

**A USER'S GUIDE TO THE WOVEN ELECTRONIC
BOOK SYSTEM WITH SCRIPTS**

for WEBSs 3.0b2

Jacques Monnard

Our research on “The Integrated Electronic Book” was sponsored, from December 1985 until November 1991, by the the Swiss National Science Foundation, under grants No. 1.018-0.84 and 1.813-0.88.

Institute of Informatics
University of Fribourg
CH - 1700 Fribourg

May 1993

Table of Contents

Introduction.....	1
1 The IEB Concept.....	3
1.1 Documents.....	4
1.2 Blocks and Links.....	5
1.3 Sets.....	5
1.4 Scripts.....	5
1.5 The IEB Database.....	5
2 Using WEBSs.....	9
2.1 Recommendations.....	9
2.2 Using Documents.....	10
2.2.1 Creating New Documents.....	10
2.2.2 Opening and Importing Documents.....	10
2.2.3 Getting Information about a Document.....	10
2.2.4 Other Operations with Documents.....	10
2.3 Using Browsers.....	11
2.3.1 Creating a Browser.....	11
2.3.2 Creating Nodes.....	11
2.3.3 Connecting Nodes.....	11
2.3.4 Selecting Nodes.....	12
2.3.5 Adding a Father to a Node.....	13
2.3.6 Deleting Nodes.....	13
2.3.7 Copying/Cutting a Subtree into the Clipboard.....	13
2.3.8 Pasting a Subtree from the Clipboard.....	14
2.3.9 Contracting and Expanding Nodes.....	14
2.3.10 Coloring and Shading Nodes.....	14
2.3.11 Editing Nodes Contents.....	15
2.3.12 Opening Documents.....	15
2.3.13 Closing Documents.....	16
2.3.14 Printing Documents.....	16
2.3.15 Searching for a String in Documents.....	16
2.4 Using the Active Browser.....	16
2.4.1 Activating a Browser.....	16
2.4.2 Showing the Active Browser.....	16
2.4.3 Sequentially Opening Documents.....	17
2.5 Working with Blocks and Links.....	17
2.5.1 Creating Links.....	17
2.5.2 Following Links.....	18
2.5.3 Deleting Links.....	18

2.5.4	Creating Blocks.....	18
2.5.5	Deleting Blocks.....	18
2.5.6	Other Operations with Blocks.....	18
2.5.7	Adding Information to Blocks and Links.....	19

2.6	Viewing and Editing Object Properties.....	19
2.6.1	Activating an Object Information Dialog Box.....	19
2.6.2	Editing Types.....	21
2.7	Working with Sets.....	21
2.7.1	Accessing the List of Sets.....	21
2.7.2	Creating and Editing a Set.....	22
3	The Scripting Environment.....	24
3.1	What are Scripts ?.....	24
3.1.1	Unbound Scripts.....	24
3.1.2	Triggered Scripts.....	24
3.1.3	Object-Oriented Programming Fundamentals.....	25
3.2	Working with Scripts.....	26
3.2.1	The Script Browser.....	26
3.2.2	Creating a New Script.....	27
3.2.3	Compiling a Script.....	27
3.2.4	Saving a Script.....	28
3.2.5	Other Operations with Scripts.....	28
3.2.6	Executing Unbound Scripts.....	29
3.2.7	Automatic Execution of Scripts.....	29
4	Scripting Language Reference.....	30
4.1	Basics.....	30
4.2	Structure of a Script.....	31
4.2.1	The Script Header.....	31
4.2.2	The Main Block.....	33
5	WEBSs Reference Guide.....	38
5.1	The Apple Menu.....	38
5.2	The File Menu.....	39
5.3	The Edit Menu.....	41
5.4	The WEBSs Menu.....	41
5.5	The Script Menu.....	43
5.6	The Browser Menu.....	44
5.7	The Window Menu.....	45
5.8	The Text Menu.....	45
5.9	The Format Menu.....	45
5.10	The Node Menu.....	45
5.11	The Table Menu.....	45
	Appendix A: Getting Started with WEBSs: the Online Tutorial.....	47

A.1	Actions.....	47
A.2	Remarks.....	48
Appendix B: Trouble Shooting with WEBSs Databases.....		49
	Questions/Answers.....	49

Appendix C: Reserved Words, Special Symbols, Predefined Identifiers.51

C.1	Reserved Words.....	51
C.2	Special Symbols.....	51
C.3	Predefined Identifiers.....	51
	Types	51
	Constants.....	52
	Global Variables.....	52
	Routines.....	53
	Callable Methods.....	55
	Triggering Methods.....	59

Appendix D: Examples of Scripts.....60

D.1	Assigning a Predefined Window Location to a Document.....	60
D.2	Creating a Set of Visited Documents.....	60
D.3	Suggesting a Partial Ordering on Document Opening.....	60
D.4	Selectively Closing Documents.....	61
D.5	Formatting a Text Document.....	61
D.6	Creating a Simple Browser.....	61
D.7	Defining a Guided Tour.....	62
D.8	Searching and Replacing Strings in a Document.....	62
D.9	Creating a Glossary.....	63

Appendix E: Syntax of WEBSs Scripting Language.....65

Introduction

WEBSs is a highly interactive software system which allows for both creating and consulting Integrated Electronic Books (IEB) on Macintosh® computers.

This User's Guide is designed for new users of WEBSs who are already familiar with at least one Macintosh standard application. Indeed, WEBSs strictly follows the Macintosh user interface guidelines. Thus, both the mouse and the menus always function in the manner described in the Macintosh User's Manual.

This guide has been divided into five chapters and five appendices:

1 The IEB Concept

This chapter presents our conceptual model of an IEB. Its goal is to provide you with a good general understanding of what can be expected from a system such as WEBSs, which is essentially based on the *hypertext*¹ concept.

2 Using WEBSs

This chapter explains how to perform some of WEBSs most important functions such as opening and importing documents, using browsers or creating and following links.

3 The Scripting Environment

This chapter presents the concept of scripts and describes the scripting environment.

4 Scripting Language Reference

This chapter presents the essentials of the scripting language.

5 WEBSs Reference Guide

This chapter lists WEBSs menus and menu commands and describes their use.

Appendix A: Getting Started with WEBSs : The Online Tutorial

This appendix explains how to install and use WEBSs and its online tutorial on your Macintosh.

Appendix B: Trouble Shooting with WEBSs Databases

This appendix gives you tips for rectifying the inconsistencies which may arise between the WEBSs database and the documents on your disk.

Appendix C: Reserved Words, Special Symbols, Predefined Identifiers

Appendix D: Examples of Scripts

Appendix E: Syntax of WEBSs Scripting Language

Note: If you want to get started quickly with WEBSs, you can manipulate the online tutorial (bundled with the WEBSs package) in order to progress step by step through a 60 minutes session with the program that will introduce you to WEBSs key features (see Appendix A). Note that WEBSs provides Balloon Help™ for all menus and menu

¹ For more information on the notion of hypertext, please refer to : J. Conklin, *Hypertext: An Introduction and Survey*, Computer, Sept. 1987.

commands. If you want to use WEBSs more intensively, you should also read the first chapter of this guide and observe the recommendations in §2.1. To learn more about how to perform specific operations, you can then refer to Chapters 2 and 3.

This version of WEBSs was designed and implemented by Jacques Monnard and Jacques Pasquier-Boltuck. Any questions regarding the software should be mailed to :

Jacques Monnard / Jacques Pasquier-Boltuck
Institute for Automation and Operations Research
University of Fribourg
CH-1700 FRIBOURG
Switzerland
E-mail : monnardj@cfruni51.bitnet / pasquier@cfruni51.bitnet

Note: The current version of WEBSs can be obtained via ftp from site ufmisc.unifr.ch (134.21.1.33) with user name “webss” and no password.

1 The IEB Concept

An Integrated Electronic Book (IEB) represents an organized gathering of knowledge on a given subject. An IEB created with WEBSs is composed of a **collection of documents** containing information pertaining to the book's subject, that can be interconnected with the help of networks of **blocks** and **links**. **Sets** may also be created to regroup collections of objects. **Scripts** allow users to automate series of actions and to modify the behaviour of specific objects. Finally, **an IEB database** allows for the integration of the above components within a meaningful unit or book. Figure 1 shows a Macintosh folder containing the WEBSs application and two IEBs (one called "Simulation" and the other "Markov").

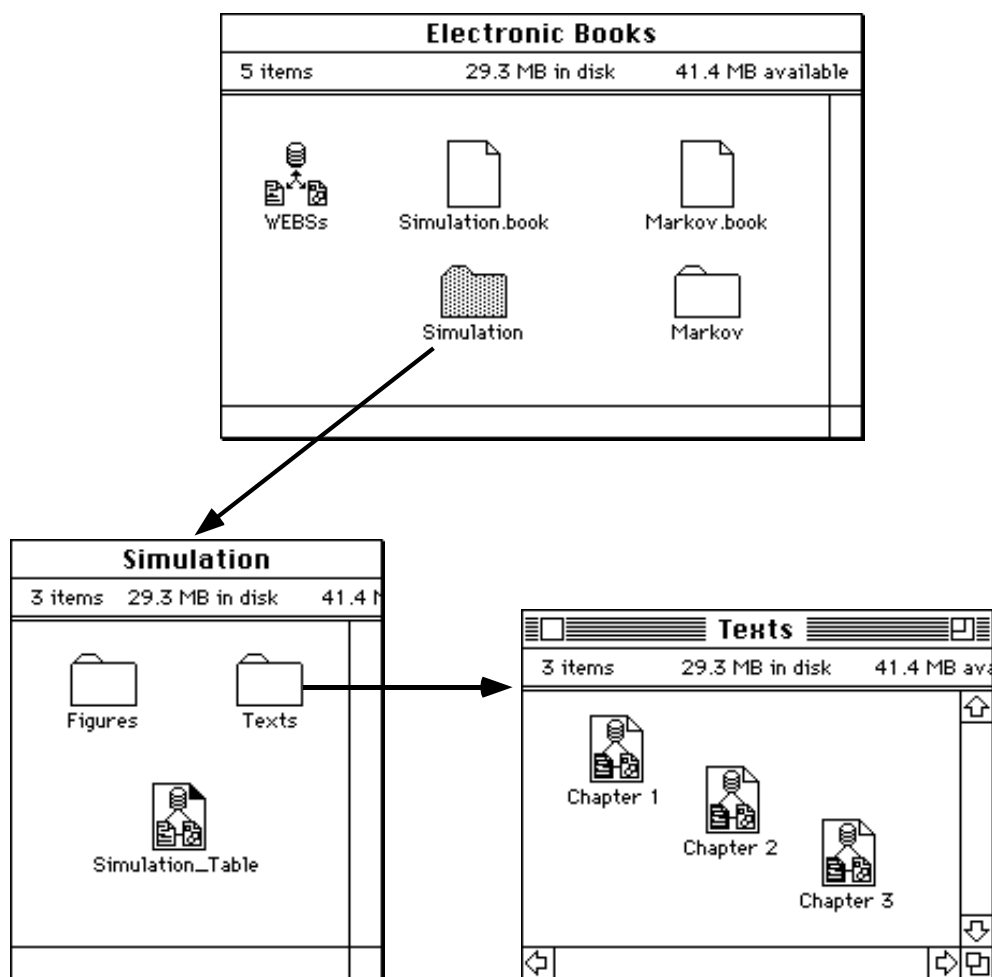


Figure 1: Two WEBSs IEBs viewed through the Macintosh desktop interface

1.1 Documents

In order to create IEBs, WEBSs supports four types of documents :

- **Textual documents.** These documents correspond to the text of the different chapters, sections, sub-sections and paragraphs found in classical books. They can be created with WEBSs own text editor, which allows for simple formatting operations : setting of font, size, style, color and justification for a piece of text. Alternately, you can use another word processor like Microsoft Word™ or MacWrite™, save your documents in “Text Only” format (note that this format doesn’t retain formatting) and then transform them into WEBSs documents.
- **Graphic documents.** These documents correspond to the illustrations, figures and other graphics found in classical books. WEBSs does not offer a graphical editor, but pictures created with applications like Canvas™ or MacDraw™ and saved in PICT format can be transformed into WEBSs documents.
- **Browser documents.** These documents can be compared to the tables of contents found in classical books. More precisely, a browser represents a tree-like structure, where each node can be associated with one or more documents. Browsers are used to hierarchically organize subsets of documents.

The possibility to freely create browsers of this sort, represents a powerful structuring tool, offering the potential for defining many different ways of organizing the same original collection of documents. In fact, once activated, a browser defines a particular reading environment, permitting, for example, the use of the commands Previous Node and Next Node (see “Using the Active Browser” in Chapter 2) in order to achieve a sequential reading of the IEB. A browser can also be used in order to selectively print or search for a string in all the documents associated with its nodes.

- **Logico-mathematical models.** These types of documents vary according to the particular subject, e.g. Markov chains or linear optimization models. An IEB on a complex subject may even offer several different types of models. It is possible to carry out complex calculations and manipulations (parameter changes, etc.) on these logico-mathematical models, the results of which can usually be cut and pasted within regular textual or graphic documents. Due to their versatility, it is clear that the educational potential of these models extends far beyond the simple guided exercises available in classical textbooks, and constitutes one of the principal advantages of an IEB over its paper competition.

The models supported by your version of WEBSs can be found by examining the File menu. This User’s Guide does not provide further information on this subject. Indeed, each of these models represents a “mini-application” of its own and should, thus, be provided with its own guide. Furthermore, new types of

models are still being created.

1.2 Blocks and Links

A **block** simply represents a selection made inside a document (for example, a string of characters in a textual document or a rectangle in a graphical document). A **link** corresponds to the connection between two blocks which the user may choose to follow in order to jump from one location in a document to another (or even to another document).

1.3 Sets

A **set** represents a collection of (possibly heterogeneous) objects – documents, blocks, links, scripts or other sets. Sets play an important role in scripts, as we will see in Chapters 3 and 4.

1.4 Scripts

A **script** is a program which is executed automatically by the system. Scripts have two main uses:

- First, they allow users to define new high-level functions by automating and sequencing any series of actions. By writing the appropriate script, a user can for example open all the documents that belong to a specific set with a single operation.
- Secondly, the behavior of the objects that constitute an electronic book can be enriched by creating scripts that will be automatically executed each time a triggering object performs a specific action.

Note: You can use WEBSs and even execute scripts without knowing anything about script writing. Scripting is meant for users who want to enhance the application functionality and tailor an electronic book to their needs. Therefore scripts have two chapters of their own in this User's Guide, i.e. Chapters 3 and 4. If you just want to execute existing scripts, take a look at §3.2.6.

1.5 The IEB Database

In the framework of WEBSs, an **IEB database** is like any other **Macintosh file**. This file automatically inherits the name of the

IEB it represents with the extension “.book”. Individual IEBs are always supported by a file of this sort, which must be located in the same folder as the WEBSs application. If this condition is not respected, WEBSs simply ignores the existence of the entire IEB, i.e. you will not even be able to open its documents.

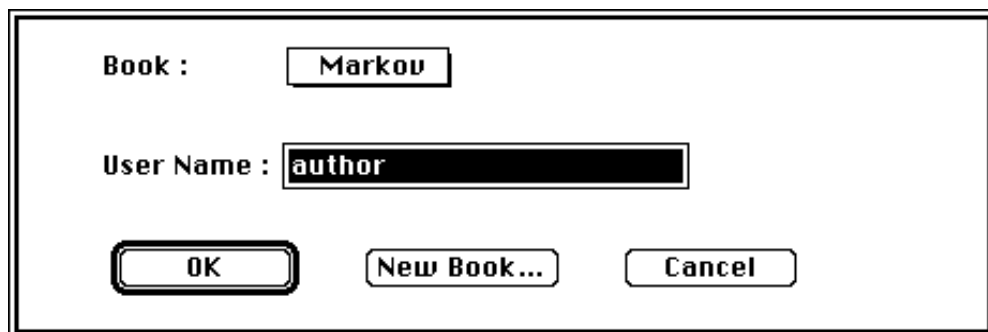
At the greatest level of detail, an IEB database is composed of webs. A web is made up of the following elements :

- a list of the documents belonging to the web, including the browsers that the system proposes to you as candidates for being the active browser (see “Using the Active Browser” in Chapter 2),

- for each document, a list of its blocks,
- a list of links,
- a list of sets,
- a list of scripts.

