, the button will not handle the exception raised by **value:** in the advent of an error. The **guardedValue:** method provides a useful alternative in such cases. See also the **executeWithArguments:** method.

Method Types

Executing a block	-executeWithArguments: -guardedValue: -valueWithArguments: -value -value: -value:value: -value:value:value: -value:value:value:value: -value:value:value:value:value: -value:value:value:value:value:value: -value:value:value:value:value:value:value: -value:value:value:value:value:value:value:value: -value:value:value:value:value:value:value:value:value: -value:value:value:value:value:value:value:value:value:value: -value:value:value:value:value:value:value:value:value:value:value: -value:value:value:value:value:value:value:value:value:value:value:value:
Control structures support	-whileTrue -whileTrue:
Exception handling	-onException:
Copying	-setValue: -clone
Editing	-inspect
String representation	-description
Other properties	-argumentCount
Testing equality	-isEqual:
Hashing	-hash

Instance Methods

argumentCount

-(int) **argumentCount**

Answers the number of arguments required to evaluate the receiver.

clone

-(Block*) **clone**

Returns an auto-released copy of the receiver.

description

-(NSString *) **description**

Returns a string representation of the receiver using the block literal format.

executeWithArguments:

-(FSInterpreterResult *) **executeWithArguments:**(NSArray *)*arguments*

Executes the receiver with arguments provided in *arguments*. Returns an FSinterpreterResult instance representing the outcome of the execution.

guardedValue:

-(id) **guardedValue:**(id)*arg1*

Executes the receiver, using *arg1* as an argument, and returns the result. If an error occurs during execution, displays the block call stack to the user and returns nil.
This method is useful, for example, when you want a block to be the target of an NSControl (for instance a button). Instead of using `value:` as the action method, you may use this method in order to have the block inspector pops-up automatically in case of error.

hash

@protocol NSObject
-(unsigned int) **hash**

Returns a hash code based on the receiver value.

inspect

-(void) **inspect**

Opens the receiver inspector, a window in which you can interactively edit the receiver source code. The source code displayed in the inspector is always in-sync with the block.

isEqual:

@protocol NSObject
-(BOOL) **isEqual:**(id)*anObject*

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A block is considered equal to another object if either there is identity between the block and the object (i.e. if they are in fact the same object in memory) or if they are both compact blocks with the same selector.

onException:

-(id) **onException:**(Block*) *handler*

Precondition: receiver argumentCount = 0 & (handler argumentCount <= 1)

The receiver is evaluated so that, if its evaluation results in a Cocoa exception, then *handler* is evaluated. The *handler* is given an NSException as argument. The result is either the result of the evaluation of the receiver if no exception is raised, or the result of the evaluation of *handler*.

Example:

```
"Will play a beep and log the NSException raised by the division by zero"

[1/0] onException:[:e| sys beep. sys log:e]
```

return

-(id) **return**

Precondition: The receiver is being executed.

Stops the current execution of the receiver. The result of the current execution is an FSVoid object.

return:

-(void) **return:**(id)*returnValue*

Precondition: The receiver is being executed.

Stops the current execution of the receiver. The result of the current execution is *returnValue*.

C code	F-Script Code

`int search(int searched)`
`{`
`  int i = 0;`
`  while(1)`
`  {`
`  if (a[i] == searched) return i;`
`  i++;`
`  }`
`}`
	`b := [:searched| |i|`
	`  i := 0.`
	`  [true] whileTrue:`
	`  [`
	`  (a at:i) = searched ifTrue:[b return:i].`
	`  i := i+1.`
	`  ]`
	`]`

setValue:

-(void) **setValue:** (id)*operand*

Precondition: *operand* is a Block

Sets the value of the receiver to the *operand* value. The receiver becomes an exact copy of *operand.*

value

-(id) **value**

Executes the receiver and returns the result.

value:

-(id) **value:**(id)*arg1*

Executes the receiver, using *arg1* as an argument and returns the result.

value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2*

Executes the receiver, using *arg1* and *arg2* as arguments and returns the result.

value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3*

Executes the receiver, using *arg1*, *arg2* and *arg3* as arguments and returns the result.

value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4*

Executes the receiver, using *arg1*, *arg2, arg3* and *arg4* as arguments and returns the result.

value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4* and *arg5* as arguments and returns the result.

value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5* and *arg6* as arguments and returns the result.

value:value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6* **value:**(id)*arg7*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5, arg6* and *arg7* as arguments and returns the result.

value:value:value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6* **value:**(id)*arg7* **value :**(id)*arg8*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5, arg6*, *arg7* and *arg8* as arguments and returns the result.

value:value:value:value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6* **value:**(id)*arg7* **value:**(id)*arg8* **value:**(id)*arg9*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5, arg6*, *arg7, arg8* and *arg9* as arguments and returns the result.

value:value:value:value:value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6* **value:**(id)*arg7* **value:**(id)*arg8* **value:**(id)*arg9* **value:**(id)*arg10*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5, arg6*, *arg7, arg8, arg9* and *arg10* as arguments and returns the result.

value:value:value:value:value:value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6* **value:**(id)*arg7* **value:**(id)*arg8* **value:**(id)*arg9* **value:**(id)*arg10* **value:**(id)*arg11*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5, arg6*, *arg7, arg8, arg9, arg10* and *arg11* as arguments and returns the result.

value:value:value:value:value:value:value:value:value:value:value:value:

-(id) **value:**(id)*arg1* **value:**(id)*arg2* **value:**(id)*arg3* **value:**(id)*arg4* **value:**(id)*arg5* **value:**(id)*arg6* **value:**(id)*arg7* **value:**(id)*arg8* **value:**(id)*arg9* **value:**(id)*arg10* **value:**(id)*arg11* **value:**(id)*arg12*

Executes the receiver, using *arg1*, *arg2, arg3*, *arg4, arg5, arg6*, *arg7, arg8, arg9, arg10, arg11* and *arg12* as arguments and returns the result.

valueWithArguments:

-(id) **valueWithArguments:**(NSArray *)*operand*

Executes the receiver with arguments provided in *operand* and returns the result.

whileFalse

-(id) **whileFalse**

Repeatedly evaluates the receiver as long as it evaluates to false.

whileFalse:

-(id) **whileFalse:**(Block*)*operand*

Repeatedly evaluates *operand* as long as the receiver evaluates to false.

whileTrue

-(id) **whileTrue**

Repeatedly evaluates the receiver as long as it evaluates to true.

whileTrue:

-(id) **whileTrue:**(Block*)*operand*

Repeatedly evaluates *operand* as long as the receiver evaluates to true.

C code

```
i = 0;
while (i < 100)
{
    i = i +1;
}
```

F-Script code

```
i := 0.
[i < 100] whileTrue:
[
    i := i+1.
]
```

29 FSBoolean *(F-Script User Object)*

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	FScript/FSBoolean.h

Class Description

FSBoolean is a semi-abstract class which represents Booleans in F-Script. F-Script provides a literal notation for creating FSBoolean objects. This class provides the traditional Boolean operations and supports the control structure style of F-Script.

Boolean literals in F-Script are **true**, **false**, **YES**, **NO**.

FSBoolean has two concrete subclasses, True and False. Each has a single instance. The instance of the True class is referred to as **true**. The instance of the False class is referred to as **false**.

The FSBoolean class implements a small number of generic methods which apply to both **true** and **false**. The other methods are just declared in this class and implemented by the True and False concrete subclasses. You should never instantiate either FSBoolean class, or the True and False classes. Instead, you must use the **true** and **false** predefined objects (these can be obtained programmatically using the **+fsTrue, +fsFalse** and **+booleanWithBool:** class methods).

Method Types

Obtaining FSBoolean instances	+ booleanWithBool: + fsTrue + fsFalse
Testing equality	-isEqual:
Logical operations	& | and: not or:
Control structures	-ifFalse: -ifFalse:ifTrue: -ifTrue: -ifTrue:ifFalse:
Copying	-clone
Obtaining string representations	-description

Additional support	+ <

Class Methods

booleanWithBool:

+ (FSBoolean *) **booleanWithBool:**(BOOL)*theBool*

Returns the **true** object if *theBool* is true.
Returns the **false** object if *theBool* is false.

fsTrue

+ (FSBoolean *) **fsTrue**

Returns the **true** object.

fsFalse

+ (FSBoolean *) **fsFalse**

Returns the **false** object.

Instance Methods

& *(operator_ampersand:)*

-(FSBoolean*) **&** (FSBoolean*)*operand*

The logical AND operator. Returns **true** if the receiver and "*operand* value" are both true. Otherwise, returns **false**.

| *(operator_bar:)*

-(FSBoolean*) **|** (FSBoolean*)*operand*

The logical OR operator. Returns **false** if the receiver and "*operand* value" are both false. Otherwise, returns **true**.

< *(operator_less:)*

-(FSBoolean*) **<** (FSBoolean *)*operand*

If the receiver is false and *operand* is true returns **true**, else returns **false**.