An IEB database is always composed of one public and zero or more private webs.

Depending on the name you enter in the dialog box that appears when you launch WEBSs or choose the Change Book command from the WEBSs menu (see Figure 2), you are either an author or a reader, and have access to one or two webs :



The dialog box has a title bar. Inside, there is a label 'Book :' followed by a text box containing the word 'Markou'. Below this is a label 'User Name :' followed by a text box containing the word 'author'. At the bottom of the dialog box, there are three buttons: 'OK', 'New Book...', and 'Cancel'.

Figure 2: The dialog box for selecting a book. You select an existing book from the Book pop-up menu, or alternately create a new one by clicking the “New Book” button.

- If you type “author” (or “Author”, since upper and lower case letters are equivalent), then you are considered as the author of the IEB. All the documents, blocks, links, scripts and sets you create belong to the **public** web of the current IEB. This means that other users are able to see and manipulate them in read-only mode, and even to modify them if they have announced themselves as authors. The public web is the active web, and is the only open web.
- If you enter any other name (e.g. “Reader X”), then you are a reader. All the documents, blocks, links, sets and scripts you create belong to your own **private** “Reader X” web of the current IEB. This means that no other user is able to see them. You also have read-only access to all the components of the public web. While you cannot modify documents belonging to the public web, you can create blocks and links in them. The private web is the active web.

As we can see, the web, depending on whether it belongs to the author or to a reader of an IEB, defines the public or private nature of the elements it contains. We now understand that the definition of an IEB within the WEBSs system follows a very flexible process which is summarized below and presented in Figure 3 :

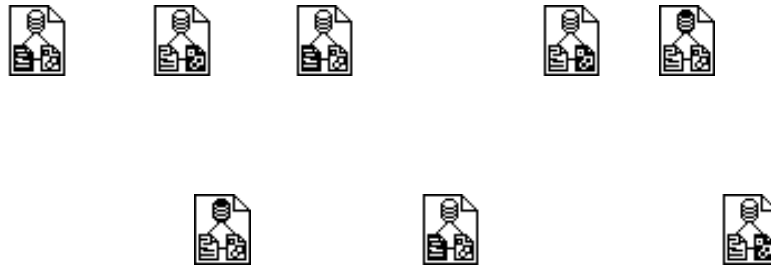


Figure 3a: Collection of an IEB documents. The different icons represent textual and graphical documents, as well as logico-mathematical models.

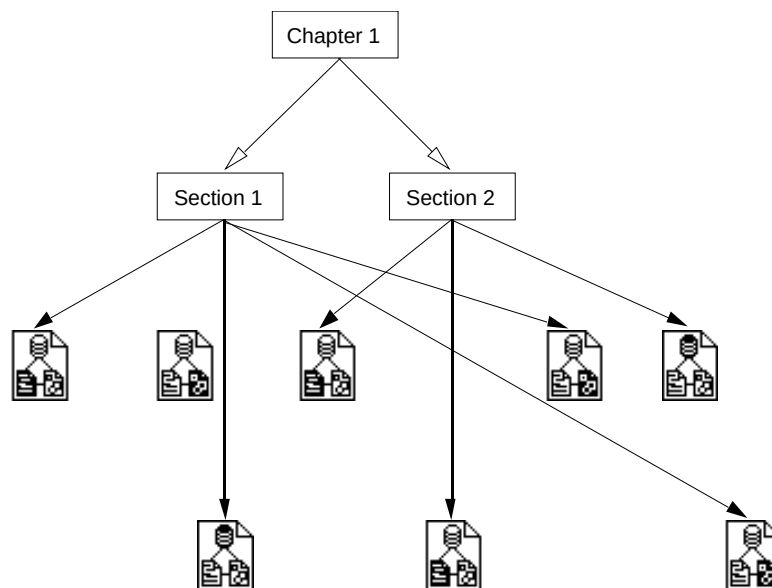


Figure 3b: The browsers allow for organizing the documents

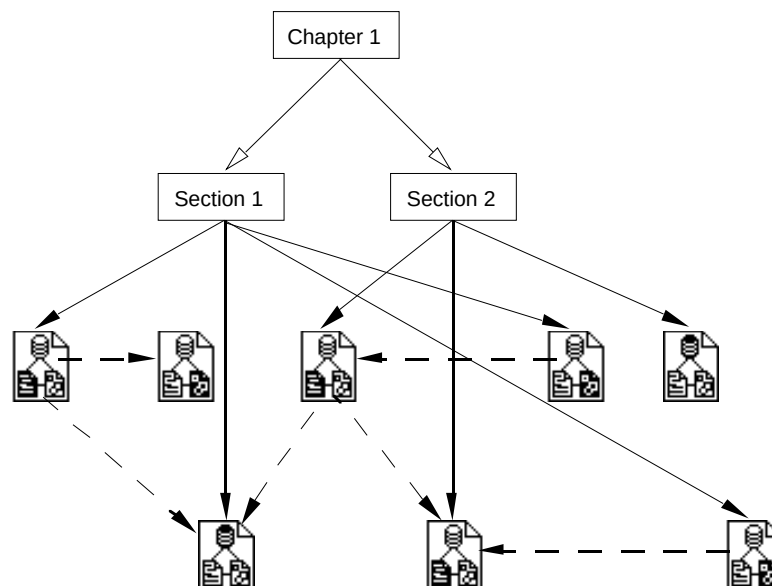


Figure 3c: The webs add an hypertext dimension to the IEB

1. A certain number of documents are created by an author inside a public web.
2. The author can then define a large network of blocks and links between the documents and organize all or part of them with the aid of various browsers and sets, as well as enrich the behaviour of his IEB with scripts.
3. Readers can consult these public documents using the particular reading environment defined jointly by the links, sets and scripts created by the author and by the browser that the individual reader has chosen to activate.
4. Finally, readers have at their disposal their own private webs, in which they can create private documents, define a private networks of blocks and links through all of the documents which are accessible to them, and even create their own browsers, sets and scripts. We see, therefore, that it is possible for users of WEBSs to enrich the reading environment offered by the author with the aid of a group of personal elements.

Before finishing this chapter, it is important for you to understand that the main role of an IEB database is to offer to authors and readers of IEBs a flexible means for defining a coherent hypertext working environment. Such an environment is essentially defined by three parameters :

1. the name you introduce at the beginning of a session with WEBSs, which classifies you as either an author or a reader;
2. the current IEB (see the Change Book command of the WEBSs menu in Chapter 5);
3. the active browser, if any (see the Activate command of the Browser menu in Chapter 5).

Note that while parameters 1 and 2 define which documents, sets and scripts and network(s) of links you can access, parameter 3 allows you to apply temporary hierarchical structures to these documents in order to make your interaction with them more efficient.

2.1 Recommendations

As was noted in Chapter 1, an IEB is composed of both a set of Macintosh documents and the IEB database, which is itself constituted of one or several webs. Theoretically, the consistency between the IEB documents and the IEB database is always ensured by WEBSs. Problems, however, may arise if the application experiences an unexpected crash or if you move, rename or delete one or several of the IEB documents using the Macintosh Finder instead of the appropriate commands of WEBSs File menu. Therefore, we recommend that you observe the rules below when working with WEBSs. Should you still encounter difficulties with an IEB, you should consult Appendix B which identifies potential problems and explains what to do in order to remedy them.

1. An IEB database file (i.e. the “AnyIEB.book” document automatically created by WEBSs) must always be at the same location on the disk as the WEBSs application. Examples of such files are the “Simulation.book” and “Markov.book” files of Figure 1.
2. All the documents of an IEB – with the exception of the IEB database file – must always be regrouped into a single IEB main folder (which may contain subfolders). The IEB main folder itself must also be at the same location on the disk as the WEBSs application. Examples of such folders are the “Simulation” and “Markov” folders of Figure 1. It is possible to disregard this rule and to scatter your IEB documents anywhere on one or even several disks, but this is a bad habit and will cause you problems, particularly if you want to create a backup of your IEB on a diskette.
3. To move, rename or delete documents belonging to an IEB,

use the appropriate commands from WEBSs File menu (see §5.2).

4. Each time you have made an important modification, either on an IEB document or on the active web (by creating or modifying blocks, links, sets or scripts), we recommend that you apply the Save Web command of the File menu in order to ensure the accuracy of the IEB data not only in main memory, but also on your disk, thus protecting yourself against an unexpected crash.
5. Avoid creating large text documents. A size of one to three printed pages is the most reasonable.
6. If you want to access objects by name in your scripts – particularly documents, giving them unique names will facilitate your task (see §4.2).

In the following, the names of menus and commands are in **bold face**.

2.2 Using Documents

2.2.1 Creating New Documents

WEBSs allows the creation of text and browser documents within the system. Depending on the version of WEBSs you use, different kinds of logico-mathematical models may also be created (check the New ... Document items under the File Menu).

To create a new document :

Choose the desired **New ... Document** command from the **File** menu.

After you have made any changes to the new document, you can use the **Save As** command from the **File** menu to save it under the name you choose. Later changes can be saved with the **Save** command.

2.2.2 Opening and Importing Documents

Any text, graphic, browser or model document which belongs to the current author's or reader's web can be opened, as well as any document with a standard TEXT or PICT file type. In this latter case, you will have to respond to a dialog box in order to install the document in the active web.

To open or import a document :

1. Choose the **Open** command from the File menu
2. Select the desired document in the dialog box that appears.

2.2.3 Getting Information about a Document

Choose the **Get Document Info** command from the **File** menu (see §2.6).



2.2.4 Other Operations with Documents

To move a document to another location or disk or rename it, use the **Move Document** command from the **File** menu.

To delete a document (both in the web and on disk), use the **Delete Document** command from the **File** menu.

2.3 Using Browsers


A browser is a tree-structured document, where each node may have one or more associated documents. An example of a browser is shown in Figure 4.

- The first node created in a browser is the root of the tree, and is indicated by a thick border. The root cannot be deleted. There is always exactly one root node in a browser, i.e. a browser may only contain one tree structure.
- Nodes and sub-nodes are connected by lines. The **Hide/Show Lines** command from the **Table** menu allows you to hide/show lines. This can be useful when the nodes are arranged vertically (see Figure 5)
- A gray document icon  on the left of a node indicates that it has associated documents.
- You can contract a node to hide its sub-nodes. The inverse action, expanding, shows the sub-nodes. This allows you to selectively hide some parts of the tree, and to concentrate on the nodes that interest you. A contracted node is indicated by an ellipsis ... under its name. If any sub-node of a contracted node has associated documents, a white document icon  is displayed on the left of the contracted node (e.g. “Chapter 2” in Figure 4).

2.3.1 Creating a Browser

Choose the **New Browser Document** command from the **File** menu.

2.3.2 Creating Nodes

1. Select the rectangle tool from the palette on the left of the window.
2. Click in the window (don't release the mouse yet), move the new node to the desired location, and release the mouse. A dialog box appears that allows you to change the node name and add/remove associated documents (see §2.3.11). If you do not want this dialog box to appear, press the -key when creating the node.

2.3.3 Connecting Nodes

1. Select the line tool from the palette on the left of the window.
2. Click on the father node, drag the pointer to the son node, and release the mouse.

A connection can only be created from the root or from a node which already belongs to the tree. You can only create connections when lines are shown (see Hide/Show Lines under the Table menu).

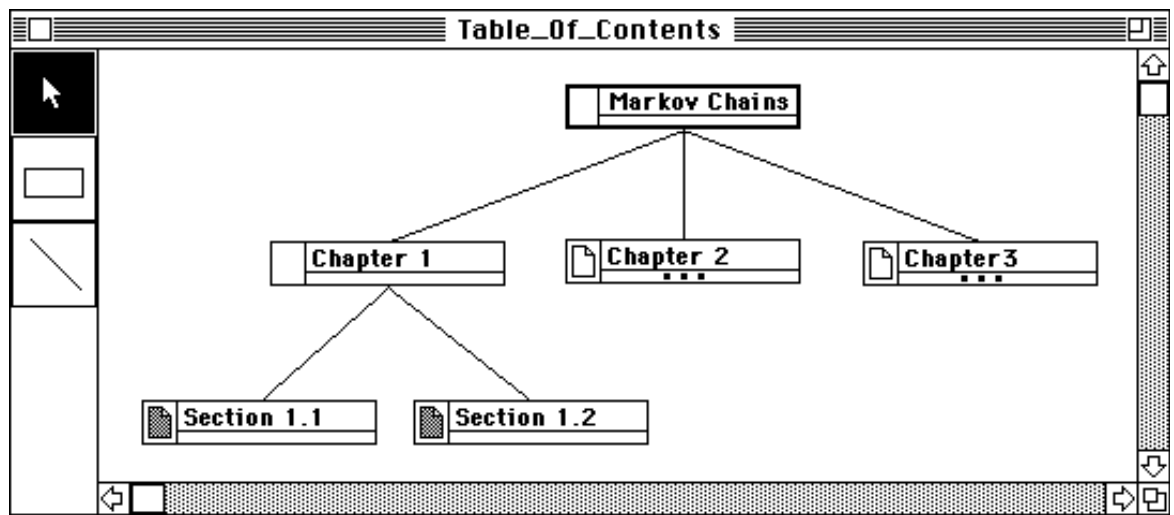


Figure 4: An example of a browser, with contracted nodes.

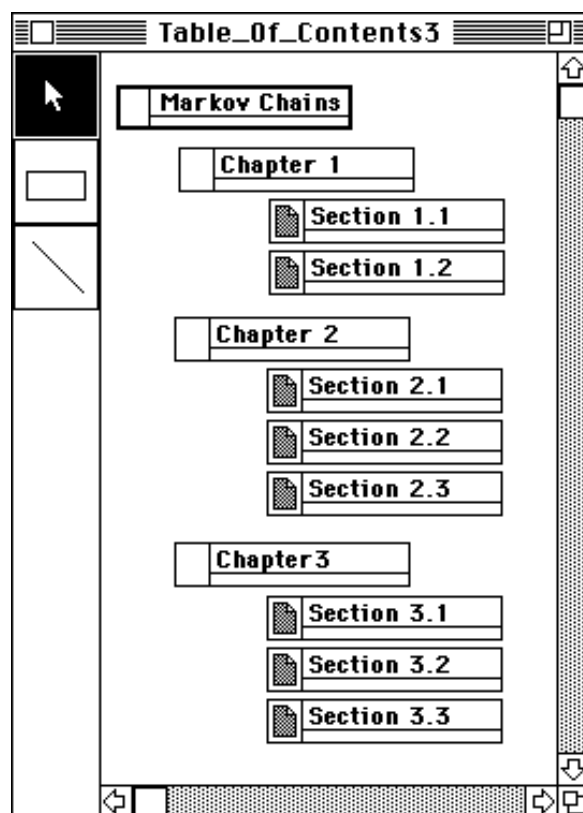


Figure 5: The same browser as in Figure 4, arranged vertically, and with expanded nodes. Connections between nodes are hidden.

2.3.4 Selecting Nodes

1. Select the arrow tool in the palette.
2. Click on the desired nodes or drag a rectangle around them. Pressing

the shift key while clicking or dragging allows for selection/deselection of multiple nodes.

Nodes can be moved freely within the window. Auto-scrolling allows you to scroll farther than the window's borders.

2.3.5 Adding a Father to a Node

To insert a node between a son and its father (see Figure 6) :

1. Select the son node.
2. Choose the **Add Father** command from the **Table** menu. A dialog box appears that allows you to give a name to the new node. The Add Father command is only enabled when the current selection consists of a single node already belonging to the table's tree.

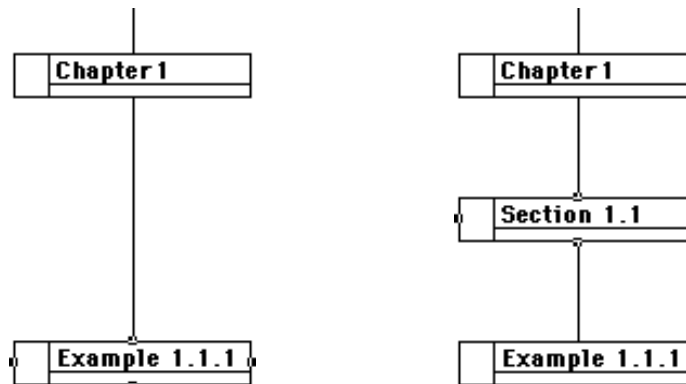


Figure 6: An example of the use of the Add Father/Remove commands. Selecting the node “Example 1.1.1” in the left tree and executing an Add Father command results in the tree on the right. Selecting “Section 1.1” in the right tree and executing a Remove command results in the left tree.

2.3.6 Deleting Nodes

To remove nodes from the document :

1. Select the nodes to be removed.
2. Choose the **Remove** command from the **Table** menu.

When a node is removed, what happens with its sons depends on whether the node was contracted or not. If it was contracted, then its sons are also deleted. If it was not contracted, then its sons are connected with its father (as in Figure 6). This second possibility allows you to remove a node without removing its sons.

2.3.7 Copying/Cutting a Subtree into the Clipboard

1. Select a single node in the tree.
2. Choose the **Copy** or **Cut** command from the **Edit** menu. Both commands will copy the subtree starting at the selected node into the clipboard. In addition, the Cut command will

also delete the subtree (the **Clear** command deletes the subtree, but does not copy it into the clipboard).

These commands have the same effect whether the selected node is contracted or not (i.e. hidden sub-nodes are copied resp. deleted).

2.3.8 Pasting a Subtree from the Clipboard

1. Select a non-contracted node in the tree.
2. Choose the **Paste** command from the edit menu. The subtree will be inserted as a son of the selected node.

You can only use the Paste command with lines shown (see Hide/Show Lines under the Table menu).

Figure 7 below gives an example of Cut/Paste operations. Note that you can use this facility across several browsers.

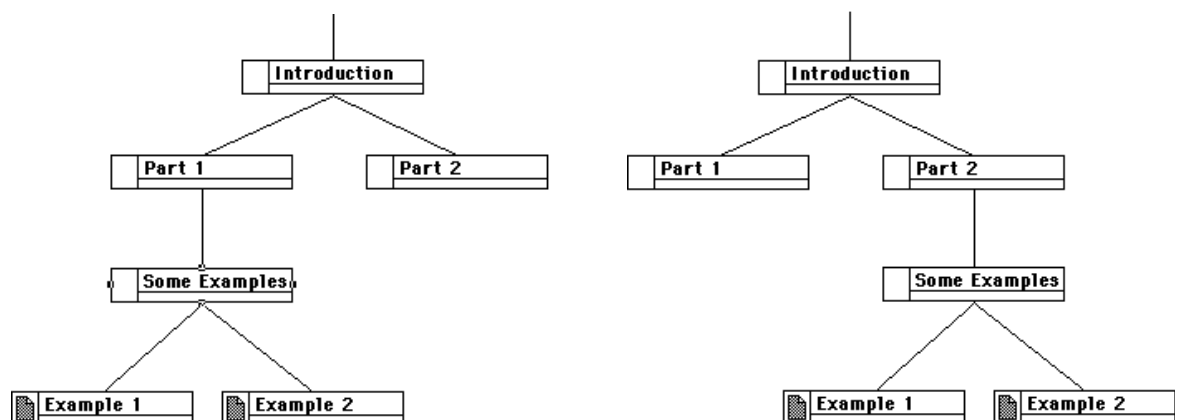


Figure 7: Example of Cut/Paste operations. The tree on the right was obtained from the one on the left by (1) selecting the node “Some Examples”, (2) executing a Cut command, (3) selecting the node “Part 2” and (4) executing a Paste command.

2.3.9 Contracting and Expanding Nodes

Choose the Contract resp. Expand commands from the Table menu. A short-cut for contracting and expanding is **⌘-double-click**. Note that a simple double-click opens the associated documents.

2.3.10 Coloring and Shading Nodes

1. Select the desired node(s).
2. Choose the desired color or shading from the **Node** menu.

2.3.11 Editing Nodes Contents

1. Select the desired node(s).
2. Choose the **Get Node Info** command from the **Node** menu. A short-cut for this command is **⌘-double-click**.

A dialog box showing the node name and its associated documents appears (see Figure 8).

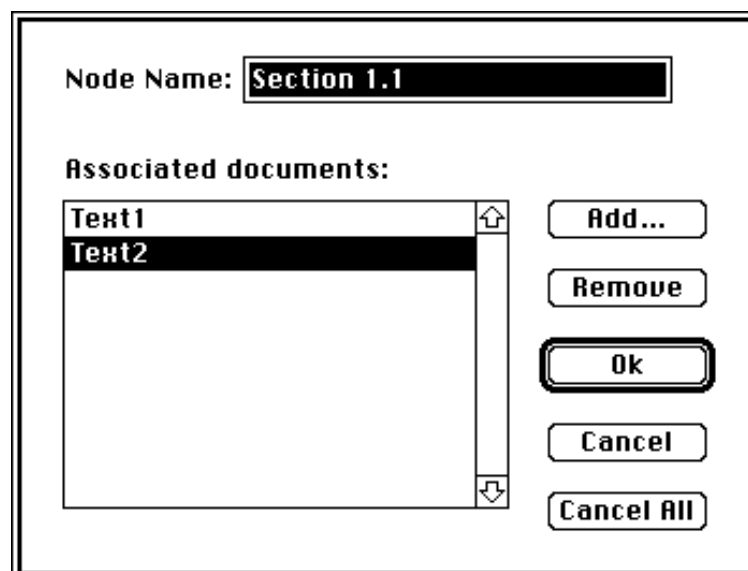


Figure 8: Dialog box for editing the contents of a node.

To associate a document with a node :

Click the **Add** button. This will bring up a Standard File dialog box, where you can select a document. Note that only accessible documents will be shown (i.e. those that belong to the current public and private webs). Browser documents can be associated with nodes; however a browser cannot be associated with its own nodes.

To disassociate a document :

1. Select the document(s).
2. Click the **Remove** button.

Note that if you are a reader, and you execute a Get Node Info command on the nodes of a public browser, you will not be able to make any changes to the nodes.

2.3.12 Opening Documents

To open the documents associated with one or several nodes :

1. Select the desired node(s)
2. Choose the **Open Documents** command from the **Table** menu.
Double-clicking on the node(s) will have the same effect.

If a node is contracted, all the documents associated with its sub-nodes will be opened.

2.3.13 Closing Documents

To close the documents associated with one or several nodes :

1. Select the desired node(s)
2. Choose the **Close Documents** command from the **Table** menu.
Control-double-clicking on the node(s) will have the same effect.

If a node is contracted, all the documents associated with its sub-nodes will be closed.

2.3.14 Printing Documents

To print all the documents associated with one or several nodes at once :

1. Select the desired node(s)
2. Choose the **Print Selected** command from the **File** menu.

This will print all the documents associated with the selected nodes and with their sub-nodes if they are contracted.

2.3.15 Searching for a String in Documents.

To search for a string in the textual documents associated with one or several nodes :

1. Select the desired node(s)
2. Choose the **Find Across** command from the **Table** menu.

2.4 Using the Active Browser

2.4.1 Activating a Browser

Select the desired browser in the **Activate** sub-menu of the **Browser** menu.

The browser document will be opened (if necessary) and marked as the active browser (a check mark will be displayed next to its name in the

Activate sub-menu), deactivating any previously activated browser. Closing an active browser's window will deactivate it.

The actions described below are only possible with an active browser.

2.4.2 Showing the Active Browser

Choose the **Show Active** command from the **Browser** menu.

2.4.3 Sequentially Opening Documents

If you want to sequentially read the documents associated with the active browser, you can use the **Next Node** and **Previous Node** commands from the **Browser** menu. These commands will :

- find the current selection within the active browser,
- move to the next resp. previous node with associated documents, or to the next/previous contracted node whose sub-nodes have associated documents, select it,
- open the associated documents.

If no node is selected, then these commands will go to the first resp. last node with associated documents. Thus, if you want to read all the documents associated with the active browser, you can just deselect any node, and start from the beginning/end with the Next Node/Previous Node commands.

You can switch the **Close Between Previous/Next** toggle under the **Browser** menu to automatically close the documents associated with the previous selection after execution of the command.

The tree is traversed in depth-first order. For the example from Figure 4, this gives the following sequence : Markov Chains, Chapter 1, Section 1.1, Section 1.2, Chapter 2 and Chapter 3. Unconnected nodes are visited last.

2.5 Working with Blocks and Links

Note: Blocks can be created in textual, graphical, and model documents, but not in browser documents.

2.5.1 Creating Links

The process of creating links is similar to copying and pasting text. You must :

1. Make a selection in a document (e.g. select a string in a text document, a rectangle in a graphical document, or other objects in a model document) or select an existing block marker (indicated by a •).
2. Choose the **Start Link** command from the **WEBSs** menu. If a block marker was not selected, a start block will be created from the selection and its marker will be inserted. A link will

now be pending. You can issue any number of commands unrelated to creating links while a link is pending.

3. Make a selection in the same or another document, or select another block marker.
4. Choose the **Complete Link** command from the **WEBSs** menu. This will complete the creation of the link, creating an end block and inserting its marker if necessary.

Several links can be connected to the same block. In order to accomplish this, you must simply select the desired block marker as endpoint when creating links.

2.5.2 Following Links

To follow a link, you have two possibilities :

- a) 1. Select a block marker.
2. Choose the **Follow Link** command from the **WEBSs** menu.
- b) Double-click on the block marker.

If several links are attached to the same block, a dialog box will ask you to select the link you want to follow. Links can be followed in either direction.

2.5.3 Deleting Links

1. Select a block marker associated to one end of the link.
2. Choose the **Delete Link** command from the **WEBSs** menu. If several links are attached to the same block, a dialog box will ask you to select the link you want to delete.

2.5.4 Creating Blocks

To create blocks that you will use later as endpoints for links or as recipients for your scripts:

1. Make a selection in a document (e.g. select a string in a text document, a rectangle in a graphical document, or other objects in a model document).
2. Choose the **Create Block** command from the **WEBSs** menu.

Block markers can be moved freely in graphical documents.

2.5.5 Deleting Blocks

1. Select a block marker.
2. Choose the **Delete Block** command from the **WEBSs** menu.

If links are connected to the deleted block, they are also destroyed.

2.5.6 Other Operations with Blocks

To visualize the extent of a block, select its marker and use the **Show Block Extent** command from the **WEBSs** menu.

The **Hide Blocks/Show Blocks** command from the **WEBSs** menu allows you to temporarily remove/put back all the block markers in the active window.

2.5.7 Adding Information to Blocks and Links

Use the **Get Block Info** and **Get Link Info** commands from the **WEBSs** menu (see §2.6).

2.6 Viewing and Editing Object Properties

2.6.1 Activating an Object Information Dialog Box

WEBSs provides a common dialog box for viewing and modifying attributes of any object in an electronic book, whether a document, a block, a link, a set or a script (see Figure 9). Depending on the kind of object you want to examine, you access it in the following way :

For a document :

1. Activate the desired document.
2. Choose the **Get Document Info** command from the **File** menu.

For a block :

1. Select the block marker.
2. Choose the **Get Block Info** command from the **WEBSs** menu

For a link :

1. Select the block marker at the start or the end of the link.
2. Choose the **Get Link Info** command from the **WEBSs** menu.

If several links are attached to the same block, a dialog box will ask you to select the desired link.

For a set :

1. Choose the **Edit Sets** command from the **WEBSs** menu.
2. Select a set in the list
3. Click the Get Info button.

For a script :

1. Activate the script browser.
2. Select a script in the script browser.
3. Choose the **Get Script Info** command from the **Script** menu.

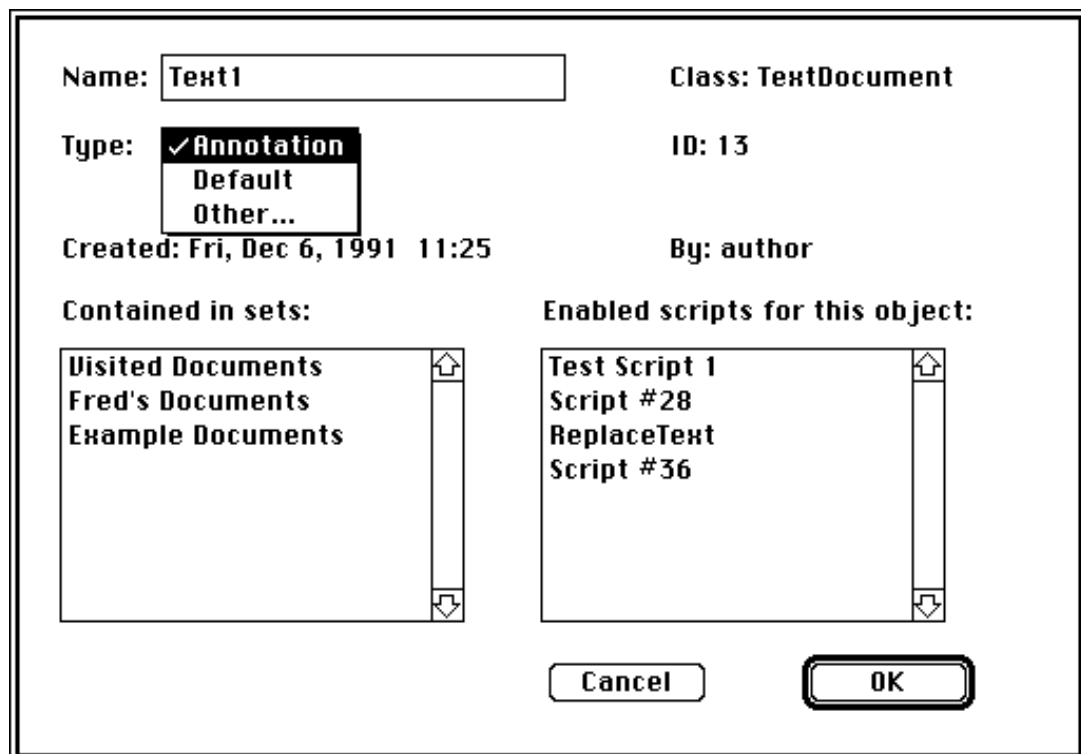


Figure 9: The dialog box for viewing and editing object properties. The two lists showing the sets to which the object belongs, as well as all the scripts that apply to it, are especially relevant for script writers.

The Object Information dialog displays the object's name, its class, id (each object in an IEB gets a unique id when it is created), creation date, and creator. Note also the type attribute : every object in an electronic book has a type which can take any value from a user-defined list, enabling you to define your own categories of objects (e.g. the link types *Reference*, *Relation*, and *Index*). This attribute may also be used as a selector in a script to assign a different behavior to distinct types of objects (see §4.2.1). The type attribute of an object should not be mistaken with its class (see §3.1.3).

Note: You can modify the object name and type if you have the required access rights. However, you cannot modify the name of a document from this dialog box; instead you must use the Move Document command from the File menu.

2.6.2 Editing Types

From the Object Information dialog, you can also create, view, and modify types by means of a type editor (see Figure 10). To access it :

Choose the **Other** item from the **Type** pop-up menu.

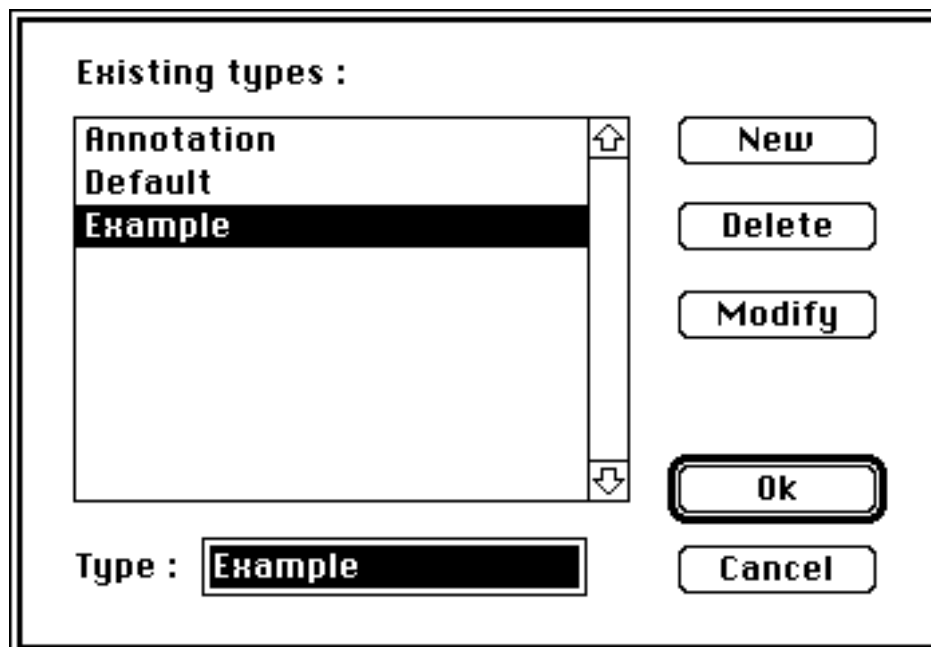


Figure 10: The Type editor.