+ *(operator_plus:)*

-(NSNumber*) **+** (id)*operand*

Precondition: *operand* = **true** | (*operand* = **false**)

Returns a number whose value is given in the table below:

receiver	*operand*	returned value
false	false	0
false	true	1
true	false	1
true	true	2

and:

-(FSBoolean*) **and:**(Block*)*operand*

Precondition: receiver = **false** | (*operand value* = **true** | (*operand value* **= false**))

"Short circuit" logical AND. If the receiver is **false**, returns **false**. Otherwise, returns the Boolean result of sending the "value" message to *operand*. If the result of sending "value" to *operand* is not a Boolean, an error is raised.

clone

-(FSBoolean*) **clone**

Returns the receiver, as there is only one instance for each Boolean value.

description

-(NSString*) **description**

Returns a string representation of the receiver: "true" or "false."

ifFalse:

-(id) **ifFalse:**(Block *)*falseBlock*

If the receiver is **false** returns the result of sending the message "value" to *falseBlock*. If the receiver is **true,** returns **nil.**

ifFalse:ifTrue:

-(id) **ifFalse:**(Block *)*falseBlock* **ifTrue:**(Block *)*trueBlock*

If the receiver is **false**, returns the result of sending the message "value" to *falseBlock*. If the receiver is **true** returns the result of sending the message "value" to *trueBlock*.

ifTrue:

-(id) **ifTrue:**(Block *)*trueBlock*

If the receiver is **true** returns the result of sending the message “value” to *trueBlock*. If the receiver is **false,** returns **nil.**

ifTrue:ifFalse:

-(id) **ifTrue:**(Block *)*trueOprerand* **ifFalse:**(Block *)*falseBlock*

If the receiver is **true**, returns the result of sending the message “value” to *trueBlock*. If the receiver is **false** returns the result of sending the message “value” to *falseBlock*.

isEqual:

- (BOOL) **isEqual:***object*

Returns YES if the receiver and *object* represent the same boolean value, otherwise returns NO.

not

-(FSBoolean*) **not**

Returns **false** if the receiver is **true**, otherwise returns **true**.

or:

-(FSBoolean*) **or:** (Block *)*operand*

Precondition: receiver = **true** | (*operand value* = **true** | (*operand value* **= false**))

“Short circuit” logical OR. If the receiver is **true**, returns **true**. Otherwise, returns the Boolean result of sending the message “value” to *operand*. If the result of sending “value” to *operand* is not a Boolean, an error is raised.

30 FSGenericPointer *(F-Script User Object)*

Inherits From:	FSPointer
Conforms To:	NSObject (NSObject)
Declared In:	FScript/FSGenericPointer.h

Class Description

An FSGenericPointer denotes a memory location.

An FSGenericPointer object contains a C pointer and a string describing the type of the referenced data, using the Objective-C run-time type encoding system. For instance, the type of a Pointer containing an int * is "i" (because "i" is the Objective-C type encoding for an int).

FSGenericPointers can be dereferenced using the `at:` and `at:put:` methods. Note that FSGenericPointers of type "v", which represent "void *" pointers, cannot be dereferenced. However, it is possible to change the type of an FSGenericPointer with the `setType:` method.

Note: due to a limitation in the current Objective-C run-time, invoking, from F-Script, a method whose return type is "unsigned char *" yields a Pointer object with a type of "c" instead of "C". This means that the resulting Pointer object behave as if it were based on a "char *" pointer instead of "unsigned char *". This might lead to incorrect results when dereferencing the Pointer object, because the referenced unsigned char(s) will be treated like signed char(s).

Method Types

Freeing	- free
Dereferencing	- at: - at:put:
Setting the type of a pointer	- setType:

Instance Methods

at:

-(id) **at:**(id)*index*

Precondition: *index* is an NSNumber representing a positive integer and is a valid offset for dereferencing the pointer.

Dereferences the pointer, using *index* as the offset, and returns the value of the corresponding memory zone. If this value is not an object, a mapping occurs automatically in order to return an object representing the value. The mapping follows the common F-Script mapping rules (cf.

section 15.2), except for "char *" pointers: when a "char *" pointer is dereferenced, the result is an NSNumber containing the value of the pointed char.

Example:

In this example, we assume we have an object named **myObject**, which has a method called **intPointer** which returns a value of type (int *). That is, the method returns the address of an array of int. By executing the instruction `myPointer := myObject intPointer`, we generate an FSGenericPointer object representing the int * returned by our method and assign it to the variable **myPointer**. Then to get the value of the first int referenced by the pointer, we can execute: `myPointer at:0`, which returns an NSNumber. To get the value of the second int referenced by the pointer we execute `myPointer at:1` , etc.

at:put:

-(id) **at:**(id)*index* **put:**(id)*elem*

Precondition: *index* is an NSNumber representing a positive integer and is a valid offset for dereferencing the pointer. *elem* is compatible with the receiver's type.

Dereferences the pointer, using *index* as the offset, and write the value of *elem* in the corresponding memory zone. If needed, a mapping occurs from *elem* to a non-object type. The mapping follows the common F-Script mapping rules (cf. section 15.2).

free

-(void) **free**

Free the memory zone referenced by the receiver, using the C free() function.

setType:

-(void) **setType:**(NSString *)*theType*

Precondition: *theType* is a valid Objective-C run-time type encoding string.

Set the type of the receiver to *theType*.

31 FSNSArray *(F-Script User Category)*

Category of: NSArray

Declared In: FScript/FSNSArray.h

Category Description

This category of NSArray adds a number of methods to the standard NSArray class, in order to support F-Script's array programming model. See also the FSNSMutableArray category.

Method Types

Indexing	-at: -replicate:
Getting only distinct elements	-distinct -distinctId
Copying	-clone
Getting indices	-index
Set operations	-intersection: -union: -difference:
Reductions	\ -scan:
Equality of elements	= ~=
Joining	><
Rotating	-rotatedBy:
Reversing	-reverse
Concatenation	++
Searching	! !!
Sorting	-sort
Inspecting	-inspectWithSystem: -inspectWithSystem:blocks:

Obtaining string representations	-printString
Miscellaneous	-prefixes -subpartsOfSize: -transposedBy:

Instance Methods

\ *(operator_backslash:)*

-(id) **** (Block *)*operand*

Precondition: *operand* is a Block with two arguments.

The "reduction" operation applies the *operand* block cumulatively to all the receiver elements and returns the result. More precisely, it evaluates *operand* using the two first elements of the receiver as arguments, then evaluates *operand* again using the result of the previous evaluation and the next receiver element as an argument, and does so until the last receiver element has been reached.

```
{1,2,3,4,5} \ #+
```

is executed as if it were

```
1+2+3+4+5
```

and returns `15`.

When the receiver is empty, it returns **nil**.
When the receiver has only one element, it returns this element.

= *(operator_equal:)*

-(id) **=** (id)*operand*

Sends the **=** message to all the receiver elements, using operand as an argument (or each operand element if it is an array), and returns the results in an array.

```
{1,2,3,4,5} = {1,2,3,10,11}  returns {true, true, true, false, false}
{1,2,3,4,5} = 5              returns {false, false, false, false, true}
```

This method redefines the **=** method inherited from FSNSObject.

~= *(operator_tilde_equal:)*

-(id) **~=** (id)*operand*

Sends the **~=** message to all the receiver elements, using operand as an argument (or each operand element if it is an array), and returns the results in an array.

```
{1,2,3,4,5}~= {1,2,3,10,11}  returns {false, false, false, true, true}
{1,2,3,4,5}~= 5              returns {true, true, true, true, false}
```

This method redefines the **~=** method inherited from FSNSObject.

>< ***(operator_greater_less:)***

-(Array *) **><** (NSArray *)*operand*

The "join" operation. For each receiver element, for example *e*, this method computes an Array containing the positions of *e* in *operand.* Returns all these Arrays packed into an Array of Arrays. This method involves comparisons between the receiver elements and the *operand* ones; these comparisons are made using the notion of equality (i.e. **isEqual:**), not the notion of identity (i.e. **==**).

Example:

`{1,2,'foo'} >< {4,'foo',1,'foo','foo'}` returns `{{2},{},{1,3,4}}`

++ ***(operator_plus_plus:)***

-(Array *) **++** (NSArray *)*operand*

The concatenation method. Returns an array which is the concatenation of the receiver and *operand*. The receiver and *operand* remain unchanged.

`{1,2,3} ++ {4,5}` returns `{1,2,3,4,5}`

! ***(operator_exclam:)***

-(NSNumber *) **!** (id)*operand*

Returns the index of *operand* in the receiver. The notion of equality (i.e. **isEqual:**) is used for comparisons between *operand* and the receiver elements. Only the first occurrence of *operand* in the receiver is taken into account. If *operand* is not found in the receiver, the first illegal index of the receiver is returned.

`{10,11,12,13,14} ! 11` returns `1`
`{10,11,12,13,14} ! 20` returns `5`

!! ***(operator_exclam_exclam:)***

-(NSNumber *) **!!** (id)*operand*

Same as **!** but uses the notion of identity (i.e. pointer equality) for the comparisons.

at:

-(id) **at:**(id)*index*

Precondition: *index* is an integer in the range [0,receiver count - 1] or an array of integers in the range [0,receiver count - 1] or an NSIndexSet containing integers in the range [0,receiver count - 1] or an array of Booleans of the same size as the receiver.

If *index* is a number, this method returns the element of the receiver whose index is equal to *index*. Note that indexing starts at zero.

If *index* is an array of numbers or an NSIndexSet, this method returns, in an array, the elements of the receiver whose indices are given in *index*.