All existing types in the current IEB are listed. Type names are not case sensitive. The type Default exists in every book, and cannot be modified or deleted.

- To create a new type, enter its name in the Type field and click the **New** button.
- To modify an existing type, select it in the list, modify its name in the Type field, and click the **Modify** button.
- To delete a type, select it in the list, and click the **Delete** button.

Note: All changes you make in the type editor are discarded if you click the Cancel button in the Object Information dialog box.

2.7 Working with Sets

2.7.1 Accessing the List of Sets

To access the facilities provided by WEBSs for creating and editing sets (see Figure 11):

Choose the **Edit Sets** command from the **WEBSs** menu.

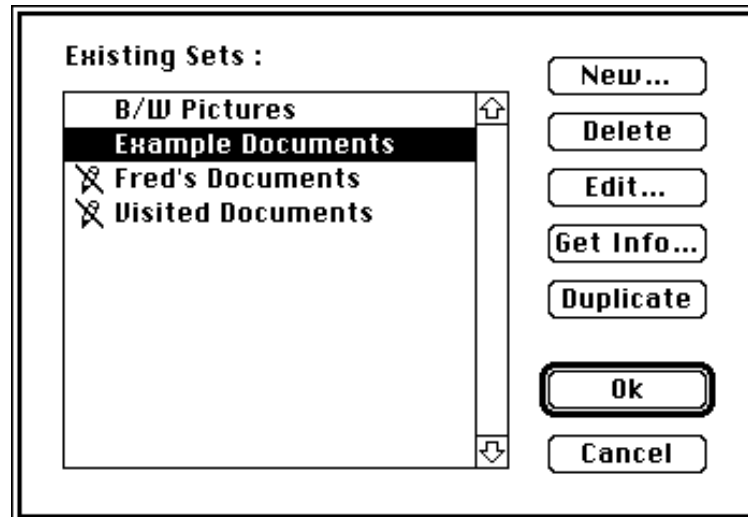



Figure 11: Dialog box showing all the sets in the open web(s).

If you have identified yourself as the reader, the sets that belong to the public web are indicated by a read-only icon . You cannot delete or modify them, but you can still visualize their contents or duplicate them.

The buttons have the following effect :

New : brings up a dialog box for creating a new set (see §2.7.2).

Delete : deletes the selected set.

Edit : brings up a dialog box for editing the contents of the selected set (see §2.7.2).

Get Info : brings up the dialog box for editing set properties (see §2.6).

Duplicate : makes a copy of the selected set.

OK : accepts all the changes you made after invoking the “Edit Sets” dialog (including all changes you made with the set editor) and closes the dialog box.

Cancel : discards all the changes you made after invoking the “Edit Sets” dialog (including all changes you made with the set editor) and closes the dialog box.

2.7.2 Creating and Editing a Set

When you click the New or Edit buttons in the dialog box of Figure 11, a set editor appears as shown in Figure 12.

The radio buttons at the bottom enable you to select the kind of objects you want to see : documents, blocks, links, sets or scripts. The two lists display all the objects of the selected kind that belong to the set (left), and those that are not in the set (right). Only objects that belong to the open web(s) are shown. Note that a set can contain heterogeneous objects (e.g. documents and blocks).

To add or remove objects from a set, select them and click the **Add/Remove** button (the name of the Add button will change to Remove when you select objects in the list on the left).

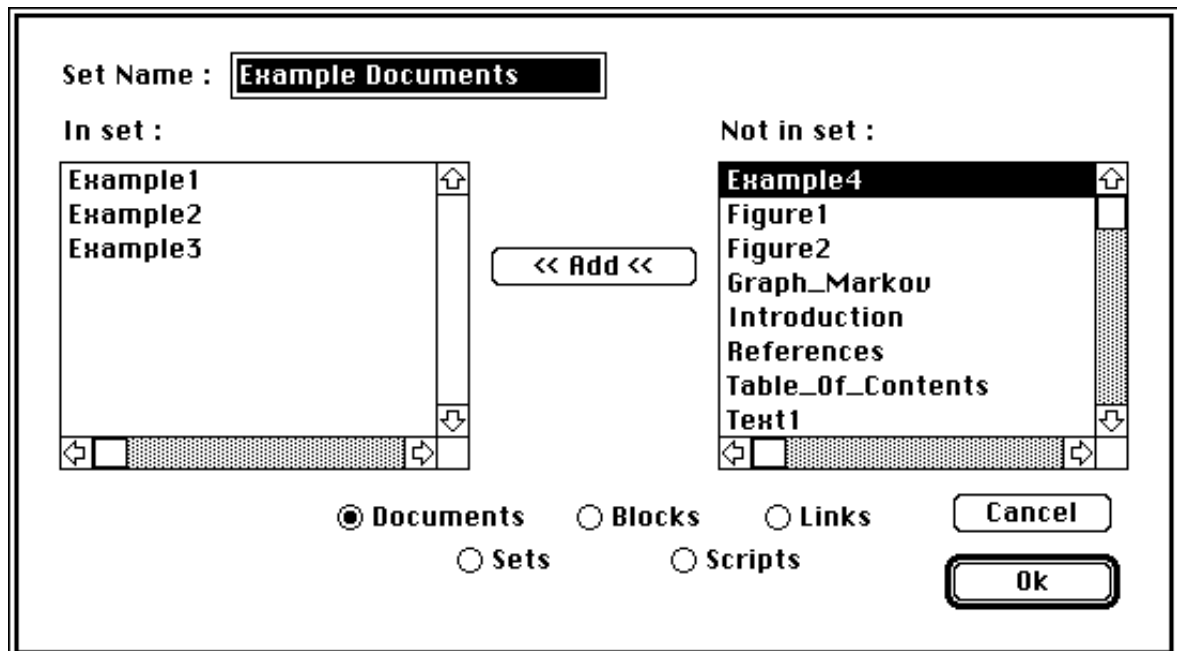


Figure 12: Dialog box for editing a set.

In this chapter, we introduce briefly the concept of scripts as it is implemented in WEBSs and describe WEBSs scripting environment.

3.1 What are Scripts ?

In its simplest form, a script is a program (a list of commands) which is executed automatically by the system. Scripts may also include sequencing instructions such as loops and conditionals, as well as calls to predefined routines and other scripts. In WEBSs, there are two kinds of scripts : unbound scripts and triggered scripts.

3.1.1 Unbound Scripts

Unbound scripts are launched directly by the user. They allow you to regroup a complex set of actions into a single command and to play it back at will in different contexts; they represent a kind of high-level² macro facility, as is available in many applications. Such a script could for instance automatically create a glossary by searching for all occurrences of specific words in a group of documents and creating links to a document where their meaning is explained. Unbound scripts can also be invoked by other scripts (see §4.2.2).

3.1.2 Triggered Scripts

The idea of triggered scripts is simple : you can decide that a script will be automatically executed when a certain action is performed by a specific object, like the opening of a document. This enables you to affect the way an object reacts to a given action. For instance, the standard behavior when following a link is to show the link endpoint. But a script can be attached to a link named “Special Link #1” such that following it displays a message on the screen before going to the endpoint. The actions that can trigger a script are listed in Appendix C.

A triggered script can be attached not only to an individual object, but also to all objects in a specific set, or to all objects of a certain class. Thus, triggered scripts allow you to obtain a common behavior from a group (set or class) of objects with a minimal effort. For example, you

² WEBSs scripts consist of high-level operations such as the closing of a document, and not low-level actions like a mouse click at location (x,y).

can write a script to add any textual document when it is opened to a set named “Visited Text Documents”, which may later be used for miscellaneous purposes (history, searching)

Since WEBSs scripting system relies heavily on object-oriented programming, we give below a short introduction to the main concepts of object-oriented programming. Readers already familiar with these ideas can go directly to the description of the scripting environment.

3.1.3 Object-Oriented Programming Fundamentals

We said above that all the elements that constitute an IEB – i.e. documents, blocks, links, sets and scripts – are objects. In object-oriented programming, *objects* are entities that contain data – in WEBSs these are the attributes of an object such as its name, id, creator, etc. – and can execute actions in response to *messages* they receive. The code that tells an object how to react when it receives a message is called a *method*. A document, for instance, when it receives a Close message, calls its Close method, which will ask the user if he wants to save changes and close the document window.

Groups of objects with common characteristics are called *classes*. All the objects that belong to the same class have the same set of attributes. They also know how to behave in response to the messages the class contains, e.g. all links know what to do when they receive the Follow message. Individual objects are referred to as members of a class.

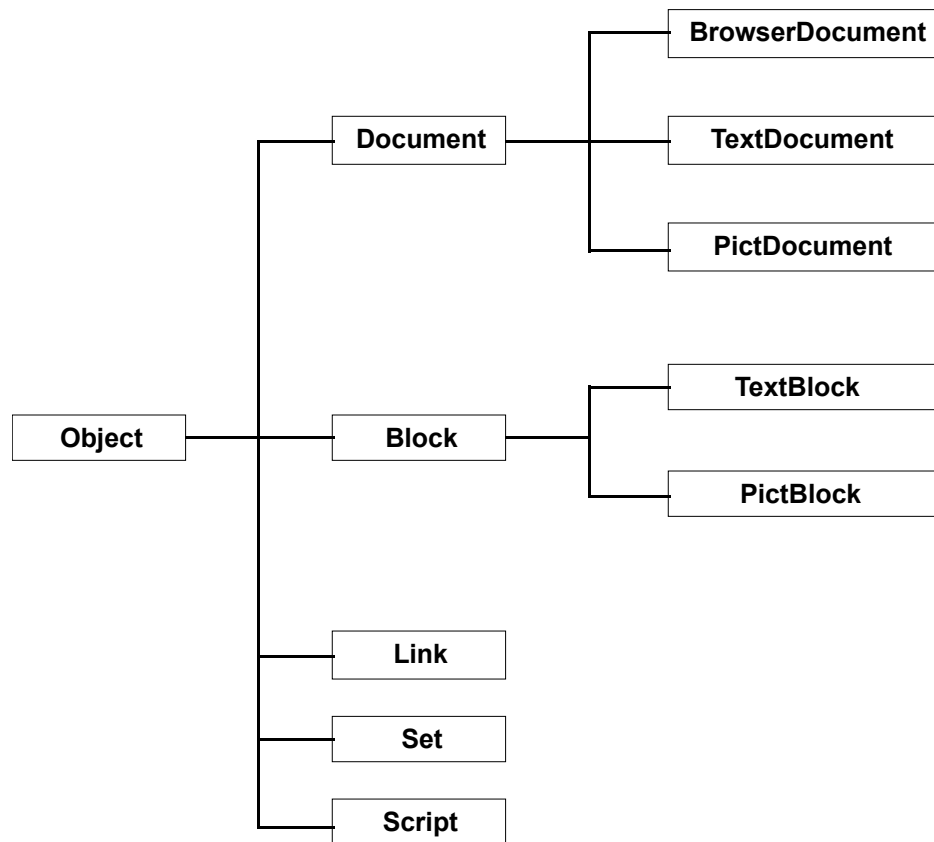


Figure 13: WEBSs hierarchy of classes. The class Object is the ancestor of all other classes.

The concept that makes classes significant is *inheritance*. In an object-oriented world, objects inherit attributes and behavior from their ancestors in a chain of heredity. Figure 13 shows WEBSs hierarchy of classes. As you can see, the class Document has subclasses called PictDocument, TextDocument and BrowserDocument. The key idea in inheritance is that if the class Document contains a method called, for example, Close, every member of every subclass of Document can use that method. If you send the message Close to a member of class TextDocument, it will simply pass the message up the hierarchy until there is an

ancestor class (here Document) which has a method named Close. Appendix C contains a list of all the methods contained in the different classes.

Note that while you cannot define new classes, you can create your own categories of objects by adding new values to the list of user-defined types (see §2.6.2).

3.2 Working with Scripts

The main component of the scripting environment is the script browser. This is where most script-related operations take place: creation of scripts, edition, syntax checking, etc. The script browser is described below.

Since scripts may be attached to objects and sets, you should also know how to get information about objects (see §2.6), and how to create and edit sets (see §2.7).

3.2.1 The Script Browser

The script browser (see Figure 14), provides rapid access to all the scripts in an IEB. It can be shown and hidden with the **Show Script Browser** resp. **Hide Script Browser** commands from the **Script** menu.

In the pop-up menu on the top left, you select the kind of scripts you want to visualize : unbound scripts, or triggered scripts for classes, sets or individual objects. Depending on this choice, the left pane shows a list of classes, sets or object names. By choosing an item in this list, the corresponding list of scripts is displayed in the right pane, showing their name and header. The contents of the script selected from this list is displayed in the lower pane, where it can be edited.

- The Unbound Script category contains all unbound scripts, as well as all scripts that have never been compiled successfully. When this item is selected in the pop-up menu, the left pane contains exactly one item, also named Unbound Scripts, which is automatically selected.
- When you select the Classes, Sets or Objects items in the left pop-up menu, the left pane shows only a list of the classes, sets or objects for which there are scripts. As new scripts are created, the names of the classes, sets and objects to which they are attached are automatically inserted into the corresponding list.
- The list of scripts displayed in the right pane can be sorted by script name or by

triggering method by selecting the corresponding item in the Order pop-up. Unbound scripts, which have no triggering method, are always sorted by name.

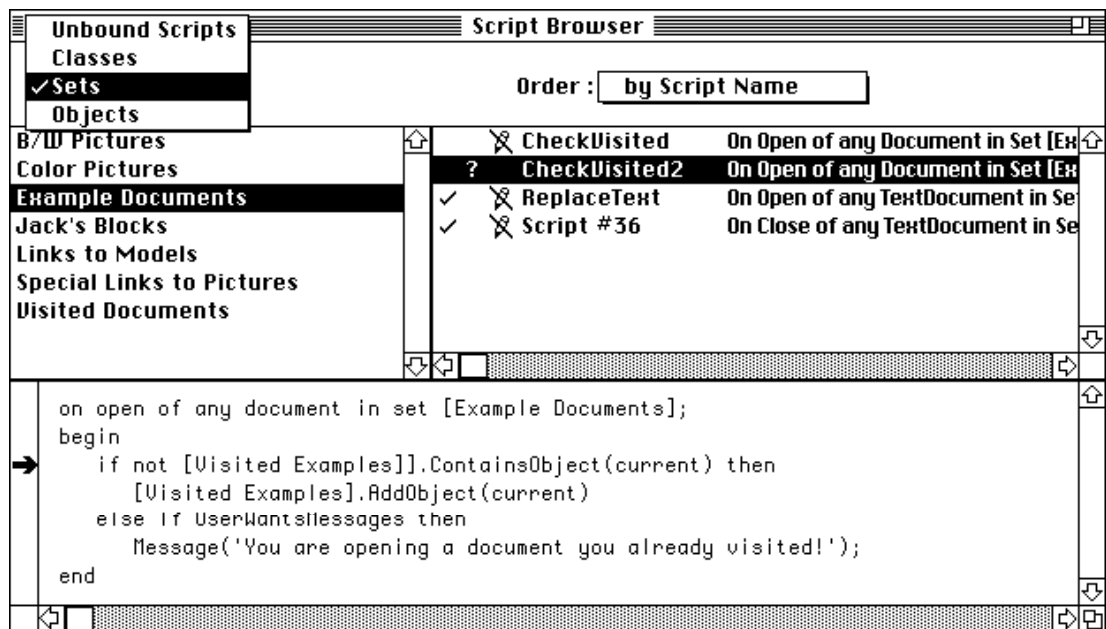



Figure 14: The script browser. A list of scripts for the set “Example Documents” is displayed in the right pane, and script “CheckVisited2” is currently being edited. The arrow on the left indicates the line where a syntax error was found.

The symbols that appear on the left of script names have the following meanings:

- A check mark indicates that the script is enabled and will automatically be executed when the triggering conditions defined by its header are met. Clicking on the check mark toggles its state, and allows you to turn on and off the automatic execution of individual scripts.
- ? A question mark is displayed if a script has not been compiled or contains errors.
-  Public scripts – created by the author of an IEB – are indicated by a read-only icon. While readers cannot modify these scripts, they can enable and disable them, copy their contents into the Clipboard and paste it in a reader’s script, or duplicate them.

3.2.2 Creating a New Script

Choose the **New Script** command from the **Script** menu. If a script was being edited, a dialog box will ask you if you want to save changes.

Alternately, if no script is selected, you can just start typing the text of the script in the lower pane, which will be empty.

3.2.3 Compiling a Script

To compile a script, choose the **Compile Script** command from the **Script** menu.

Two possibilities may arise:

- If the script contains no error, it will be saved in the web. If it is the first time it is compiled successfully, WEBSs will give it a name, and put it at the right place in the script browser; WEBSs will infer its kind and related information (triggering method and class, set or object to which the script is attached) from its header, and, if necessary, insert a new class, set or object name in the left pane.
- If the script compiler detects an error in the script, it will display a dialog box with a self-explaining error message (such as “Missing identifier”), and put an arrow on the left edge of the text pane, where the error was found (see Figure 14). In this case, the script will not be saved in the web (to save an uncompiled script, see §3.2.4).

3.2.4 Saving a Script

To save a script without compiling it:

Choose the **Save Script** command from the **Script** menu.

A question mark ? will be displayed on the left of the script name to indicate that it has not yet been compiled successfully (see Figure 14). If the script is saved for the first time, WEBSs will give it a name and place it in the Unbound Scripts category, where you can find it later. Otherwise, the script will remain where it was in the script browser.

Note: When you choose the Save Script command or successfully compile a script with the Compile Script command, the script is “saved” in the active web. The web, however, is not written to disk until you choose the Save Web command from the WEBSs menu. Therefore, we recommend that you use the Save Web command regularly when you create and modify scripts.

3.2.5 Other Operations with Scripts

To revert a script to the version previously stored in the web, choose the **Revert Script** command from the **Script** menu.

To duplicate a script, choose the **Duplicate Script** command from the **Script** menu. A reader can duplicate an author’s script.

To delete a script, choose the **Delete Script** command from the **Script** menu.

To change the name of a script, choose the **Get Script Info** command from the **Script** menu. Note that scripts must have unique names.

3.2.6 Executing Unbound Scripts

To execute an unbound script, you have two possibilities :

- a) In the script browser, with the Unbound Scripts category displayed in the pop-up menu, select an error-free script (without a ? on its left) and choose the **Execute Script** command from the **Script** menu.
- b) Choose the **Execute Other Script** command from the **Script** menu. A dialog box will appear, showing you a list of all the executable unbound scripts. You can then select a script to execute.

The execution of a script can be aborted by pressing ⌘-. (command-period).

Note: Only unbound scripts without parameters that don't return a result can be executed directly. Other unbound scripts must be invoked by another script (cf. §4.2.2).

3.2.7 Automatic Execution of Scripts

Scripts that are attached to an object, a set or a class of objects will be launched automatically by the system when the triggering conditions defined by their header are met. If several scripts are triggered by the same object/action pair, they are all executed successively. The order of execution goes from general to specific scripts: first class-bound scripts, then set-bound scripts, and finally object-bound scripts.

If nothing happens when you execute an action that should trigger a script, check the following elements :

- Verify that the automatic execution of scripts is globally enabled (i.e. the **Automatic Script Execution** item from the **Script** menu is checked).
- Verify that the script has been successfully compiled, i.e. that there is no ? on its left in the script browser.
- Verify that the script is individually enabled, i.e. that it has a . To enable a script, click on the left of its name, near the edge of the pane, where the check marks are displayed (cf. Figure 14). A script cannot be enabled if it contains errors.
- Verify that the class, type³ and/or object name used in your script header corresponds to the class, type or name of the object that should trigger the script, resp. that the object really belongs to the set mentioned in the script. To check this, use the object information dialog box which shows the class, type and

³

You can also decide that only objects with a specific type will trigger a script (cf. §4.2.1).

name of an object, as well as all the sets to which it belongs and all the scripts that apply to it (see §2.6). Use the set editor to put an object into a set (see §2.7).

This chapter presents the essentials of the scripting language. See Appendix D for examples of scripts.

4.1 Basics

The scripting language is very similar to Pascal, including variable declarations and statements, with a few additions such as a dot notation to invoke methods on objects (as in Object Pascal), object specifiers to access objects by name, and a FOR EACH construct to perform a group of statements on every object in a set.

A script consist of textual units (terminals) which are classified into special symbols (+, −, >, etc.), identifiers, objects specifiers, numbers and strings. Two terminals are separated by a space, a comment or an end of line. However, this is mandatory only in those cases where the lack of such a separator would merge the two terminals in one. For example, in “IF x = y THEN” spaces are nessary in front of x and after y, but could be omitted around the equal sign. Reserved words, special symbols and predefined identifiers are listed in Appendix C.

An **identifier** consists of a letter followed by zero or more letters, digits or underscores. There are three categories of identifiers : reserved words, predefined identifiers, and identifiers created by the programmer. Identifiers are not case sensitive – e.g. “anIdent” and “ANIDENT” represent the same identifer – and have a maximal length of 63 characters.

An **object specifier** is the name of an object placed in square brackets, e.g. [document1]. While upper and lower case letters are equivalent, all the characters inside the brackets are significant, i.e. [document1] and [document1] do not refer to the same object. The object needs not exist when the script is compiled. At execution time, the system searches among all the objects in the open web(s) for an object with the specified name

and generates an error if none is found. If you are unsure whether the open webs contain an object with a given name, you can use the predefined routine `GetNamedObject` (see Appendix C).

Numbers have only integer values and consist of one or more digits.

A **string** is a sequence of 0 to 255 printable characters placed between single quotes. If a quote is to be included in a string, it must be doubled, as in 'This " is a quote in a string'. The sequences of characters “\n” and “\N” in a string are interpreted as an end-of-line character (this can be useful if you want to display a message on two lines, or if you want to insert blank lines in a text document).

Comments are delimited by curly brackets (“{” and “}”) and can be nested.

4.2 Structure of a Script

A script consists of two parts, the script header and the main block (in the following, we use a standard EBNF notation):

```
Script ::= ScriptHeader ";" MainBlock
```

- The script header defines the object(s) to which the script is attached and the method that will trigger its execution,
- The main block contains declarations of variables and a list of statements to be executed when the script is invoked.

4.2.1 The Script Header

Script headers can have four different formats. The first three formats, used in triggered scripts, specify the triggering method and whether the script is attached to all objects of a certain class, to all objects in a certain set, or to an individual object. The last format is reserved for unbound scripts.

Scripts for all objects of a class

Format of the header:

```
"ON" MethodIdentifier "OF" "ANY" ClassIdentifier [ TypeSpec ]
```

Example :

```
ON Open OF ANY TextDocument
```

A script beginning with this header will be triggered when any member of class **TextDocument** receives the **Open** message (the list of the triggering methods can be found in Appendix C).

If we specified **Document** as the class, then the script would be triggered by any member of a subclass of **Document** (e.g. **TextDocument** or **PictDocument**). More generally, every object that belongs to a subclass of the class mentioned in a script header can trigger a script (provided that the other conditions are met).

Interesting in this header is the optional **TypeSpec** specification. The type attribute of objects may serve as a selector in a header to assign a different behavior to distinct types of objects; if a type is specified in a script header, only objects of that type will trigger the script. For example, a script beginning with a header of the form **ON Open OF ANY Document OF TYPE Reference** will only be activated by a document whose type is **Reference**. The type specified in the header must exist when the script is compiled (see §2.6 for information on how to create types and modify an object's type).

Scripts for all objects in a set

Format of the header:

"ON" MethodIdentifier "OF" "ANY" ClassIdentifier [TypeSpec]
"IN" ObjectSpecifier

Example :

ON Open OF ANY TextDocument IN [ExampleDocuments]

This header means that any object from class (or a subclass of) `TextDocument` that belongs to a set named `ExampleDocuments` will activate the script.

Scripts for individual objects

Format of the header:

"ON" MethodIdentifier "OF" ClassIdentifier [TypeSpec] ObjectSpecifier

Example :

ON Follow OF Link [Example1]

With such a header, the script will be triggered when a member of class (or a subclass of) `Link` with name `Example1` receives the `Follow` message.

Note: Scripts for individual objects are bound to object names and not to the objects themselves : a script can refer to a name even if no object with this name exists⁴. This allows you to write a script in advance if you know the name of the future object. When an object is later created with the name mentioned in the header, the script will automatically be applicable. Also, if two objects have the same name⁵ and belong to the class mentioned in the header (or a subclass of it), both will trigger the script.

Unbound scripts

Format of the header:

"ON" "Execute"["(" FormalParameterList ")"] [":" TypeIdentifier]

Example :

On Execute(theDoc : Document): Integer

The optional parameter list and result type indicate that the script accepts parameters and/or returns a result.

⁴ The same rule applies for set names in set-bound scripts.

⁵ This is possible for example for two documents in different sub-folders.

4.2.2 The Main Block

The main block has the format :

```
MainBlock ::= [ VariableDefinitionPart ]
              CompoundStatement
```

Variable definition

Variables are declared with the same format as in Pascal. For example :

```
VAR    i, j, k : Integer;
        str :String;
VAR    doc :    Document;
```

The scripting language includes the types `Integer` (with allowed values ranging from -2^{31} to 2^{31}) and `String`, as well as all the classes from Figure 13, plus any classes specific to the models available in your version of WEBSs.

A variable declared with type `C`, where `C` is a class from Figure 13, may be assigned any expression whose type is a subclass of `C`. For example, a variable of type `Object` can contain any object in an IEB.

WEBSs also provides some global variables (listed in Appendix C). In particular, statements within a triggered script may use the variable `current` to refer to the object that triggered the current execution. Global variables cannot be modified by a script.

Compound statements

A compound statement is a sequence of statements separated by semicolons and placed between the two reserved words `BEGIN` and `END`.

```
CompoundStatement ::= "BEGIN" Statement { ";" Statement } "END"
```

The syntax for statements is similar to that of Pascal, including loops, conditions and assignments.

Example :

```
BEGIN
    i := 1;
    WHILE i <= n DO
        BEGIN
            i := i + 1;
            FOR k := 1 TO i DO count := count + k;
        END;
    IF i = 10 THEN count := count + 1 ELSE count := count - 1;
    REPEAT
        ...
    UNTIL count > i
END
```

The FOR EACH statement

To execute a group of statements on every object in a set, a special construct exists :

```
ForEachStatement ::= "FOR" "EACH" VariableIdentifier [ "OF" "TYPE" Factor ]  
                  "IN" Expression DO Statement
```

Example :

```
VAR doc : Document;  
...  
FOR EACH doc OF TYPE Annotation IN authorWeb DO  
    doc.Open;
```

This statement will call the **Open** method for every member of a subclass of **Document** which belongs to the set **authorWeb** (a predefined global variable) and has the type **Annotation**.

Note: Whenever the type attribute of an object (not to be mistaken with its class) is required in a statement (e.g. in a **SetType** call or a **FOR EACH** statement), you can use either an existing type – as in the above example –, a call to the **GetType** method with an object, or an integer variable in which you have stored a type (the **SetType** and **GetType** methods are described in Appendix C). No control is made to ensure that the integer value corresponds to an existing type.

Expressions

While there is no real boolean type, the logical operators **AND**, **OR** and **NOT** and the relational operators **=**, **<>**, **<**, **<=**, **>**, **>=**, and **IN** can be used in expressions. When evaluating an expression, any value different from 0 is considered true, and 0 is considered false (two predefined constants **false** and **true** exist with the values 0 and 1).

- The **NOT** operator, applied to an expression, returns 1 if the expression is 0 and 0 otherwise.
- The logical operators **AND** and **OR** work with the binary representation of integers.
- The **/** operator computes the integer division.
- The **IN** operator tests whether an object belongs to a set.

Examples : 4 > 3 gives 1; NOT (3<=5) gives 0;
 NOT 5 gives 0; NOT 0 gives 1;
 3 OR 5 gives 7, because $3_2 = 011$, $5_2 = 101$ and $011 \text{ OR } 101 = 111$;
 3 / 2 gives 1;
 [Example1] IN [TheExamples] gives 1 if the object is in the set, 0 otherwise.

Expressions are evaluated from left to right. The operators have different priorities, from the highest to the lowest :

- the negation : **NOT**,

- the multiplicative operators : AND, *, /,
- the additive operators : OR, +, −,
- the relational operators : =, <>, <, <=, >, >= and IN.

You can compare strings with the = and <> operators, and concatenate them with the + operator.

You can also compare objects with the = and <> operators, and test for the validity of an object reference by comparing it with the constant `nullObject`.

Routine calls

Predefined routines – some of which return a result – are listed in appendix C. A routine call has the format :

RoutineCall ::= Identifier ["(" ActualParameterList ")"]

The number and type of the actual parameters must match the formal parameters used in the declaration of the routine. There are two kinds of parameters : variable and value parameters. In the case of variable parameters (preceded by the symbol **VAR** in the routine declaration), the actual parameter must be an identifier denoting a variable. The formal identifier used in the routine then stands for that variable (call by name). If the parameter is a value parameter, the corresponding actual parameter must be an expression (of which a variable is a special case). The expression will be evaluated prior to the routine activation, and its value will be assigned to the formal parameter which now constitutes a local variable.

Examples :

```
VAR theString : String;
BEGIN
    DoMenu('New Text Document');
    IF ReadString('Your name?', theString, 'myName') THEN ...
```

Method calls

You can call a method of an object using a dot notation. The object can be referenced by an object specifier or a variable. It can also be the result of a routine call. The format for a method call is the following :

ObjectAccess ::= ObjectSpecifier | VariableIdentifier | RoutineCall
MethodCall ::= ObjectAccess "." Identifier ["(" ActualParameterList ")"]

The methods contained in each class are listed in Appendix C. As was said in §3.1.3, a method M contained in a class C is inherited by all its descendants, i.e. every member of every subclass of C can call method M.

Note that attributes of objects are also accessed through methods (e.g. there are **SetName** and **GetName** methods), and that some methods have parameters and/or return a result.

Examples :

```
theLink.SetName('Special Link #1');  
IF [Example1].IsOpen THEN ...;
```

Note: When you use an object specifier, the system searches at run-time for *any object* in the open webs whose name matches the name you specified. If you have several objects with the same name, you cannot know whether the first object the system will find will be the right one, and an error may occur. Therefore, you should be careful when giving names to objects and using object specifiers. The routine `GetObject` allows you to search for an object with a specific (unique) id (see Appendix C).

Declaration and call of unbound scripts

While the routines and methods that can be called by scripts are predefined (i.e. you cannot create new ones), you can write unbound scripts, possibly with some parameters and a result, and use them as global procedures.

Here is an example of an unbound script with one parameter and a result:

```
ON Execute(name : String):Document;  
BEGIN  
    theDoc := ...;  
    IF name = 'example' THEN RETURN theDoc  
    ELSE ...;  
END
```

If the script returns a result, then it must contain at least one return statement, which consists of the symbol `RETURN` followed by an expression. A return statement terminates the execution of the script, and the expression specifies the value returned to the calling script.