If *index* is an array of Booleans, this method returns, in an array, the elements of the receiver whose positional matching element in *index* is **true**.

```
{10,11,12,13,14} at:1                              returns  11
{10,11,12,13,14} at:{1,4}                          returns  {11,14}
{10,11,12,13,14} at:{1,4,4}                        returns  {11,14,14}
{10,11,12,13,14} at:{false,true,true,false,true}   returns  {11,12,14}
```

clone

-(Array *) **clone**

Returns an auto-released copy of the receiver.

difference:

-(Array *) **difference**:(NSArray *)*operand*

Returns, in an array, the receiver elements which are not elements of *operand*.
The notion of equality is used for comparing.

```
{1,4,4,4,5} difference:{4,5,6,7,8}   returns  {1}
```

distinct

-(Array *) **distinct**

Returns an array referencing, only once, each object referenced by the receiver.
Use this method to get the different elements of an array.
The notion of equality (i.e. **`isEqual:`**) is used for the comparisons.
The order of elements in the receiver is not preserved in the returned array.

```
{1,2,2,1,1,1,3} distinct     returns     {1,3,2}
```

distinctId

-(Array *) **distinctId**

Same as **distinct** but uses the notion of identity for the comparisons.

index

-(Array *) **index**

Returns, in an array, all the valid indices of the receiver, in ascending order.

```
{10,11,12,'foo'} index
```
returns
```
{0,1,2,3}
```

inspectWithSystem:

-(void) **inspectWithSystem:**(System *)*system*

Open a graphical inspector displaying the elements of the receiver. When the inspector creates a block, it does so in the context of the workspace associated with the argument. Typically, when you call this method from F-Script, you pass the current system object. Example:
```
{1,2,3,4} inspectWithSystem:sys
```

inspectWithSystem:with:

-(void) **inspectWithSystem:**(System *)*system* **blocks:** (NSArray *)*blocks*

Precondition: *blocks* is an array of one-arguments or zero-argument blocks.

Open a graphical inspector displaying the elements of the receiver. When the inspector creates a block, it does so in the context of the workspace associated with *system*. The inspector uses the blocks you provides to compute its column's values.

intersection:

-(Array *) **intersection**:(NSArray *)*operand*

Returns, in an array, the elements referenced by both the receiver and *operand*.
The notion of equality is used for comparing.

```
{1,4,4,4,5} intersection:{4,5,6,7,8}
```
returns
```
{4,5}
```

prefixes

-(Array *) **prefixes**

Returns, in an array of array, all the prefixes of the receiver. Each sub-array of the result is a prefix. As this method has an O(n2) complexity, you should use it carefully.

`{1,2,3,4} prefixes` returns `{{1},{1,2},{1,2,3},{1,2,3,4}}`

printString

-(NSString*) **printString**

Returns a string representation of the receiver, in the syntactic form of an array literal.

replicate:

-(Array *) **replicate:**(NSArray *)*operand*

Precondition: *operand* is an array, with the same size as the receiver, of positive integers.

Each *operand* element represents the number of references to the corresponding receiver element which will be inserted in the returned array.

`{10,11,12,13,14} replicate:{0,1,2,3,1}` returns `{11,12,12,13,13,13,14}`

reverse

-(Array *) **reverse**

Returns an array with the same elements as the receiver, but in reverse order.

`{1,2,3,4} reverse` returns `{4,3,2,1}`

rotatedBy:

-(void) **rotatedBy:**(NSNumber *)*operand*

Returns a circular permutation of the receiver. The permutation is made to the left or to the right, depending on the *operand* sign.

`{1,2,3,4,5,6} rotatedBy:2` returns `{3,4,5,6,1,2}`

`{1,2,3,4,5,6} rotatedBy:-2` returns `{5,6,1,2,3,4}`

scan:

-(Array *) **scan:**(Block *)*operand*

Precondition: *operand* is a Block with two arguments.

The same as the reduction operation but returns an array with all the intermediate results computed during the reduction.

```
{1,2,3,4,5} scan: #+    returns    {1,3,6,10,15}
```

sort

-(Array *) **sort**

Precondition: The elements of the receiver implement the **operator_less:** method to provide a total order.

Returns, in an array, the indices that will arrange the receiver in ascending order. The **sort** method is stable, that is, if two members compare as equal, the order of their indices in the returned array is preserved.

```
{12,15,13,11,14} sort    returns    {3,0,2,4,1}
```

subpartsOfSize:

-(Array *) **subpartsOfSize:**(NSNumber *)*size*

Precondition: *size* is a non-negative integer.

Returns, in an array, all the receiver subparts of size *size*.
As this method has an O(n2) complexity, you should use it carefully.

```
{1,2,3,4} subpartsOfSize:0   returns   {}
{1,2,3,4} subpartsOfSize:1   returns   {{1},{2},{3},{4}}
{1,2,3,4} subpartsOfSize:2   returns   {{1,2},{2,3},{3,4}}
```

transposedBy:

-(Array *) transposedBy:(NSArray *)*operand*

Precondition: The receiver is a hypercube and *operand* is an array and contains a permutation of the integers in range [1..number of dimensions of the receiver].

The transposition operation.

Let's begin by explaining the concept of hypercube.
F-Script does not explicitly cater for multidimensional arrays. An F-Script array is a mono-dimensional array, a list. As an array can contain other arrays, you "simulate" multidimensional arrays using arrays of array. F-Script uses the term "hypercube" to describe a special kind of such a simulated multi-dimensional array. Note that some arrays of array are not hypercubes. For example, the array **{{11,12,13}, {21,22,23}}** is a hypercube (in this case a 2*3 matrix), but

the array **{{11,12,13},{21,22}}** is not a hypercube as one dimension does not have a constant number of elements.
An order is defined on the dimensions of an hypercube; for example, in the hypercube **{{11,12,13},{21,22,23}}**, the first dimension is those having two elements and the second is those having three elements.

This method computes and returns a hypercube which is a transposition of the receiver according to the transposition vector passed as *operand*. A transposition consists in restructuring a hypercube so that its coordinates appear in a permuted order. The element of the transposition vector at index **i** specifies the dimension of the receiver which becomes the dimension **i** in the result. For example, **{1,2,0}** as a transposition vector specifies that the first dimension of the result is the second dimension of the receiver, that the second dimension of the result is the third dimension of the receiver and that the third dimension of the result is the first dimension of the result.

```
{{11,12,13}, {21,22,23}} transposedBy:{1,0}
```
returns `{{11,21}, {12, 22}, {13, 23}}`

union:

-(Array *) **union**:(NSArray *)*operand*

Returns, in an array, the union of the elements referenced by the receiver and *operand*. The notion of equality is used for comparing.

`{1,4,4,4,5} union:{4,5,6,7,8}` returns `{1,4,5,6,7,8}`

32 FSNSAttributedString (F-Script User Category)

Category of: NSAttributedString

Declared In: FScript/FSNSAttributedString.h

Category Description

This category provides inspector support to NSAttributedString objects.

Instance Methods

inspect

-(void) **inspect**

Opens an inspector displaying the receiver.

33 FSNSDate *(F-Script User Category)*

Category of: NSDate

Declared In: FScript/FSNSDate.h

Category Description

This category provides some user methods for use with NSDate objects. For Date creation, see the methods `asDate` in FSNSString and FSNSNumber.

Instance Methods

clone

-(NSDate *) **clone**

Returns an auto-released copy of the receiver.

max:

-(NSDate *) **max:**(NSDate *)*operand*

Returns the maximum of the receiver and *operand.*

min:

-(NSDate *) **min:**(NSDate *)*operand*

Returns the minimum of the receiver and *operand.*

> ***(operator_greater:)***

-(FSBoolean*) **>** (NSDate *)*operand*

If the receiver is greater than *operand*, returns the Boolean object **true**, otherwise returns the Boolean object **false**.

>= ***(operator_greater_equal:)***

-(FSBoolean*) **>=** (NSDate *)*operand*

If the receiver is greater than or equal to *operand*, returns the Boolean object **true**, otherwise returns the Boolean object **false**.

< ***(operator_less:)***

-(FSBoolean*) **<** (NSDate *)*operand*

If the receiver is less than *operand*, it returns the Boolean object **true**, otherwise returns the Boolean object **false**.

<= ***(operator_less_equal:)***

-(FSBoolean*) **<=** (NSDate *)*operand*

If the receiver is less than or equal to *operand*, it returns the Boolean object **true**, otherwise returns the Boolean object **false**.

- ***(operator_hyphen:)***

-(NSNumber*) **-** (NSDate *)*operand*

Returns the difference in seconds between the receiver and *operand*.

34 FSNSFont *(F-Script User Category)*

Category of: NSFont

Declared In: FScript/FSNSFont.h

Category Description

This category provides inspector support to NSFont objects.

Instance Methods

inspect

-(void) **inspect**

Opens an inspector displaying the receiver.

35 FSNSImage *(F-Script User Category)*

Category of: NSImage

Declared In: FScript/FSNSImage.h

Category Description

This category provides inspector support to NSImage objects.

Method Types

inspecting -inspect

Instance Methods

inspect

-(void) **inspect**

Opens an inspector displaying the receiver.

36 FSNSManagedObjectContext *(F-Script User Category)*

Category of: NSManagedObjectContext

Declared In: FScript/FSNSManagedObjectContext.h

Category Description

This category provides inspector support to NSManagedObjectContext objects.

Method Types

inspecting -inspectWithSystem:

Instance Methods

inspectWithSystem:

-(void) **inspectWithSystem:**(System *)*system*

Open a graphical inspector displaying the receiver. When the inspector creates a block, it does so in the context of the workspace associated with the argument. Typically, when you call this method from F-Script, you pass the current system object. Example: `aManagedObjectContext inspectWithSystem:sys`

37 FSNSMutableArray *(F-Script User Category)*

Category of: NSMutableArray

Declared In: FScript/FSNSMutableArray.h

Category Description

This category of NSMutableArray adds a number of methods to the standard NSMutableArray class, in order to support F-Script's array programming model. See also the FSNSArray category.

Method Types

Adding elements	-add: -insert:at :
Removing elements	-removeAt:
Setting elements	-at:put: -setValue:

Instance Methods

add:

-(void) **add:**(id)*elem*

Inserts *elem* at the end of the receiver.

at:put:

-(id) **at:**(id)*index* **put:**(id)*elem*

Precondition: *index* is an integer in the range [0,receiver count - 1] or (*index* is an array or an NSIndexSet containing integers in the range [0,receiver count - 1] and (*elem* is an array of the same size or *elem* is not an array) or (index is an array of Booleans of the same size as the receiver and (*elem* is an array of size s, where s is equal to the number of Booleans with a value of true in index or elem is not an array).