The format of a call to an unbound script (which must have been successfully compiled) is similar to a standard method call. You must use an object specifier (a variable is not allowed here) with the name of the script to be invoked, followed by a dot, the method identifier `Execute` and any actual parameters. If the above script is named `Test`, then a possible call is:

```
VAR aDoc : Document;  
BEGIN  
    ...  
    aDoc := [Test].Execute('example')  
    ...  
END
```

Rules for type compatibility of classes

The following rules apply for the compatibility of classes in assignments, method and routine calls, and RETURN statements :

- In an assignment to a variable of class C, the right side must be an expression whose type is C or a subclass of C. For instance, assume that *aBlock* is a variable of type **Block** and *anExpr* an expression of type **TextBlock**. This assignment is OK:

```
aBlock := anExpr
```

since the class of *aBlock* is an ancestor of the class of *anExpr*. If *anExpr* is an expression of type **Object**, the above assignment is not OK, because **Object** is not a subclass of **Block**. However, you can “change the type” of the expression to **UNIVObject**⁶ with the **AsUNIVObject** routine (see Appendix C). **UNIVObject** is a special type, which is compatible with any class. Thus, if *anExpr* is still of type **Object**, you can write :

```
aBlock := AsUNIVObject(anExpr)
```

This makes *anExpr* appear to be of class **UNIVObject**, which is compatible with class **Block**. Note that if *anExpr* doesn’t actually contain an object of class **Block**, you will have problems later when using the variable *aBlock*.

- The rule is similar for the type compatibility of value parameters in a method or routine call⁷: if the formal parameter is of class C, the actual parameter must be an expression whose type is C or a subclass of C. Again, you can make an expression compatible with any class with a call to the **AsUNIVObject** routine.
- The same rule applies for RETURN statements in unbound scripts: the expression returned must be of the same class or a subclass of the class that appears after the colon in the script header. You can also call the **AsUNIVObject** routine to change the type of the expression to **UNIVObject**.

In certain cases, the use of the **AsUNIVObject** routine is necessary to avoid error messages from the script compiler such as “Operands are of incompatible types” or “Type of parameter is not compatible with declaration” (see script #6 in Appendix D). This routine must be used with caution however: it allows you to bypass the type checking done by the script compiler and, if misused, it may lead to unpredictable results in subsequent statements.

Note that an object specifier also has the type **UNIVObject**. Therefore, it can be assigned to a variable of any class and is also compatible with a formal parameter of any class. You cannot declare variables or parameters of type **UNIVObject** in your scripts.

⁶ UNIV stands for universal.

⁷ For variable parameters, the actual parameter must always be a variable whose class is the same or a subclass of the class of the formal parameter.

5 WEBSs Reference Guide

The goal of this chapter is to provide a short explanatory text – a written counterpart to WEBS Balloon Help – for each of WEBSs menus and menu commands. In order to best serve this purpose, this chapter is divided into 11 sub-sections : one for each of the eleven pull-down menus which constitute the WEBSs application menu bar. The menus described within this guide are either the permanent menus of WEBS – the Apple, File, Edit, WEBSs, Script, Browser and Window menu – or those relative to textual or browser documents – the Text, Format, Node and Table menus (Figure 15). Menus related to specialized model documents (e.g. the Calculate menu for Markov documents) are not presented.

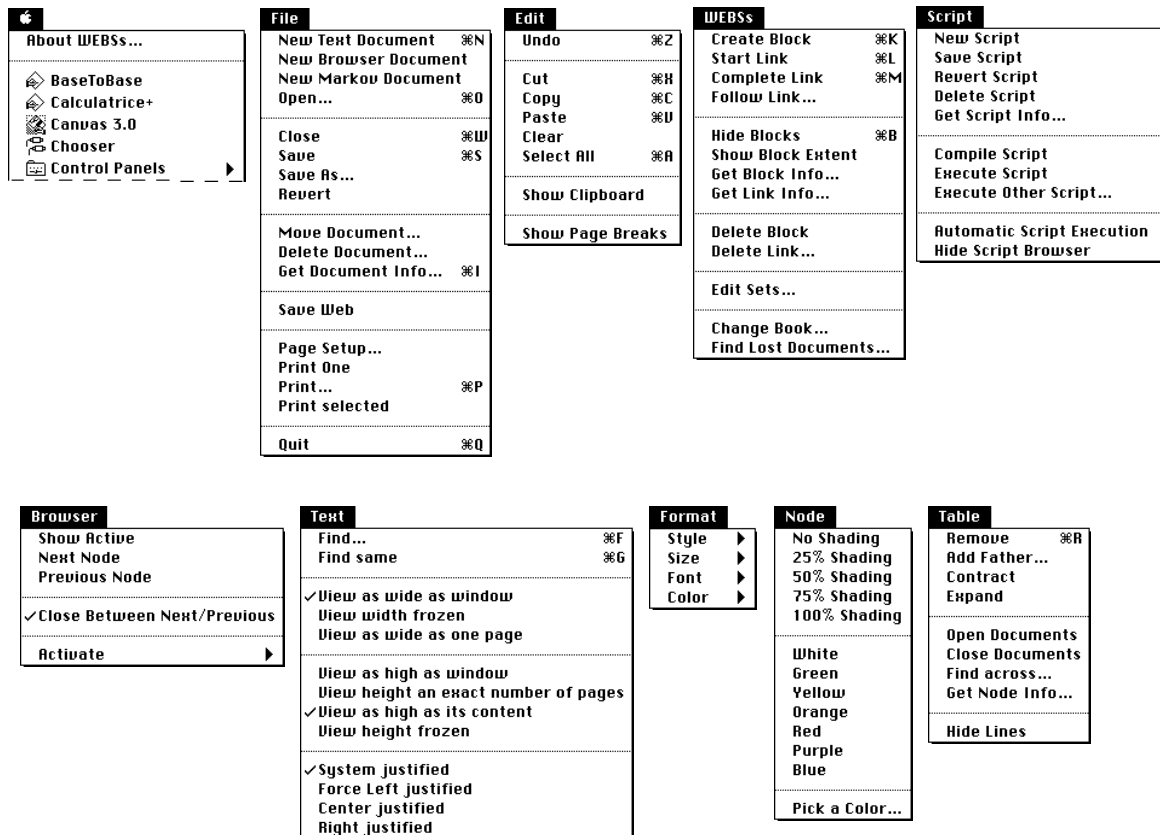


Figure 15: The main menus of WEBS. (The Window menu is not shown).

5.1 The Apple Menu

More information about WEBSs may be obtained by selecting the About WEBSs item in the **Apple** Menu. This menu also lists all the items contained in the Apple Menu Items folder.

5.2 The File Menu

The **File** menu contains commands concerning opening, saving, deleting, moving and printing documents. The active web can also be saved from this menu. **For moving or deleting documents** belonging to a web, the appropriate command must be selected from this menu.

New Text Document

This command allows you to create a new text document within the currently active web. A text document can contain texts of various styles, colors and fonts and their associated editing facilities can be compared to those of a simplified word processor. This command also opens two new pull-down menus : Text and Format.

New Browser Document

This command allows you to create a new browser document within the currently active web. A browser has a tree-structure, and its nodes can be associated with one or more documents belonging to the current author's or reader's web. This command also opens two new pull-down menus : Node and Table. See the Browser, Node and Table menus for more information about browsers.

Open

This command allows you to open any text, graphic, model or browser document, which belongs to the current author's or reader's web. This command also permits the opening of any document having a standard Macintosh TEXT or PICT format. In this latter case, you will have to respond to a dialog box in order to install the document in the currently active web.

Close

This command closes the active window.

Save

This command saves the active document on disk. Note that,

after saving a large document, it is a good and safe practice to immediately execute a Save Web command in order to memorize on disk the corresponding modifications within the active web.

Save As

This command saves the active document with a new name or in a different location. Note that the blocks and links of the old document are not copied into the new one. The Save As command can be used by a reader for copying into his/her own private web an author's document that he/she wants to modify.

Revert

This command reverts the active document to the previously saved version.

Move Document

This command allows you to rename and/or move the active document into another folder. In order to ensure the consistency of the document's web, it is important to use this command rather than the Finder for performing these actions. Note, also, that :

- The Move Document command does not permit the creation of new folders (you must create them from the Finder).
- It is not possible to move the active document to another volume than the one it is presently on.
- In the event that you still plan to move documents with the Finder, see the Find Lost Documents command of the WEBSs menu.

Delete Document

This command allows you to delete the active document. In order to ensure the consistency of the document's web, it is important to use this command rather than the Finder for performing this action. Naturally, a reader cannot delete an author's document.

In the event that you still plan to move documents with the Finder, see the Find Lost Documents command of the WEBSs menu.

Get Document Info

This command brings up an editable dialog box that displays information about the current document.

Save Web

This command saves the active web, i.e. the author's web or the private reader's web. A dialog box will, if necessary, prompt you to save all the open documents which have been modified since the last saving. It is good and safe practice to apply this command from time to time, particularly if important changes have been made.

Page Setup

This command sets paper size, orientation and special printing effects for the active document.

Print One

This command produces an immediate printing of all pages of the active document.

Print

This command allows you to print selected pages of the active document.

Print Selected

This command prints all the documents associated with selected nodes in a browser document. In order to perform this command :

- the active window must contain a browser,
- at least one of the selected nodes must have an associated document.

Quit

This command ends the current WEBSs session. WEBSs prompts you to save changes to any open documents and automatically saves the active web.

5.3 The Edit Menu

The **Edit** menu contains commands for changing the content of documents, and undoing and repeating actions.

Undo

This command reverses the most recent action you performed in a document.

Cut

This command deletes the selection and stores it on the Clipboard, replacing any existing Clipboard contents. Blocks and links are not copied from a textual document.

Copy

This command copies the selection to the Clipboard, replacing any existing Clipboard contents. Blocks and links are not copied from a textual document.

Paste

This command inserts the contents of the Clipboard into the active document at the insertion point or in place of the selection. Blocks and links cannot be pasted into textual documents.

Clear

This command deletes the selection without placing it on the Clipboard.

Select All

This command selects the whole contents of the active document or script.

Show Clipboard/Hide Clipboard

This command displays/hides the Clipboard window.

Show Page Breaks

This command, when checked, displays page breaks.

5.4 The WEBSS Menu

The **WEBSS** menu contains all the commands concerning blocks and links. All actions take place within the currently active web, i.e. the author's web or the private reader's web. This menu also contains commands for editing sets, and for restoring the consistency between the documents and the database of an IEB.

Create Block

This command creates a block out of the current selection and inserts a block marker in the document.

Start Link

This command begins the process of creating a link. This process is completed by the Complete Link command. If a block is selected, it will become one end of the future link. If other objects are selected, this command creates a block out of these items.

Complete Link

This command terminates a link initiated by the Start Link Command.

Follow Link

This command traverses the link associated with the selected marker (•); i.e. it opens/activates the document containing the destination point and scrolls to the position of the corresponding marker. If the marker is associated with more than one link, a dialog box allows for selecting the appropriate one. This command can also be executed by double-clicking on the marker.

Hide Blocks/Show Blocks

This command temporarily removes all the block markers (•) from the current window. After the Hide Blocks command is selected, it changes to Show Blocks so that the block markers may be redisplayed at any time.

Show Block Extent

This command shows the exact selection which corresponds to the selected block marker (•). This command is particularly useful for graphic documents for which the block marker can be freely moved independently of their associated extent.

Get Block Info

This command brings up an editable dialog box that displays information about the block corresponding to the selected block marker (•).

Get Link Info

This command brings up an editable dialog box that displays information about the link attached to the selected block marker (•). If the block is associated with more than one link, a dialog box allows for selecting the appropriate one.

Delete Block

This command removes from the active web the block corresponding to the selected block marker (•). All links attached to this block are also deleted. Naturally, a reader cannot delete an author's block.

Delete Link

This command removes from the active web the link associated with the selected block marker (•). A dialog box allows for selecting the appropriate link to be deleted if there is more than one. Naturally, a reader cannot delete an author's link.

Edit Sets

This command brings up a dialog box that displays all the sets in the open web(s). You create and edit sets from this dialog.

Change Book

This command allows you to activate a different IEB than the current one, or to change your user name, without having to quit the WEBSs application. Note that this action will force you to close all of your currently opened documents.

Find Lost Documents

This command forces WEBSs to check the list of documents in the active web. Each time a document of the list is not on the disk at the location indicated by the web, a dialog box prompts you, either to remove the document from the web, or to indicate its new position. Note that this command does not check the author's web in the event that you are currently identifying yourself as a private reader.

5.5 The Script Menu

The **Script** menu contains commands for creating, saving, compiling, and executing scripts.

New Script

This command creates a new empty script.

Save Script

This command stores the script being edited into the active web. The web, however, is not written to disk until you choose the Save Web command from the WEBSs menu. Therefore, we recommend that you use the Save Web command regularly when you create and modify scripts.

Revert Script

This command reverts the script being edited to the version that was previously stored in the active web.

Duplicate Script

This command creates a copy of the selected script.

Delete Script

This command deletes the selected script.

Get Script Info

This command brings up an editable dialog box that displays information about the selected script.

Compile Script

This command checks the syntax of the script being edited. If an error is found, it is displayed in a dialog box. If the script does not contain any errors, it is automatically saved into the active web.

Execute Script

This command executes the selected script. Only unbound scripts can be executed from this menu. The execution of a script can be aborted by pressing ⌘-. Only unbound scripts without parameters that don't return a result can be executed directly.

Execute Other Script

This command brings up a dialog box showing all the unbound scripts, from where you can select a script to execute. The execution of a script can be aborted by pressing ⌘-.

Only unbound scripts without parameters that don't return a result can be executed directly.

Automatic Script Execution

This command, when checked, activates the automatic execution of scripts : if a triggering action is performed by an object with an attached script, then the script will be executed.

Show/Hide Script Browser

This command shows/hides the script browser.

5.6 The Browser Menu

The **Browser** menu contains commands which allow you to activate and use all the browser documents available in the open web(s).

Show Active

This command brings the active browser to the front .

Next Node

This command :

- finds the last selection made within the active browser,
- moves to the next node with associated documents, selects it,
- opens the corresponding documents,
- if the Close Between Next/Previous item from the Browser menu is checked, closes the document associated with the previous selection.

This command facilitates sequential reading with respect to the active browser.

Previous Node:

This command :

- finds the last selection made within the active browser,
- moves to the previous node with associated documents, selects it,
- opens the corresponding documents,
- if the Close Between Next/Previous item from the Browser menu is checked, closes the document associated with the previous selection.

This command facilitates sequential reading with respect to the active browser.

Close Between Next/Previous

This command tells WEBSs to close the documents associated with the previous selection after the execution of a Next Node resp. Previous Node command.

Activate

This command displays a sub-menu composed of a list of all the existing browser documents available in the open web(s). Once one of these browsers is selected, it is

opened – if necessary – and becomes the active browser, i.e. the one used by the Show Active, Next Node and Previous Node commands of the Browser menu. It is also labelled in the list with a check mark. Note that the list of available browsers presents first the author's ones and secondly the reader's ones.

5.7 The Window Menu

The Window menu provides you with a chronologically ordered list of all your open documents and allows you to activate one of them. The active document is labelled with a check mark. The names of documents that have been modified and not saved are underlined.

5.8 The Text Menu

The **Text** menu contains commands for modifying the way texts are displayed in a text window. This menu also contains commands for finding strings within the active text document. In order for you to comfortably read a text document on the screen, you should choose the “View as wide as window” option.

Find

This command allows you to search for a string within the active text document. For a search across several documents, use the Find across command from the Table menu.

Find Same

This command searches for the same string as the most recently requested in a Find command.

5.9 The Format Menu

The **Format** menu contains commands for changing the style, size, font and color of characters in a textual document.

5.10 The Node Menu

The **Node** menu contains commands for changing the shading

and color of the nodes in a browser document.

5.11 The Table Menu

The **Table** menu contains commands for editing and using browser documents. Check the New Browser Document command in the File menu for more information on this type of document. Note that some commands (Contract, Expand, Open Documents, Close Documents and Get Node Info) have key shortcuts. Note also, that you can use commands from the Edit menu in order to cut, copy and paste nodes.

Remove

This command removes the selected nodes from the document. When a node is removed, what happens with its sons depends on whether the node was contracted or not. If it was contracted, then its sons are also deleted. If it was not contracted, then its sons are connected with its father. This second possibility allows you to remove a node without removing its sons.

Add Father

This command inserts a node between the selected node and its father.

Contract

This command hides all the sub-nodes of the selected node(s). This command can also be executed with a ⌘-double-click on the node. Note that a simple double-click opens the associated documents.

Expand

This command displays (up to one hierarchical level deeper) the hidden sub-nodes of the selected node(s). Note that it is easy to recognize an expandable node by the ellipsis (...) under its name. This command can also be executed with a ⌘-double-click on the node. Note that a simple double-click opens the associated documents.

Open Documents

This command opens the document associated with the selected node(s). This command can also be executed by double-clicking on the node. If a node is contracted, all the documents associated with its sub-nodes will be opened.

Close Documents


This command closes the document associated with the selected node(s). This command can also be executed with a control-double-click on the node. If a node is contracted, all the documents associated with its sub-nodes will be closed.

Find across

This command allows you to search for a string across all the text documents associated with the selected nodes. In order to perform this command :

- the active window must contain a browser,
- at least one of the selected nodes must have an associated document.

Get Node Info

This command brings up a dialog box which allows for editing the selected nodes. This command can also be executed with a -double-click on the node.

Hide Lines/Show Lines

This command hides/shows the lines that connect the nodes.

Appendix A: Getting Started with WEBSs: the Online Tutorial

The goal of this appendix is to explain to you how to start the online tutorial, which introduces you in a step-by-Une icons.

A.1 Actions

To install and use the WEBSs application and its online tutorial, you must:

1. Copy the “WEBSs Books” folder (either from the original WEBSs diskette or through another mean such as ftp) on your hard disk⁸.
2. Open the “WEBSs Books” folder and start the WEBSs application. WEBSs initial dialog box will appear (see Figure A-1).
3. Verify that the current active IEB is “Tutorial” as shown in Figure A-1. Otherwise, modify it by clicking on the name which appears instead of “Tutorial” in the Book pop-up menu.
4. Replace the “author” user name by a name of your choice. By doing this, WEBSs will consider you as a private reader and you will be unable to damage the original documents of the tutorial. This is important if other users plan to follow the online tutorial on your system. Note that user names are not case sensitive.
5. Click the OK button.

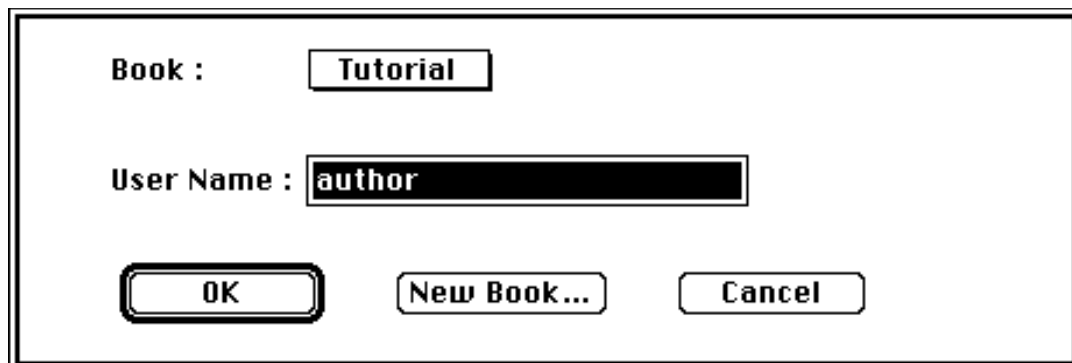


Figure A-1: WEBSs initial dialog box

6. Open the document entitled “1.First”. In order to perform this action, you must use the **Open** command of the **File** menu, select the document “1.First” in the “Texts” folder inside the “Tutorial” folder and click the Open button.

⁸ You can also run WEBSs from a diskette. In this case, you should use a copy of the original diskette (WEBS is not copy-protected). However, WEBSs is a large application and this makes it run more slowly. Therefore, we recommend that you use WEBSs on a hard disk.

7. Follow the instructions which appear on your screen. Good luck!

A.2 Remarks

- If you get tangled up in WEBS, use **Balloon Help**, which is available for all menus and menu commands (Balloon Help requires System 7).
- The disposition of the windows within the online tutorial has been optimized for a 13" monitor (i.e. the standard Mac II color monitor). Therefore, if you are using a smaller (or larger) monitor, you may want to reorganize your windows.
- If you want to move the tutorial to another location on your disk, remember that both the IEB database and the IEB main folder (e.g. the "Tutorial.book" file and the "Tutorial" folder) must always be at the same location on disk as the WEBSs application. For other problems, see Appendix B.

Appendix B: Trouble Shooting with WEBSs Databases

This appendix describes the inconsistencies which may arise between the WEBSs database and the concrete documents on your disk and gives you tips for rectifying them.

Questions/Answers

Question 1: What should I do if I have moved, renamed or deleted one or several documents of an IEB using the Macintosh Finder ?

Answer 1: Use the Find Lost Documents command of the WEBSs menu as explained in §5.4. Do not hesitate to make use of this command if you feel that WEBSs is having difficulties finding the documents of an IEB. It takes only a few seconds and does not damage the IEB database. Remember also that both the IEB database (i.e. the “AnyIEB.book” file) and the IEB main folder must always be at the same location on disk as the WEBSs application.

Question 2: What should I do if the blocks⁹ of an IEB document are no longer at their correct position ?

Answer 2: Such a problem may arise if, after having modified a document, you save it and then experience an unexpected crash before you have the chance to save its web. The best solution in such a case, is to delete all the blocks causing difficulties and then to recreate them, as well as their associated links. Normally, this would only happen within text documents and, if as recommended at the beginning of Chapter 2, you avoid creating oversized text documents, it should not take you too long to correct the problem.

Question 3: What should I do if I have lost or damaged an IEB database and I can no longer open all its documents ?

Answer 3: WEBSs can open a Macintosh document in the two following situations :

⁹ By blocks, we mean both the block markers and the block extents.

- S1. The document does not belong to WEBSs (i.e. it does not have the “WEBS” creator’s signature), but its file type (e.g. TEXT or PICT) is compatible with WEBSs. In this case, when you open the document for the first time with WEBSs, its creator’s signature is automatically changed to “WEBS” and the document access path is simultaneously memorized into the current active web.

- S2. The document already belongs to WEBSs. In this case the document should also belong to one of the webs of the IEB and can only be opened if the web is still operational. If the web is not operational, the only solution is to modify¹⁰ the creator signature of the document to something other than “WEBS”. This will solve the problem by placing you in S1 above.

¹⁰ There are many ways to modify the creator of a document. Possibilities include the ResEdit™ application program or the Disktop™ desk accessory.

C.1 Reserved Words

These words are keywords and cannot be used as identifiers :

AND ANY BEGIN DO EACH ELSE END FOR IF IN NOT OF ON OR REPEAT RETURN
THEN TO TYPE UNTIL VAR WHILE.

C.2 Special Symbols

These symbols can be used only in specific contexts and have a well-defined meaning in this context :

[] { } () < > <= >= <> = := : ; , + - / * ' ^

C.3 Predefined Identifiers

Types

The types **Integer** and **String** are predefined.

The following classes from WEBSs hierarchy are also available :

Object

Document

TextDocument, PictDocument, BrowserDocument

Block

TextBlock, PictBlock

Link

Set

Script

The type **UNIVObject**¹¹ is also predefined (but you cannot define variables or parameters of type **UNIVObject**). Whenever it appears in the following, it means that the type of the object returned by a routine or method is compatible with any class. Note that an object specifier also has the type **UNIVObject**.

¹¹ UNIV stands for universal. See §4.2.2.

Constants

The following constants are available :

false

This constant, of type **Integer**, has the value 0.

true

This constant, of type **Integer**, has the value 1.

nullObject

This constant, of type **Object**, contains a null object reference. It can be used to test the validity of an object reference, e.g. as a result of a call to the **GetNamedObject** routine.

All the items that appear in the type pop-up menu of the Object Information dialog (see §2.6) are also predefined integer constants (the constant **Default** exists in every IEB).

Global Variables

These variables are read-only, i.e. you cannot change their value.

authorWeb

authorweb is a variable of type **Set** which contains a reference to the (public) author web.

current

current contains a reference to the object that triggered the current script execution. Its class is the same as that used in the script header. It is only defined in triggered scripts.

currentScript

currentScript contains a reference (of type **Script**) to the script that is currently being executed. You can use it for instance to disable a script after it has been executed.

frontDocument

frontDocument is a variable of type **Document** which contains a reference to the front document. When no document is open, it contains the value **nullObject**.

lastBlock

lastBlock is a variable of type **Block** which contains a reference to the last block created by a Create Block, Start Link or Complete Link menu command. Use it for instance to give a name or type to a block you have just created.

lastLink

lastLink is a variable of type **Link** which contains a reference to the last block created by a Complete Link menu command. Use it for instance to give a name or type to a link you have just created.

readerWeb

readerWeb is a variable of type **Set** which contains a reference to the (private) reader web. When the user is the author (i.e. no reader web is open), **readerWeb** contains the value **nullObject**.

userName

userName is a variable of type **String** which contains the name that was entered by the user in the dialog box that appears when launching WEBSs or choosing the Change Book command.

Routines

Each routine is described, along with its parameters and result type. When a formal parameter is preceded by the word **VAR**, the actual parameter must be a variable reference.

Abort

Abort stops the execution of the script. If the script was triggered by an action, the action is aborted.

AsInteger(theString : String): Integer

AsInteger returns the **Integer** representation of *theString*.

AsString(theValue : Integer): String

AsString returns the **String** representation of *theValue*. It can for example be used to display a number with the **Message** routine.

AsUNIVObject(theObject : Object): UNIVObject

AsUnivObject is provided for special cases where you want to make the expression contained in *theObject* compatible with any class. In certain cases, the use of the **AsUNIVObject** routine is necessary to avoid error messages from the script compiler such as “Operands are of incompatible types” or “Type of parameter is not compatible with declaration”. This routine must be used with caution however : it allows you to bypass the type checking done by the script compiler and may lead to unpredictable results in subsequent statements.

Beep

Beep emits a system beep.

Copy(source : String; index, count : Integer): String

Copy returns a string containing *count* characters from *source*, beginning at the position indicated by *index*. If the values of *index* or *count* are out of range or if there are not *count* characters in *source* starting at *index*, **Copy** returns the empty string.

Del(VAR dest : String; index, count : Integer)

Del removes *count* characters from the value of *dest*, beginning at the position indicated by *index*. If the values of *index* or *count* are out of range or if *index* is greater than **Length(dest)**, **Del** is ignored. If the attempted deletion extends beyond the end of *dest*, *dest* becomes truncated at *index* – 1.

DoMenu(theCommand : String)

DoMenu executes the menu command *theCommand*, if it is enabled. When the systems searches for *theCommand*, it does not distinguish upper and lower case and skips the character "...". For instance, **DoMenu**('New Text Document...') and **DoMenu**('New Text Document') will both invoke the same command. In a triggered script, it is recommended that you call the **Select** method on the triggering object to update the menus before calling this routine.

GetIdObject(theId : Integer): UNIVObject

GetIdObject searches in the open web(s) for an object with id *theId* and returns it. If no matching object is found, **GetIdObject** returns the constant **nullObject**. Since the id of an object is unique within an IEB, you can use this method to access a specific object. The value returned by this routine is of type **UNIVObject**, and is compatible with any class.

GetNamedObject(theName : String): UNIVObject

GetNamedObject searches in the open web(s) for an object with name *theName* and returns it. Upper and lower case are equivalent. If no matching object is found, **GetNamedObject** returns the constant **nullObject**. *theName* can be any expression which evaluates to a string. The value returned by this routine is of type **UNIVObject**, and is compatible with any class.

GetScreenSize(VAR width, height : Integer)

GetScreenSize returns the width and height in pixels of the main screen (the screen which contains the menu bar)

Insert(source : String; VAR dest : String; index : Integer)

Insert inserts *source* into *dest* at the position indicated by *index*. If the value of *index* is out of range, **Insert** is ignored.

IsAvailable(theCommand : String): Integer

IsAvailable returns **true** if the menu command *theCommand* is enabled, **false** otherwise.

Length(theString : String): Integer

Length returns an **Integer** value that is the current length of *theString*.

Message(theMessage : String)

Message displays a dialog box with the text *theMessage*. The dialog box has an OK button to dismiss it.

NewDirectory(theDirectory : String)

NewSet creates a new directory (folder) named *theDirectory* in the current directory.

NewLink : Link

NewLink returns a new link (named "Link #xx") which belongs to the active web. To give the link another name, use the **SetName** method. Since the new link has no anchors, you must use the **SetAnchors** method to assign it two blocks as endpoints.

NewSet : Set

NewSet returns a new empty set (named “Set #xx”) which belongs to the active web. To give the set another name, use the **SetName** method.

OpenDocuments : Set

OpenDocuments returns a set containing a list of all the open documents. Note that the set returned by **OpenDocuments** is destroyed when the execution of the calling script is terminated, unless you put it into the author or reader web with a call to **AddObject**.

Pos(subString, theString : String): Integer

The **Pos** routine searches for *subString* within *theString* and returns an **Integer** value that is the index of the first character of *subString* within *theString*. If *subString* is not found, **Pos** returns zero. The first character in a string has the index 1.

ReadInteger(theText : String; VAR theValue : Integer; theInitialValue : Integer): Integer

ReadInteger displays a dialog box with the message *theText*, and an editable number field containing the value *theInitialValue*. If the user clicks the OK button, **ReadInteger** returns the value **true** and the variable *theValue* contains the value of the editable number field. If the user clicks the Cancel button, **ReadInteger** returns the value **false** and the variable *theValue* is unchanged.

ReadOk(theText : String): Integer

ReadOk displays a dialog box with the message *theText*. If the user clicks the OK button, **ReadOk** returns the value **true**. If the user clicks the Cancel button, **ReadOk** returns the value **false**.

ReadObject(theText : String; theSet : Set; VAR theObject : Object)

ReadObject displays a dialog box with the message *theText*, and a scroll list containing all the objects in *theSet*.. If the user clicks the OK button, **ReadObject** returns **true** and the variable *theObject* contains the object selected by the user. If the user clicks the Cancel button, **ReadObject** returns the value **false** and the variable *theObject* is unchanged.

ReadString(theText : String; VAR theString : String; theInitialString : String): Integer

ReadString displays a dialog box with the message *theText*, and an editable text field containing the string *theInitialString*. If the user clicks the OK button, **ReadString** returns the value **true** and the variable *theString* contains the value of the editable text field. If the user clicks the Cancel button, **ReadString** returns the value **false** and the variable *theString* is unchanged.

SetDirectory(theDirectory : String)

SetDirectory sets the current directory to value of the string expression *theDirectory*. The current directory is used by several methods and routines (e.g. **NewDirectory**, **Save** and **SaveAs**). When the application is started, it corresponds to the application directory.

You can specify either a partial path from the application directory, or a full path name. For example, if the application directory is 'MyHardDisk:WEBSs Books' you could write either **SetDirectory**('Tutorial') or **SetDirectory**('MyHardDisk:WEBSs Books:Tutorial') to set the current directory to 'MyHardDisk:WEBSs Books:Tutorial'. **SetDirectory**(':') sets the current directory to the application directory.

Wait(nrOfSeconds : Integer)

Wait suspends the execution of the script for *nrOfSeconds* seconds and then continues.

WaitKeyOrMouse

WaitKeyOrMouse suspends the execution of a script until the user clicks the mouse button or presses a key. A dialog box containing the message "Please click or press a key to continue" is displayed.

Callable Methods

Methods that can be called from a script are listed by class. If M is a method of class C, then every member of every subclass of C can call method M.

Class Object

GetId : Integer

GetId returns the id of the object. The id of an object is unique within an IEB.

GetName : String

GetName returns the name of the object.

GetOwningWeb : Set

GetOwningWeb returns the web to which the object belongs (i.e. either the author web or the reader web).

GetType : Integer

GetType returns the type of the object (as an integer value).

SetName(theName : String)

SetName the name of the object to *theName*.

SetType(theType : Integer)

SetType sets the type of the object to *theType*. No control is made to ensure that *theType* is a valid type (i.e. that the value of the expression *theType* corresponds to an existing type).

Class Document

Close

Close closes the document.

Delete

Delete deletes the document from the disk and removes it from the owning web.

DeselectAll

DeselectAll deselects everything in the document.

GetAttachedDocuments : Set

GetAttachedDocument returns a set containing all the documents that can be reached by following the links which have an endpoint in the document. Note that the set returned by GetAttachedDocument is destroyed when the execution of the calling script is terminated, unless you put it into the author or reader web with a call to AddObject.

GetBlocks : Set

GetBlocks returns a set containing all the blocks in the document. Note that the set returned by GetBlocks is destroyed when the execution of the calling script is terminated, unless you put it into the author or reader web with a call to AddObject.