If *index* is a number, this method replaces the element of the receiver stored at *index* by *elem*.

If *index* is an array of numbers or an NSIndexSet, this method replaces the elements of the receiver specified by *index* by the corresponding elements in *elem* or by *elem* if *elem* is not an array.

If *index* is an array of Booleans, this method replaces the elements of the receiver whose positional matching element in *index* is **true** by the corresponding elements in *elem* or by *elem* if *elem* is not an array.

This method returns *elem.*

insert:at:

-(void) **insert**:(id)*elem* **at:**(NSNumber *)*index*

Precondition: *index* is an integer in the range [0,receiver count].

Inserts *elem* into the receiver at *index*, moving other elements to make room if needed.

removeAt:

-(void) **removeAt:**(id)*index*

Precondition: *index* is an integer in the range [0,receiver count - 1] or an array of integers in the range [0,receiver count - 1] or an NSIndexSet containing integers in the range [0,receiver count - 1] or an array of Booleans of the same size as the receiver.

If *index* is a number, this method removes the element located at *index*.

If *index* is an array of numbers or an NSIndexSet, this method removes the elements of the receiver whose indices before the operation are given in *index*.

If *index* is an array of Booleans, this method removes the elements of the receiver whose positional matching element in *index* before the operation is **true**.

The positions of the remaining elements in the receiver are adjusted to fill the gaps.

setValue:

-(void) **setValue:** (id)*operand*

Precondition: *operand* is an array.

Sets the value of the receiver to that of *operand*. The receiver becomes an exact copy of *operand.*

38 FSNSMutableString *(F-Script User Category)*

Category of: NSMutableString

Declared In: FScript/FSNSString.h

Category Description

This category complements the NSMutableString class with user methods for string manipulations in F-Script.

Method Types

Adding characters	-insert:at :
Copying	-setValue: -clone

Instance Methods

clone

-(NSString *) **clone**

Returns an auto-released copy of the receiver. The returned object will be an NSMutableString.

insert:at:

-(void) **insert:**(NSString *)*str* **at:**(NSNumber *)*index*

Precondition: *index* is an integer in the range [0,receiver length].

Inserts the substring *str* into the receiver at *index*, moving other characters to make room if necessary.

setValue:

-(void) **setValue:** (id)*operand*

Precondition: *operand* is a NSString.

Sets the value of the receiver to the value of *operand*. The receiver becomes an exact copy of *operand.*

39 FSNSNumber *(F-Script User Category)*

Category of: NSNumber

Declared In: FScript/FSNSNumber.h

Class Description

This category of NSNumber provides number support.

F-Script provides a literal notation for number objects. The syntax for number literal is:
{-}<integerPart>{.<decimalPart>}{exponentLetter<exposantIntegerPart>{.<exposantDecimalPar>}}

With exponentLetter ::= 'e' | 'd' | 'q'

Some examples of number literal are: **1** , -**3.14**, **1.23e-2**

The F-Script literal notation generates numbers in double precision.

Operations provided by this category are performed in double precision.

Method Types

Basic operations	+ - * /
Mathematical functions	-abs -arcCos -arcSin -arcTan -cos -cosh -exp -ln -log -negated -raisedTo: -sin -sign -sinh -sqrt -tan -tanh
Getting maximum and minimum of two numbers	-max: -min:
Getting the remainder	-rem:

Rounding up numbers	-ceiling -floor -truncated
Accessing numeric values	-fractionPart -integerPart
Comparing numbers	> >= < <= -between:and:
Bit manipulation	-bitAnd: -bitOr: -bitXor:
Converting numbers	-asDate -unicharToString
Generating random numbers	-random -random: -seedRandom
Generating arrays	-iota
Control structures	-timesRepeat: -to:do: -to:by:do:
Generating points	<>
Copying	-clone

Instance Methods

+ ***(operator_plus:)***

-(NSNumber*) **+** (id *)*operand*

Precondition: *operand* is a NSNumber or a FSBoolean

If *operand* is a number, returns the sum of the receiver and *operand*. If operand is **true**, returns the sum of the receiver and 1. If operand is **false**, returns the sum of the receiver and 0.

- ***(operator_hyphen:)***

-(NSNumber*) **-** (NSNumber *)*operand*

Returns the difference between the receiver and *operand*.

***** ***(operator_asterisk:)***

-(NSNumber*) ***** (NSNumber *)*operand*

Returns the product of the receiver and *operand*.

/ *(operator_slash:)*

-(NSNumber*) **/** (NSNumber *)*operand*

Precondition: *operand* ~= 0

Returns the quotient of the receiver and *operand*.

> *(operator_greater:)*

-(FSBoolean*) **>** (NSNumber *)*operand*

If the receiver is greater than *operand*, returns **true**, otherwise returns **false**.

>= *(operator_greater_equal:)*

-(FSBoolean*) **>=** (NSNumber *)*operand*

If the receiver is greater than or equal to *operand*, returns **true**, otherwise returns **false**.

< *(operator_less:)*

-(FSBoolean*) **<** (NSNumber *)*operand*

If the receiver is less than *operand*, returns **true**, otherwise returns **false**.

<= *(operator_less_equal:)*

-(Boolean*) **<=** (NSNumber *)*operand*

If the receiver is less or equal than *operand*, returns **true**, otherwise returns **false**.

<> *(operator_less_greater:)*

-(NSPoint) **<>** (NSNumber *)*operand*

Precondition: receiver and operand must be in the range [-FLT_MAX, FLT_MAX].

Returns a point whose coordinates are the receiver value for x and *operand* value for y.

abs

-(NSNumber*) **abs**

Returns the absolute value of the receiver.

arcCos

-(NSNumber*) **arcCos**

Precondition: receiver between:-1 and:1

Returns the arccosinus of the receiver (a number whose cosinus is equal to the receiver) in radians. The result of this method is in the range [0 , pi].

arcSin

-(NSNumber*) **arcSin**

Precondition: receiver between:-1 and:1

Returns the arcsinus of the receiver (a number whose sinus is equal to the receiver) in radians. The result of this method is in the range [-pi/2 , pi/2].

arcTan

-(NSNumber*) **arcTan**

Returns arctangent of the receiver (a number whose tangent is equal to the receiver) in radians. The result of this method is in the range [-pi/2 , pi/2].

asDate

-(NSDate*) **asDate**

Takes the receiver as a time interval in seconds and returns a NSDate object representing the date with this time interval from a system reference Date.

bitAnd:

-(NSNumber *)**bitAnd:**(NSNumber *)*operand*

Precondition: receiver and *operand* must be integers in the range [0, UINT_MAX].

Answer the result of the bit-wise logical AND of the binary representation of the receiver and the binary representation of *operand*.

bitOr:

-(NSNumber *)**bitOr:**(NSNumber *)*operand*

Precondition: receiver and *operand* must be integers in the range [0, UINT_MAX].

Answer the result of the bit-wise logical OR of the binary representation of the receiver and the binary representation of *operand*.

bitXor:

-(NSNumber *)**bitXor:**(NSNumber *)*operand*

Precondition: receiver and *operand* must be integers in the range [0, UINT_MAX].

Answer the result of the bit-wise exclusive OR of the binary representation of the receiver and the binary representation of *operand*.

between:and:

-(FSBoolean *) **between:**(NSNumber *)*inf* **and:**(NSNumber *)*sup*

Returns **true** if the receiver is greater than or equal to *inf* and less than or equal to *sup,* otherwise returns **false**. Note that "**a between:b and:c**" differs from "**a >= b & (a <= c)**" and from "**a >= b & [a <= c]**" in that it prevents **a** from being evaluated twice, which is important if **a** is a complex or not-side-effect-free expression.

ceiling

-(NSNumber*) **ceiling**

Returns the smallest integer that is greater than or equal to the receiver.

clone

-(NSNumber*) **clone**

Returns an auto-released copy of the receiver.

cos

-(NSNumber*) **cos**

Returns the cosinus of the receiver in radians.

cosh

-(NSNumber*) **cosh**

Returns the hyperbolic cosinus of the receiver in radians.

exp

-(NSNumber*) **exp**

Returns the natural logarithm base **e** raised to the power of the receiver.

floor

-(NSNumber*) **floor**

Returns the greatest integer that is less than or equal to the receiver.

fractionPart

-(NSNumber*) **fractionPart**

Returns the fraction part of the receiver (e.g., the fraction part of 12.3456e2 is 0.56).

integerPart

-(NSNumber*) **integerPart**

Returns the integer part of the receiver (e.g., the integer part of 12.3456e2 is 1234).

iota

-(Array *) **iota**

Precondition: receiver >= 0 & receiver <= UINT_MAX

Returns an array with all the integers between 0 and the receiver-1. For example, "**5 iota**" returns **{0,1,2,3,4}**.

Generates an error (i.e. raise an exception) if there is not enough memory to complete the operation.

ln

-(NSNumber*) **ln**

Precondition: receiver > 0

Returns the natural logarithm of the receiver.

log

-(NSNumber*) **log**

Precondition: receiver > 0

Returns the logarithm base 10 of the receiver.

max:

-(NSNumber*)**max:**(NSNumber *)*operand*

Returns the receiver if it is greater than *operand*. Otherwise returns *operand*.

min:

-(NSNumber*)**min:**(NSNumber *)*operand*

Returns the receiver if it is less than *operand*. Otherwise returns *operand*.

negated

-(NSNumber*)**negated**

Returns a number equal to zero minus the receiver.

random

-(NSNumber*)**random**

Precondition: receiver >= 1 & (receiver < 2147483648)
(note: 2147483648 is equal to 2 raisedTo:31)

Returns a random integer between 0 and the receiver - 1 (e.g., "`5 random`" will return either 0,1, 2, 3, or 4).

Internally, this method use the random() UNIX function as a source of pseudo-random numbers.

See also: -random: -seedRandom

random:

-(Array*)**random:**(NSNumber *)*operand*

Precondition: (receiver >= o*perand*) & (*operand* between:0 and:UINT_MAX) & (receiver fractionPart = 0) & (*operand* fractionPart = 0)

Returns an array of size equal to *operand*, of random integers between 0 and the receiver - 1. All the elements of the result have a different value and are returned in a random order.

Internally, this method use the random() and drand48() UNIX functions as a source of pseudo-random numbers.

Example: give me five different integers randomly chosen in the interval [0,9].

```
> 10 random:3
{4, 9, 2}
```

Example: give me a random permutation of the integers between [0,9].

```
> 10 random:10
{2, 4, 0, 5, 3, 6, 1, 9, 7, 8}
```

See also: -random -seedRandom

raisedTo:

-(NSNumber*) **raisedTo:** (NSNumber *)*operand*

Precondition: (receiver = 0 & (*operand* <= 0)) not & (receiver < 0 & (*operand* fractionPart ~= 0)) not

Returns the receiver raised to the power *operand*.

rem:

-(NSNumber*)**rem:**(NSNumber *)*operand*

Precondition: *operand* ~= 0

Returns the remainder of the division of the receiver by *operand*. The sign of the remainder is the same sign as the receiver.

seedRandom

-(NSNumber*)**seedRandom**

Precondition: receiver <= ULONG_MAX

Sets the value of the receiver as the seed for a new sequence of pseudo-random numbers to be used by -random and -random: to compute their results. These sequences are repeatable by invoking -seedRandom on the same seed value.

This method calls the srand48() and srandom() UNIX functions as part of its implementation. Note that, at launch time, the F-Script application automatically initialize the random() and drand48() seeds with random values. This automatic initialization does not take place when using the F-Script framework in another application.

See also: -random -random:

sin

-(NSNumber*) **sin**

Returns the sinus of the receiver in radians.

sign

-(NSNumber*) **sign**

Answer the sign of the receiver: answer 1 if the receiver is positive, 0 if the receiver equals 0, and -1 if it is negative.

sinh

-(NSNumber*) **cosh**

Returns the hyperbolic sinus of the receiver in radians.

sqrt

-(NSNumber*) **sqrt**

Precondition: receiver >= 0

Returns the square root of the receiver.

tan

-(NSNumber*) **tan**

Returns the tangent of the receiver in radians.

tanh

-(NSNumber*) **tanh**

Returns the hyperbolic tangent of the receiver in radians.

timesRepeat:

-(void) **timesRepeat:**(Block *)*operation*

Precondition: operation argumentCount = 0 & (receiver >= 0) & (receiver fractionPart = 0)

Evaluate *operation* the number of times represented by the receiver.

to:by:do:

- (void) **to:**(NSNumber *)*stop* by:(NSNumber *)*step* **do:**(Block *)*operation*

Precondition: operation argumentCount <= 1 & (step != 0)

Evaluates *operation* for each element of an interval, starting at the receiver and ending at *stop,* where each element is a *step* greater than the previous. The value of *step* may be positive or negative but it must be non-zero.
The block *operation* is given a number as argument. This number represents the current element of the iteration.

No evaluation takes place if:
1. receiver < *stop*, and *step* < 0
2. receiver > *stop*, and *step* > 0

to:do:

- (void) **to:**(NSNumber *)*stop* **do:**(Block *)*operation*

Precondition: operation argumentCount <= 1

Evaluates *operation* for each element of an interval starting at the receiver and ending at *stop,* where each element is 1 greater than the previous. No evaluation takes place if the receiver is greater than *stop*.
The block *operation* is given a number as argument, representing the current element of the iteration.

Examples:

```
"Compute into count the sum of all the integers from 1 to 10"

count := 0. 1 to:10 do:[:i| count := count + i]

"Construct in a an array with all the integers from 1 to 10"

a := {}. 1 to:10 do:[:i| a add:i]
```

truncated

-(NSNumber*) **truncated**

Answer an integer equal to the receiver truncated towards zero.

If the receiver is positive, answer the largest integer less than or equal to the receiver. If it is negative, answer the smallest integer greater or equal to the receiver.

unicharToString

- (NSString *) **unicharToString**

Precondition: *receiver* between:0 and:65535

Returns a string made up of one character whose encoding value in UNICODE is the value of the receiver. For example, `97 unicharToString` will return the string 'a' because the encoding value for the character **a** in UNICODE is 97. This method is particularly useful when dealing with Cocoa methods which return a *unichar*, as the returned *unichar* is represented by a number in F-Script.

40 FSNSObject *(F-Script User Category)*

Category of: NSObject

Declared In: FScript/FSNSObject.h

Category Description

The FSNSObject category provides various services.

Method Types

Identifying classes	classOrMetaclass
Equality and identity	= ~= == ~~
Saving onto a file	-save -save:
Enclosing in an array	-enlist -enlist:
Publishing a distributed object	-vend:
Obtaining String representations	-printString
Throwing an exception	-throw

Instance Methods

= ***(operator_equal:)***

-(id) **=** (id)*operand*

The default implementation provided by FSNSObject returns **true** if the receiver and *operand* are equal, otherwise it returns **false**. The implementation of this method provided by FSNSObject uses the **isEqual:** method to test equality. When implementing a new class, you may redefine the **isEqual:** method if needed (if you do, do not forget to redefine your **hash** method too).

Note that the arrays redefine this method so as to return an array of Booleans.

== ***(operator_equal_equal:)***

-(FSBoolean *) **==** (id)*operand*

Returns **true** if the receiver and *operand* are identical, otherwise returns **false**. Two objects are identical if they are the same objects in memory (pointer equality).

~= *(operator_tilde_equal:)*

-(id) **~=** (id)*operand*

The default implementation provided by FSNSObject returns **false** if the receiver and *operand* are equal, otherwise it returns **true**. The implementation of this method provided by FSNSObject uses the **isEqual:** method to test equality. When implementing a new class, you may redefine the **isEqual:** method if necessary (if you do, do not forget to redefine your **hash** method too).

Note that arrays redefine this method so as to return an array of Booleans.

~~ *(operator_tilde_tilde:)*

-(FSBoolean *) **~~** (id)*operand*

Returns **false** if the receiver and *operand* are identical, otherwise returns **true**. Two objects are identical if they are the same objects in memory (pointer equality).

classOrMetaclass

-(Class) **classOrMetaclass**

Return the class object for the receiver's class. This method works the same for instances and classes. This means that if the receiver is a class, the returned object will be the class of the receiver (i.e. a meta-class). This differs from the +class method, which returns the receiver itself.

enlist

-(Array *) **enlist**

Creates an array, puts the receiver into it and returns it.

Generates an error (i.e. raise an exception) if there is not enough memory to complete the operation.

enlist:

-(Array *) **enlist:**(NSNumber *)*operand*

Returns an array equal in size to *operand*, and all slots referencing the receiver. For example: `'foo' enlist:3` returns `{'foo', 'foo', 'foo'}`.

printString

-(NSString *) **printString**

Returns a string representation of the receiver. This method is called by F-Script to display objects and is also available for direct calling. To compute its result, the default implementation provided by FSNSObject use the string returned by the **description** method.

See also: -description (NSObject)

save

-(void) **save**

Asks the user for a filename, then saves the object onto this file by calling **save:**. An exception is raised if the archiving process fails (among other things, this includes failure due to I/O problems).

save:

-(void) **save:**(NSString *)*filename*

Saves the receiver on the file named *filename,* using Cocoa archiving (i.e. using NSArchiver). The archiving process should be specified by the receiver class documentation (specifically, what is archived should be specified), if it is not obvious.

An exception is raised if the archiving process fails (among other things, this includes failure due to I/O problems).

This method is based on the Cocoa archiving mechanism. Note that if an object in the object graph being archived does not conform to the Cocoa NSCoding protocol, an instance of NSNull is archived in place of this object.

throw

-(void) **throw**

Throw the receiver as an exception object.

Objective-C code	F-Script Code
`@throw myObject`	`myObject throw`

vend:

-(NSConnection *) **vend:**(NSString *)*name*

Registers the receiver under *name* in the distributed object system, and return the NSConnection instance used, or nil if the operation is impossible. The implementation of this method is conceptually equivalent to:

```
- (NSConnection *)vend:(NSString *)name
{
  NSConnection *theConnection = [[NSConnection alloc] init];

  [theConnection setRootObject:self];
  if ([theConnection registerName:name] == NO)
  {
    [theConnection release];
    return nil;
  }
  else return theConnection;
}
```

The returned NSConnection is not autoreleased. The caller is given ownership of this NSConnection and should release it when it is no longer needed.

41 FSNSString *(F-Script User Category)*

Category of: NSString

Declared In: FScript/FSNSString.h

Category Description

This category complements the NSString class.

A string literal has the form: `'I am a String'`. Such a literal generates an NSString (non-mutable). You can use `''` (two quote characters) in a string body to embed a quote.

In addition, F-Script provides C-like escape sequences. These sequences are interpreted in a special way when submitted to the F-Script interpreter in a string literal. What follows is a list of the recognized escape sequences (note that other usages of \ in a string literal submitted to the F-Script compiler are not permitted).

SEQUENCE	INTERPRETED AS	
\a	audible alert	BEL
\b	backspace	BS
\f	formfeed	FF
\n	newline	NL
\r	carriage return	CR
\t	horizontal tab	HT
\v	vertical tab	VT
\\	backslash	\
\’	quote	‘

The F-Script compiler will only accept string literals that contain ASCII characters. However, the method `unicharToString`, defined on NSNumbers by the FSNSNumber category, let you create a string made up of one character whose encoding value in UNICODE is the value of the receiver. For example, `97 unicharToString` will return the string `'a'` because the encoding value for the character a in UNICODE is 97. This method used in conjunction with the concatenation operator (i,e,. ++) makes it easy to create strings containing any UNICODE characters.

Method Types

Generating blocks from strings	-asBlock -asBlockOnError:
Indexing	at:
Comparing	> >= < <=

Array conversion	-asArray -asArrayOfCharacters
Getting classes from strings	-asClass
Creating dates from strings	-asDate
Copying	-clone
Reversing	-reverse
Concatenation	++
Obtaining string representations	-printString
Connecting to a distributed object	-connect

Instance Methods

> ***(operator_greater:)***

-(FSBoolean*) **>** (NSString *)*operand*

If the receiver is greater than *operand*, returns **true**, otherwise returns **false**.

>= ***(operator_greater_equal:)***

-(FSBoolean*) **>=** (NSString *)*operand*

If the receiver is greater than or equal to *operand*, returns **true**, otherwise returns **false**.

< ***(operator_less:)***

-(FSBoolean*) **<** (NSString *)*operand*

If the receiver is less than *operand*, returns **true**, otherwise returns **false**.

<= ***(operator_less_equal:)***

-(FSBoolean*) **<=** (NSString *)*operand*

If the receiver is less than or equal to *operand*, returns **true**, otherwise returns **false**.

++ ***(operator_plus_plus:)***

-(NSString *) **++** (NSString *)*operand*

The concatenation method. Returns a NSString which is the concatenation of the receiver and *operand*. Both the receiver and *operand* remain unchanged.

`'foo' ++ 'bar'` returns `'foobar'`

asArray

-(Array *) **asArray**

Returns an Array of strings where each string represents a glyph of the receiver. This method takes into account Unicode composed character sequences.

asArrayOfCharacters

-(Array *) **asArrayOfCharacters**

Returns an Array of strings where each string represents a character of the receiver.

asBlock

-(Block *) **asBlock**

Creates and returns a block whose source is taken from the receiver. The receiver must represent a block literal. If there is a syntax error, this method shows it in a block inspector window and generates an F-Script execution error. (See also the -blockFromString: method in class System.)

Note that each block created by this method is in charge of its own top-level workspace. The lifecycle of this workspace is closely related to the lifecycle of the block. Since it is possible for the block to create other blocks in its workspace and to enable them to be held for later execution, problems may arise if you try to execute a block whose workspace was controlled by a block that has been deallocated. Thus, you must ensure that the block in charge of a workspace is not deallocated until the workspace is no longer needed.

asBlockOnError:

-(id)**asBlockOnError:**(Block*)*errorBlock*

Creates and returns a block whose source is taken from the receiver. The receiver must represent a block literal. If there is a syntax error, *errorBlock* is executed with three arguments: a NSString with the error message, a number with the starting position of the syntax and a number with the final position of the syntax error. The result of the *errorBlock* execution is then returned. (See also the -blockFromString:onError: method in class System.)

Note that each block created by this method is in charge of its own top-level workspace. The lifecycle of this workspace is closely related to the lifecycle of the block. Since it is possible for the block to create other blocks in its workspace and to enable them to be held for later execution, problems may arise if you try to execute a block whose workspace was controlled by a block that has been deallocated. Thus, you must ensure that the block in charge of a workspace is not deallocated until the workspace is no longer needed.

asClass

-(id) **asClass**

Returns the class whose name is in the receiver, or **nil** if no class matches the receiver. This method gives access to all the classes linked to the running application.

asDate

-(NSDate *) **asDate**

Interprets the receiver as a date specification (using the natural language interpreter) and returns the relevant date object.

at:

-(id) at:(id)*operand*

Precondition: *operand* is an integer in the range [0,receiver length-1].

Returns an NSString formed by the character of the receiver at index *operand*. In F-Script, indexing starts with zero as in Objective-C.

clone

-(NSString *) **clone**

Returns an auto-released copy of the receiver.

connect

-(id) **connect**

Tries to connect to a distributed object registered in the distributed object system under a name similar to the receiver's value. This method only searches for distributed objects on the local host. The implementation of this method is conceptually equivalent to:

```
- (id) connect
{
  return [NSConnection rootProxyForConnectionWithRegisteredName:self host:nil];
}
```

max:

-(NSString *)**max:**(NSString *)*operand*

Returns the maximum of the receiver and *operand*. Does not create a new NSString object. Returns either the receiver or *operand*.

min:

-(NSString *)**min:**(NSString *)*operand*

Returns the minimum of the receiver and *operand*. Does not create a new NSString object. Returns either the receiver or *operand*.

printString

-(NSString *) **printString**

Returns an NSString representation of the receiver.

reverse

-(NSString *) **reverse**

Returns a string with the same characters as the receiver, but in reverse order.

`'bingo' reverse` returns `'ognib'`

42 FSNSValue *(F-Script User Category)*

Category of: NSValue

Declared In: FScript/FSNSValue.h

Category Description

This category complements the NSValue class. In particular, it supports NSRange, NSPoint, NSRect and NSSize manipulation.

Method Types

String representation	-printString
Copying	-clone
Range	+rangeWithLocation:length: -length -location
Point	-corner: -extent: -x -y
Rectangle	-corner -extent -origin
Size	+sizeWithWidth:height: -height -width

Class Methods

sizeWithWidth:height:

+(NSSize) **sizeWithWidth:**(float)*width* **height:**(float)*height*

Precondition: *width >= 0 & (height >= 0)*

Returns a size with the specified *width* and *height*.

rangeWithLocation:length:

+(NSRange) **rangeWithLocation:**(unsigned int)*location* **length:**(unsigned int)*length*

Returns a range with the specified *location* and *length*.

Instance Methods

clone

-(id) **clone**

Returns an auto-released copy of the receiver.

corner:

-(NSRect) **corner:**(NSPoint)*operand*

Precondition: Receiver contains an NSPoint and *operand*'s x coordinates is greater or equal to receiver's x coordinate and *operand*'s y coordinates is greater or equal to receiver's y coordinate.

Returns a rectangle whose origin is the receiver, and whose opposite corner is *operand*.

extent

-(NSPoint) **extent**

Precondition: Receiver contains an NSRect.

Returns a point that represents the extent of the receiver. The width and height of a rectangle form what we call the *extent.*

extent:

-(NSRect) **extent:**(NSPoint)*operand*

Precondition: Receiver contains an NSPoint and *operand* coordinates are non-negatives.

Returns a rectangle whose origin is the receiver, and whose width and height are provided by *operand*.

height

-(float) **height**

Precondition: Receiver contains an NSSize.

Returns the height.

length

-(unsigned int) **length**

Precondition: Receiver contains an NSRange.

Returns the length.

location

-(unsigned int) **location**

Precondition: Receiver contains an NSRange.

Returns the location.

origin

-(NSPoint) **origin**

Precondition: Receiver contains an NSRect.

Returns a point that represents the origin of the receiver.

printString

-(NSString*) **printString**

Returns a string representation of the receiver.

width

-(float) **width**

Precondition: Receiver contains an NSSize.

Returns the width.

x

-(float) **x**

Precondition: Receiver contains an NSPoint.

Returns the value of the x coordinate.

y

-(float) **y**

Precondition: Receiver contains an NSPoint.

Returns the value of the y coordinate.

43 FSObjectPointer *(F-Script User Object)*

Inherits From:	FSPointer
Conforms To:	NSObject (NSObject)
Declared In:	FScript/FSObjectPointer.h

Class Description

An FSObjectPointer denotes the location of a memory zone containing objects.

Method Types

Dereferencing	- at: - at:put:

Instance Methods

at:

-(id) **at:**(id)*index*

Precondition: *index* is an NSNumber representing a positive integer and is a valid offset for dereferencing the pointer.

Dereferences the pointer, using *index* as the offset, and returns the value of the corresponding memory zone.

at:put:

-(id) **at:**(id)*index* **put:**(id)*elem*

Precondition: *index* is an NSNumber representing a positive integer and is a valid offset for dereferencing the pointer.

Dereferences the pointer, using *index* as the offset, and put *elem* in the corresponding memory zone. Retains *elem*, and release the replaced object.

44 FSPointer *(F-Script User Object)*

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: FScript/FSPointer.h

Class Description

The FSPointer class is a semi-abstract class that provides an interface for dealing with C pointers in F-Script. In practice, you will use instances of either FSGenericPointer or FSObjectPointer, the two concrete subclasses of FSPointer.

An FSPointer denotes a memory location. FSPointer objects are used to represent C pointers in F-Script. When a method returns a C pointer (other than an object) it is automatically mapped to an FSGenericPointer object. When calling, from F-Script, a method that takes a C pointer as an argument (other than an object), the FSPointer object you provide (either an FSGenericPointer or an FSObjectPointer) is automatically mapped to a C pointer that is passed to the method. The NULL pointer is mapped to nil and vice-versa.

FSGenericPointer and FSObjectPointer differ on the following points:

- An FSGenericPointer object can stand for a C pointer pointing to any type of data. On the contrary, an FSObjectPointer can only stand for a C pointer pointing to one (or more) object(s).
- By default, it is up to you to ensure that the memory zone referenced by an FSGenericPointer is eventually freed. On the contrary, an FSObjectPointer will automatically free its memory zone on deallocation. You can get the same behavior with an FSGenericPointer by sending it the message `setFreeWhenDone:YES`.
- When the method `at:put:` is invoked on an FSObjectPointer to put an object in the memory zone referenced by the FSObjectPointer, the object is automatically retained and the replaced object is automatically released.
- When an FSObjectPointer is passed as argument to a method that takes a C pointer, all the objects in the memory zone referenced by the FSObjectPointer are automatically autoreleased before the execution of the method. All the objects present in this memory zone after the execution of the method are automatically retained.
- On deallocation, an FSObjectPointer release all the objects present in the memory zone it points to.

This somewhat complex behavior of FSObjectPointer related to memory management ensures that the lifetime of the pointed objects is extended enough to provide a good user experience during interactive sessions.

When you invoke, from F-Script, a method that takes a C pointer (other than an object) as argument you can pass an FSObjectPointer if the C pointer points to a memory zone containing objects. In other cases, you must use pass an FSGenericPointer.

Example:

Given the following method:

```
- (void) method1:(NSNumber **)p
{
  // Replace the NSNumber referenced by p by a new NSNumber
  // with a value that is the double of the original one.

  *p = [NSNumber numberWithDouble:[*p doubleValue] * 2];
}
```

You can interact with it from F-Script like this:

```
> myPointer := FSPointer objectPointer

> myPointer at:0 put:22
22

> myObject method1:myPointer

> myPointer at:0
44
```

Method Types

Creating instances	+ malloc: + objectPointer + objectPointer:
Getting the C pointer	- cPointer

Class Methods

malloc:

+(FSGenericPointer *) **malloc:**(size_t)*size*

Allocates a memory zone of size *size* with the C malloc() function, then creates and returns an FSGenericPointer instance pointing to this zone, or nil if the memory cannot be allocated. It is up to you to free the zone (see the FSGenericPointer class).

objectPointer

+(FSObjectPointer *) **objectPointer**

Equivalent to a call to +objectPointer: with 1 as argument.

objectPointer:(size_t)*count*

+(FSObjectPointer *) **objectPointer:**(size_t) *count*

Allocates a memory zone large enough to contain *count* object pointers. The C malloc() function is used for allocating the memory. The memory zone is filled with nil values. Creates and returns an FSObjectPointer instance pointing to this zone, or nil if the memory cannot be allocated. The memory zone will be freed on deallocation of the returned FSObjectPointer.

Instance Methods

cPointer

-(void *) **cPointer**

Returns the memory location denoted by the receiver.

45 FSVoid *(F-Script User Object)*

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	FScript/FSVoid.h

Class Description

In F-Script, a method must always return a result (and this result must be an object). This differs from the Cocoa model where a method can return nothing (this is denoted by the keyword **void** in Objective-C and Java). When such a method is called by F-Script, the messaging system returns an instance of FSVoid as the result of the method.
An FSVoid object cannot do anything useful and is merely used to solve this mapping problem.
There is usually only one instance of FSVoid per application. This can be obtained using the **+fsVoid** method.

USER SECTION

Method Types

Obtaining the shared instance	+fsVoid
String representation	-printString

Class Methods

fsVoid

+ **fsVoid**

Returns the shared instance of FSVoid.

Instance Methods

printString

-(NSString*) **printString**

Returns a string representation of the receiver. This is an empty string.

46 Number *(F-Script User Object)*

Inherits From: NSNumber

Declared In: FScript/Number.h

Class Description

Number is a concrete subclass of the NSNumber class cluster. A Number object holds a **double** C type number, so it provides the same precision as the "double" C type of the platform used. Number provides an optimized implementation of the methods defined by NSNumber (and its FSNSNumber category).

47 System *(F-Script User Object)*

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	FScript/System.h

Class Description

A System object is a special object that gives users of F-Script an interface to some services of the interpreter. A predefined F-Script identifier called **sys** gives access to a system object from F-Script. This object can be viewed as a handler to the current interpreter.

Some methods of System are related to the notion of space. A space (or workspace) is an environment (i.e., a set of variable names and associated objects).

Method Types

Creating blocks from strings	-blockFromString: -blockFromString:onError:
Loading an object from a file	-load -load:
Loading a space from a file	-loadSpace -loadSpace:
Saving the current space to a file	-saveSpace -saveSpace:
Getting defined variables' names	-identifiers
Copying	-setValue: -clone
Getting information on users	-fullUserName -homeDirectory -homeDirectoryForUser: -userName
Logging a string	-log:
Beeping	-beep
Performing the internal test suite	-ktest
Installing the Flight tutorial	-installFlightTutorial

Opening an object browser	-browse -browse:
Attaching a managed object context	-attach:

Instance Methods

attach:

-(void)**attach:**(id)*managedObjectContext*

For each entity associated with the managed object context, this method defines, in the workspace associated with the receiver, an array which is given the same name as the entity and which contains all the managed objects corresponding to the entity at the time of invocation.

For instance, if `myObjectContext` is a managed object context associated with an Employee entity and a Department entity, then, after executing:

```
sys attach:myObjectContext
```

the F-Script workspace will contains two new arrays, named `Employee` and `Department`. These arrays will contain the managed objects for their corresponding entities: the `Employee` array will contain all the employees and the `Department` array will contain all the departments.

This method is likely to bring in memory the whole persistent object graph associated with the object context. In practice this should not be a problem, unless you deal with very big data-sets. In such cases you should avoid using this method.

Note that the attach: method is just a convenience method. You still have full access to the Core Data framework and can use its various APIs to get access to managed objects.

beep

-(void)**beep**

Plays the system beep by calling NSBeep().

blockFromString:

-(id)**blockFromString:**(NSString*)*source*

Creates and returns a block described in *source.* The *source* string should represent a block literal. If there is a syntax error in *source*, an F-Script execution error is generated. The new block is set to have the top-level environment of the interpreter – represented by the receiver – as its parent environment.

blockFromString:onError:

-(id)**blockFromString:**(NSString*)*source* **onError:**(Block*)*errorBlock*

Creates and returns a block described in *source.* The *source* string should represent a block literal. If there is a syntax error in *source*, *errorBlock* is executed with three arguments: an NSString with the error message, a number with the starting position of the syntax error in *source* and a number with the final position of the syntax error in *source*; the result of the *errorBlock* execution is then returned.
The new block is set to have the top-level environment of the interpreter – represented by the receiver – as its parent environment.

browse

-(void) **browse**

Open a graphical object browser for the objects defined at the global level of the F-Script interpreter represented by the receiver.

browse:

-(void) **browse:**(id)*anObject*

Open a graphical object Browser on *anObject*.

clone

-(System*) **clone**

Returns an auto-released copy of the receiver.

fullUserName

- (NSString *) **fullUserName**

Returns a string containing the full name of the current user.

homeDirectory

- (NSString *) **homeDirectory**

Returns a path to the current user's home directory.

homeDirectoryForUser:

- (NSString *) **homeDirectoryForUser:**(NSString *)*userName*

Returns a path to the home directory for the user specified by *userName*. Returns nil if the home directory for *userName* is not found.

identifiers

-(Array *) **identifiers**

Returns, in the form of an Array of NSStrings, the names of the user defined variables.

ktest

-(id) **ktest**

Performs an internal test suite. Returns the result in a string.

installFlightTutorial

-(void) **installFlightTutorial**

Installs or reinstalls the "Flight" tutorial. This may create some objects in the workspace.

load

-(id) **load**

Asks the user for a fileName then loads and returns the object stored in it. An exception is raised if the loading process fails.

load:

-(id) **load:**(NSString*)*filename*

Loads and returns the object stored in the file called *filename*. An exception is raised if the loading process fails.

loadSpace

-(void) **loadSpace**

Asks the user for a fileName then loads the space stored in it and makes it the current space (the current environment) of the interpreter represented by the receiver. An exception is raised if the loading process fails.

This method is generally invoked directly by the user, in interactive mode. When invoked inside a script, there is a caveat: once this method is called, the code in the script will no longer have access to any global variables.

loadSpace:

-(void) **loadSpace:**(NSString*)*filename*

Loads the space stored in the file named *filename* and makes it the current space (the current environment) of the interpreter represented by the receiver. An exception is raised if the loading process fails.

This method is generally invoked directly by the user, in interactive mode. When invoked inside a script, there is a caveat: once this method is called, the code in the script will no longer have access to any global variables.

log:

-(void)**log:**(id)*object*

If *object* is an NSString, logs it using the NSLog() Foundation function. Otherwise, obtains a string representation of *object* using the printString method on *object* and logs this string with the NSLog() function.

saveSpace

-(void) **saveSpace**

Asks the user for a fileName then saves the current space of the interpreter represented by the receiver in the designated file by calling **saveSpace:**. An exception is raised if the archiving process fails.

saveSpace:

-(void) **saveSpace:**(NSString*)*filename*

Saves the current space of the interpreter represented by the receiver in the designated file. An exception is raised if the archiving process fails.

This method is based on the Cocoa archiving mechanism. Note that if an object in the object graph being archived does not conform to the Cocoa NSCoding protocol, an instance of NSNull is archived in place of this object.

setValue:

-(void) **setValue:** (id)*operand*

Precondition: *operand* is a System.

Sets the value of the receiver to the value of *operand*.

userName

- (NSString *) **userName**

Returns the name of the current user.

API for ObjC Programmers

48 FScriptMenuItem

Inherits From:	NSMenuItem
Conforms To:	NSMenuItem (NSMenuItem) NSObject (NSObject)
Declared In:	FScript/FScriptMenuItem.h

Class Description

This subclass of NSMenuItem lets you easily embed a complete F-Script environment into an application. Each FScriptMenuItem instance has its own F-Script interpreter and contains a submenu that provides access to the standard F-Script user interface elements. From an FScriptMenuItem, the user can access an F-Script console, open object browsers and open the F-Script preference panel.

Typically, an FScriptMenuItem is initialized with the `init` method. Then, it can be inserted into an existing menu. For instance, the following Objective-C code adds an F-Script menu to the main menu of the application:

```
[[NSApp mainMenu] addItem:[[[FScriptMenuItem alloc] init] autorelease]];
```

An FScriptMenuItem is associated with an FSInterpreterView (which is used by the console window). You use the `interpreterView` method to access the FSInterpreterView. From the FSInterpreterView, you can programmatically access the associated FSInterpreter.

Method Types

Getting the interpreter view	-interpreterView

Instance Methods

interpreterView

-(FSInterpreterView *) **interpreterView**

Returns the FSInterpreterView object associated with the receiver.

49 FSInterpreter

Inherits From:	NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	FScript/FSInterpreter.h

Class Description

An instance of this class is an F-Script interpreter. You can ask it to execute instructions given in a string.

This class allows you to programmatically integrate an F-Script interpreter into your own project.

Each interpreter manages its own workspace. The F-Script code passed to the “execute:” method is executed in the context of this workspace.

The lifecycle of a workspace is closely related to the lifecycle of the associated interpreter. Since it is possible to create block objects in a workspace and to enable them to be held for later execution, problems may arise if you try to execute a block whose workspace was controlled by an interpreter that has been deallocated. Thus, you must ensure that the interpreter in charge of a workspace is not deallocated until the workspace is no longer needed.

Method Types

Creating an interpreter	+interpreter -init
Validating syntax	+validateSyntaxForIdentifier:
Executing F-Script instructions	-execute:
Managing variables	-objectForIdentifier:found: -setObject:forIdentifier: -identifiers:
Journaling	-setJournalName: -setShouldJournal: -shouldJournal
Opening an object browser	-browse -browse:

Class Methods

interpreter

+(FSInterpreter) **interpreter**

Creates and returns an interpreter.

validateSyntaxForIdentifier:

+(BOOL) **validateSyntaxForIdentifier:**(NSString *)*identifier*

Returns whether *identifier* fits the F-Script syntax for identifiers.

Instance Methods

browse

-(void) **browse**

Open a graphical object browser for the objects defined at the global level of the interpreter.

browse:

-(void) **browse:**(id)*anObject*

Open a graphical object browser on *anObject*.

execute:

-(FSInterpreterResult *) **execute:**(NSString *)*command*

Executes the F-Script code in *command*.

identifiers

-(NSArray *) **identifiers**

Returns the list of the identifiers defined in the top-level environment of the interpreter. The list is returned as an array of NSString objects. Note that this does not include pre-defined identifiers like class names or Cocoa constants.

init

-(id) **init**

Initializes a newly allocated interpreter.

objectForIdentifier:found:

-(id)**objectForIdentifier:**(NSString *)*identifier* **found:**(BOOL *)*found*

Returns the object associated with *identifier* in the top-level environment (i.e. name space) of the interpreter. Put YES if *identifier* is defined, put NO otherwise, in the BOOL pointed by *found*. You can pass NULL for *found* if you don't need it.

setJournalName

- (BOOL)**setJournalName:**(NSString *)*filename*

Sets the name of the journal file to *filename*.

setObject:forIdentifier:

-(void)**setObject:**(id)*object* **forIdentifier:**(NSString *)*identifier*

Defines *identifier* in the top-level environment (i.e. name space) of the interpreter and associates it with *object*.

setShouldJournal:

- (void)**setShouldJournal:**(BOOL)*shouldJournal*

If *shouldJournal* is YES, future commands received by the interpreter will be recorded in the journal file. If *shouldJournal* is NO, future commands will not be recorded.

shouldJournal

-(BOOL)**shouldJournal**

Returns YES if journaling is currently enabled, otherwise returns NO.

50 FSInterpreterResult

Inherits From:	NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	FScript/FSInterpreterResult.h

Class Description

An instance of this class represents the result of the execution of some F-Script code. Typically, you get an FSInterpreterResult by sending the **execute:** message to an FSInterpreter or the **executeWithArguments:** message to a Block. You can then query the FSInterpreterResult to find out if the execution raised an error or not, and get the result of the execution or an error description.

Method Types

Querying status	-isOK -isExecutionError -isSyntaxError
Getting the results of the executions	-result
Getting error descriptions	-errorMessage -errorRange
Showing the call stack to the user	-inspectBlocksInCallStack

Instance Methods

errorMessage

-(NSString *) **errorMessage**

Precondition: *receiver* isOK = false

Returns the error message held by the receiver. Raises FSInterpreterResultIllegalCallException if the precondition is not met (i.e. if the receiver does not represent an error).

errorRange

-(NSRange)**errorRange**

Precondition: *receiver* isOK = false

Returns the range of characters of the error represented by the receiver. Raises FSInterpreterResultIllegalCallException if the precondition is not met (i.e. if the receiver does not represent an error).

inspectBlocksInCallStack

-(void)**inspectBlocksInCallStack**

Precondition: *receiver* isOK = false

Shows the user the blocks in the error call stack. This is useful when an error occurs during an interactive session.

isExecutionError

- (BOOL)**isExecutionError**

Returns YES if the receiver represents an F-Script execution error, otherwise returns NO.

isOK

- (BOOL) **isOK**

Returns YES if the receiver represents an F-Script execution without error, otherwise returns NO.

isSyntaxError

-(BOOL) **isSyntaxError**

Returns YES if the receiver represents an F-Script syntax error, otherwise returns NO.

result

-(id) **result**

Precondition: *receiver* isOK

Returns the object which has been returned by the F-Script execution. Raises FSInterpreterResultIllegalCallException if the precondition is not met (i.e. if the receiver represents an error).

51 FSInterpreterView

Inherits From:	NSView
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	FScript/FSInterpreterView.h

Class Description

This View subclass is a command line interface to an F-Script interpreter. The interpreter is bundled into the FSInterpreterView.

Method Types

Getting the interpreter object	-interpreter
Putting a Command	-putCommand:
Notifying the user	-notifyUser:
Managing fonts	-fontSize -setFontSize:

Instance Methods

fontSize

-(float) **fontSize**

Returns the size of the font used by the receiver.

interpreter

-(FSInterpreter *) **interpreter**

Returns the interpreter object used by the receiver.

notifyUser:

- (void) **notifyUser:**(NSString *)*message*

Shows *message* to the user.

putCommand:

- (void) **putCommand:**(NSString *)*command*

Puts the command *command* in the receiver as if it was a user-entered command.

setFontSize:

-(void) **setFontSize:**(float)*theSize*

Sets the size of the font used by the receiver to *theSize*.

52 F-Script Functions

This section gives detailed descriptions of the C functions provided by the F-Script framework.

FSArgumentError()

SUMMARY Generates an F-Script execution error indicating an invalid argument

SYNOPSIS

```
#import <FScript/FScriptFunctions.h>

void FSArgumentError(id argument, int index, NSString *expectedClass, NSString *methodName)
```

DESCRIPTION

This method is used when an argument passed to a user method is not of the expected class. This method generates an F-Script execution error (including an error message), by calling FSExecError().
The *argument* object is the invalid argument.
The *index* is the position of the argument in the argument list of the called method (first argument is in position 1).
The *expectedClass* string is the name of the expected class.
The *methodName* string is the name of the method called with the invalid argument.

FSExecError()

SUMMARY Generates an F-Script execution error

SYNOPSIS

```
#import <FScript/FScriptFunctions.h>

void FSExecError(NSString *errorStr)
```

DESCRIPTION

This method raise an exception signaling an F-Script execution error. *errorStr*, which should be an error message, is used as the exception "reason". The name of the exception is "FSExecutionErrorException".

FSVerifClassArgs()

SUMMARY Verifies the classes of the arguments given to a user method.

SYNOPSIS

```
#import <FScript/FScriptFunctions.h>

void FSVerifClassArgs (NSString *methodName, int nbArgs,…)
```

DESCRIPTION

Use this method to test the classes of the arguments passed to a method you implement. For example, if you implement a method with two arguments, a NSNumber (or nil) and a NSString, you may call FSVerifClassArgs() at the beginning of your method body:

```
-myMethod:(NSNumber *)arg1 :(NSString *)arg2
{
   FSVerifClassArgs(@"myMethod",2,arg1,[NSNumber class],(int)1,arg2,[NSString class],
(int)0);


    /*  functional code of myMethod here */
}
```

For each argument of your method, you must give a group of three arguments to FSVerifClassArgs :

- the argument of your method
- the expected class of the argument
- the integer 1 if the argument can be nil, otherwise the integer 0 (do not forget to cast them as (int) (see the example)).

FSVerifClassArgs generates an F-Script execution error (by calling FSExecError()) if an argument is not valid.

FSVerifClassArgsNoNil()

SUMMARY Verifies the classes of the arguments given to a user method.

SYNOPSIS

#import <FScript/FScriptFunctions.h>

void **FSVerifClassArgsNoNil** (NSString **methodName, int nbArgs,…*)

DESCRIPTION

The same as FSVerifClassArgs except that the arguments you test cannot be nil.

Example:

```
-myMethod:(NSNumber *)arg1 :(NSString *)arg2
{
   FSVerifClassArgsNoNil(@"myMethod",2,arg1,[NSNumber class],arg2,[NSString class]);


    /*  functional code of myMethod here */
}
```