GetSelectedBlock : Block

If the document is open and a block marker is currently selected, GetSelectedBlock returns the corresponding block. Otherwise, it returns the constant nullObject.

GetDocLinks : Set

GetDocLinks returns a set containing all the links that have an end point in the document. Note that the set returned by GetDocLinks is destroyed when the execution of the calling script is terminated, unless you put it into the author or reader web with a call to AddObject.

GetWindowLoc(VAR left, top, right, bottom : Integer);

GetWindowLoc returns the coordinates of the document window in the variables referenced by *left*, *top*, *right* and *bottom*.

IsOpen

IsOpen returns the value true if the document is open, false otherwise.

Open

Open opens the document.

Save(theName : String)

Save saves the document in the current directory (see routine SetDirectory) under the name *theName*. If the document was already saved on disk, Save does nothing (use DoMenu('Save') to save a modified document). An error occurs if a document with the name specified exists in the current directory.

SaveAs(theName : String; inSameDir : Integer):Document

SaveAs saves a copy of the active document under the name *theName*, makes the copy the active document and returns it. If *inSameDir* is true, the copy is saved in the same directory as the original document, otherwise it is saved in the current directory (see routine SetDirectory). An error occurs if a document with the name specified exists in the same resp. current directory.

Select

Select makes the document the active document. In a triggered script, it is recommended that you call this method to update the menus before calling the DoMenu routine to invoke a command related to the document.

SetWindowLoc(left, top, right, bottom : Integer);

SetWindowLoc sets the coordinates of the document window to the values of the parameters *left*, *top*, *right* and *bottom*.

Class TextDocument

FindString(theString : String; matchCase : Integer): Integer

The FindString routine searches for *theString* within the document and returns an Integer value that is the index of the first character of *subString* within the document. If *theString* is not found, FindString returns zero. Upper resp. lower case is considered in the search only if *matchCase* is true. FindString only searches from the current location of the cursor; if you want to search in the whole document, call the DeselectAll method first.

GetSelection(VAR index, length : Integer)

GetSelection returns the index of the first character selected in the document and the number of characters selected.

InsertString(theString : String)

InsertString replaces the current selection in the document with *theString*. If no text is selected, *theString* is inserted at the current insertion point (i.e. where the blinking cursor appears).

SelectString(index, length : Integer)

SelectString select *length* characters in the document, starting at *index*, and scrolls the document so that the selection is visible. The first character in a text document has index 1. To place the cursor at a specific position without selecting any text, specify zero for *length*.

StringAt(index, length : Integer): String

StringAt returns the string of *length* characters in the document starting at the position indicated by *index*.

Class PictDocument

SelectRect(left, top, right, bottom : Integer);

SelectRect selects the rectangle with coordinates *left*, *top*, *right* and *bottom*. and scrolls the document so that the selection is visible (you could then create a block from the selected rectangle with a DoMenu('Create Block') call).

Class BrowserDocument

Activate

Activate makes a browser the active browser, opening it if necessary. If a browser was previously activated, it is deactivated.

AddDocument(theName : String; theDoc : Document)

AddDocument adds *theDoc* to the list of documents associated with the node whose name is *theName*.

CreateNodeDown(theName : String)

CreateNodeDown creates a node in the browser with name *theName*. The node is placed under the last node. CreateNodeDown does not create any connections between the nodes.

CreateNodeRight(theName : String)

CreateNodeRight creates a node in the browser with name *theName*. The node is placed on the right of the last node. CreateNodeRight does not create any connections between the nodes.

Class Block

FollowLink(aLink : Link)

If the document that contains the block is open and the block marker is selected, FollowLink follows the link *aLink*. The block must be an end point of the link.

GetDocument : Document

GetDocument returns the document containing the block.

GetLinks : Set

GetLinks returns a set containing all the links attached to the block. Note that the set returned by GetLinks is destroyed when the execution of the calling script is terminated, unless you put it into the author or reader web with a call to AddObject.

SelectExtent

SelectExtent selects the extent of the block in the document. If the document is not open, SelectExtent does nothing.

SelectMarker

SelectMarker selects the marker of the block in the document. If the document is not open, SelectMarker does nothing.

Class TextBlock

GetExtentString : String

GetExtentString returns the string corresponding to the block extent. GetExtentString will only return the correct string if it is called when the document is open.

Class PictBlock

GetExtentRect(VAR left, top, right, bottom : Integer)

GetExtentRect returns the coordinates of the rectangle corresponding to the block extent in the variables referenced by *left*, *top*, *right* and *bottom*.

GetMarkerRect(VAR left, top, right, bottom : Integer)

GetMarkerRect returns the coordinates of the rectangle corresponding to the block marker in the variables referenced by *left*, *top*, *right* and *bottom*.

SetExtentRect(VAR left, top, right, bottom : Integer)

SetExtentRect sets the coordinates of the rectangle corresponding to the block extent to the values of the parameters *left*, *top*, *right* and *bottom*.

SetMarkerRect(VAR left, top, right, bottom : Integer)

SetMarkerRect sets the coordinates of the rectangle corresponding to the block marker to the values of the parameters *left*, *top*, *right* and *bottom*.

Class Link**GetAnchors(VAR start, end : Block)**

GetAnchors returns the endpoints of the link in the variables referenced by *start* and *end*.

SetAnchors(start, end : Block)

SetAnchors sets the endpoints of the link to the blocks referenced by the variables *start* and *end*.

Class Set**AddObject(theObject : Object)**

AddObject adds *theObject* to the set. If the object already belongs to the set, AddObject does nothing. A reader cannot insert an object into the author web.

GetNrObjects : Integer

GetNrObjects returns the number of objects in the set.

RemoveObject(theObject : Object)

RemoveObject removes *theObject* from the set. If the object does not belong to the set, RemoveObject does nothing. A reader cannot remove an object from the author web.

Class Script**Execute**

Execute launches the execution of the script. It can only be invoked with an object specifier (otherwise, the script compiler could not check the parameters of the script). The actual parameters must match the formal ones in number and type.

GetEnabled : Integer

GetEnabled returns true if the script is enabled, false otherwise. For unbound scripts, this method always returns false .

SetEnabled(state : Integer)

`SetEnabled` enables (if state is `true`) resp. disables (if state is `false`) the bound script, so that it will/will not be automatically executed when the triggering conditions defined in its header are met. This method has no effect on unbound scripts.

Triggering Methods

The methods that can trigger the execution of a script are listed by class. As explained in “Object-Oriented Fundamentals” in Chapter 3 of this guide, a method `M` contained in a class `C` is inherited by all its descendants, i.e. you can specify `M` as triggering method for any subclass of `C`.

Class Document

Close

`Close` is triggered when the document is closed.

Open

`Open` is triggered when the document is opened.

Class Link

Follow

`Follow` is triggered when the link is followed.

Class Block

Arrive

`Arrive` is triggered when a link is followed and the block at the end of the link is selected.

DoubleClick

`DoubleClick` is triggered when the block is double-clicked.

Appendix D: Examples of Scripts

The scripts shown below are contained in the document “Scripts Examples” inside the “Texts” folder of the Tutorial IEB. You can use the Copy/Paste commands from the Edit menu to copy the contents of these scripts into the script browser.

D.1 Assigning a Predefined Window Location to a Document

This script assigns to the current document the same window location as the document GraphTemplate.

```
ON Open OF ANY Document in [GraphDocs];
VAR left,top,right,bottom: Integer;
BEGIN
    [GraphTemplate].GetWindowLoc(left,top,right,bottom);
    current.SetWindowLoc(left,top,right,bottom);
END
```

D.2 Creating a Set of Visited Documents.

This script checks if the set Visited Document exists and puts the current document into it.

```
ON Open OF ANY Document;
VAR aSet : Set;
BEGIN
    aSet := GetNamedObject('Visited Documents');
    IF aSet = nullObject THEN
        BEGIN
            aSet := NewSet;
            aSet.SetName('Visited Documents')
        END;
    aSet.AddObject(current)
END
```

D.3 Suggesting a Partial Ordering on Document Opening.

This script asks the user whether he/she wants to study the document SimpleExample before the current one.

```
ON Open OF Document [ComplexExample];
BEGIN
    IF NOT ([SimpleExample] IN [Visited Documents]) THEN
        IF ReadOk('Study a simpler example first?') THEN
            BEGIN
```

```

        [SimpleExample].Open;
    Abort
END
END;

```

D.4 Selectively Closing Documents.

This script closes all the documents attached to the current document, unless they belong to the set KeepOpenDocs.

```

ON Close OF ANY Document IN [AnchorDocuments];
VAR doc : Document;
BEGIN
    FOR EACH doc IN current.GetAttachedDocuments DO
        IF NOT (doc IN [KeepOpenDocs]) THEN doc.Close;
    END
END

```

D.5 Formatting a Text Document

This script applies several formatting commands to the current selection in a textual document (i.e. to obtain the format of the title above). To use it, select the text to format, choose the Execute Other Script command from the Script menu and execute the script.

```

ON EXECUTE;
BEGIN
    DoMenu('Plain Text');
    DoMenu('Times');
    DoMenu('18 Point');
    DoMenu('Underline');
END

```

D.6 Creating a Simple Browser

This script creates a browser in the same directory as the application, which contains all the graphical documents in the set Visited Documents whose name starts with “Example”.

```

ON Execute;
VAR    doc1:  BrowserDocument;
        doc2:  PictDocument;
BEGIN
    DoMenu('New Browser Document...');

```

doc1 := AsUNIVObject(frontDocument); { frontDocument is of class Document, so we have to make it compatible with the class of doc1, i.e. BrowserDocument. Since we just created a browser document, this should not cause any problems.}

```

FOR EACH doc2 IN [Visited Documents] DO
  IF Pos('Example',doc2.GetName) = 1 THEN
    BEGIN { "Example" was found at the beginning of the document name }
      doc1.CreateNode(doc2.GetName);
      doc1.AddDocument(doc2.GetName,doc2)
    END;
  SetDirectory(':');
  doc1.Save('Example Graphic Documents')
END

```

D.7 Defining a Guided Tour

This script activates a browser and opens the documents associated with its nodes, asking the user after each visited node whether he/she wants to continue. Note that we use a variable to store a reference to the browser, instead of using each time the object specifier [Demo.Table] which would force the system to search repeatedly for the same document within all the objects in the open webs, a time-consuming task.

```

ON Execute;
VAR   theBrowser: BrowserDocument;
      continue:   Integer;
BEGIN
  theBrowser := [Demo.Table];
  theBrowser.Activate;
  theBrowser.DeselectAll;
  continue := true;
  WHILE continue AND IsAvailable('Next Node') DO
    BEGIN
      DoMenu('Next Node');
      continue := ReadOk('Press OK to continue, Cancel to abort.');
```

```

    END
  END
END

```

D.8 Searching and Replacing Strings in a Document

This script will replace all occurrences of the string *s1* in the document *doc* with *s2*.

```

ON Execute(doc: TextDocument;s1,s2: String);
VAR index: Integer;
BEGIN
  doc.Select;           { makes the document the active one }
  doc.DeselectAll;     { puts the cursor at the beginning of the document }

```

```
index := doc.FindString(s1,false);  
WHILE index > 0 DO
```

```

BEGIN
  doc.Selectstring(index,Length(s1));
  doc.Insertstring(s2);
  index := doc.FindString(s1,false)
END
END

```

A possible use of the above script (supposedly named ReplaceAll) would be:

```
[ReplaceAll].Execute([myDocument],'word1','word2')
```

D.9 Creating a Glossary

The following script automatically builds a glossary, by creating links between a series of words and all their occurrences in a set of documents.

For each block of type Glossary in the glossary document *doc*, the script will :

1. Select the block marker and get the string *theString* corresponding to the block extent;
2. For each document *textDoc* in *settheSet*, it will :
 - open the document and put the cursor at the beginning,
 - (2a) search for *theString*,
 - look if there is a block marker on the left of the word,
 - if yes, select the block marker (to avoid creating several blocks for the same word),
if no, create a block and select its marker,
 - issue a Start Link command,
 - select the glossary document *doc*,
 - issue a Complete Link command,
 - give the new link a special name and type (users and other scripts can make use of this),
 - select the document *textDoc* and repeat the above process from (2a) until *theString* cannot be found.
3. Close all documents in *settheSet*.

```

ON Execute(doc: TextDocument;theSet:Set);
VAR   index:      Integer;
      textDoc:    TextDocument;
      tBlock:     TextBlock;
      theString:  String;
BEGIN
  doc.Open;
  doc.Select;

```

```
FOR EACH tBlock OF TYPE Glossary IN doc.GetBlocks DO  
  BEGIN  
    {1} tBlock.SelectMarker;
```

```

theString := tBlock.GetExtentString;
{2} FOR EACH textDoc IN theSet DO
  BEGIN
    textDoc.Open;
    textDoc.Select;
    textDoc.DeselectAll; { Put the cursor at the beginning }
    index := textDoc.FindString(theString,false);
    WHILE index > 0 DO { While the string is found }
      BEGIN { check if there is already a block to avoid creating duplicate blocks }
        IF textDoc.StringAt(index-1,1) = '•' THEN
          textDoc.SelectString(index-1,1)
        ELSE BEGIN
          textDoc.SelectString(index,Length(theString));
          DoMenu('Create Block');
          textDoc.SelectString(index,1);
        END;
        DoMenu('Start Link');
        doc.Select;
        DoMenu('Complete Link'); { Create the link }
        lastLink.SetName('Glossary - ' + textDoc.GetName);
        lastLink.SetType(Glossary);
        textDoc.Select;
        textDoc.SelectString(index+Length(theString),0); { to advance the cursor,
          so that we don't find the same word again and again }
        index := textDoc.FindString(theString,false);
      END;
    END
  END;
{3} FOR EACH textDoc IN theSet DO
  textDoc.Close { to close all the documents in theSet }
END

```

To use this script, create a glossary document, create blocks for the entries in the glossary document, give them the type Glossary (do not forget to create the type Glossary before trying to compile the script). Put all the documents to be searched in a set, and call the above script (supposedly named CreateGlossary) :

```
[CreateGlossary].Execute([Glossary Document],[theSet])
```


Appendix E: Syntax of WEBSs Scripting Language

Identifier ::= letter {letter | digit | underscore}
String ::= " ' " {character} " ' "
ObjectSpecifier ::= "[" character {character} "]"
NumericConstant ::= digit {digit}
ConstantIdentifier ::= Identifier
ClassIdentifier ::= Identifier
TypeIdentifier ::= Identifier
TypeSpec ::= "OF" "TYPE" Identifier
IndividualObject ::= ClassIdentifier [TypeSpec] ObjectSpecifier
SetSpec ::= "ANY" ClassIdentifier [TypeSpec] "IN" ObjectSpecifier
ClassSpec ::= "ANY" ClassIdentifier [TypeSpec]
TriggerObjectSpec ::= IndividualObject | SetSpec | ClassSpec
MethodIdentifier ::= Identifier
TriggerHeader ::= "ON" MethodIdentifier "OF" TriggerObjectSpec
FormalParameterSection ::= ["VAR"] Identifier { "," Identifier } ":" TypeIdentifier
FormalParameterList ::= FormalParameterSection { "," FormalParameterSection }
ExecuteHeader ::= "ON" MethodIdentifier ["(" FormalParameterList ")"] [":" TypeIdentifier]
ScriptHeader ::= TriggerHeader | ExecuteHeader
VariableList ::= Identifier { "," Identifier } ":" TypeIdentifier
VariableDefinition ::= "VAR" VariableList { ";" VariableList } ";"
VariableDefinitionPart ::= { VariableDefinition }
ActualParameterList ::= Expression { "," Expression }
RoutineIdentifier ::= Identifier
RoutineCall ::= RoutineIdentifier ["(" ActualParameterList ")"]
VariableIdentifier ::= Identifier
ObjectAccess ::= ObjectSpecifier | VariableIdentifier | RoutineCall
MethodCall ::= ObjectAccess "." MethodIdentifier ["(" ActualParameterList ")"]
RelOperator ::= "=" | "<>" | "<" | "<=" | ">" | ">=" | "IN"
AddOperator ::= "+" | "-" | "OR"
MulOperator ::= "*" | "/" | "AND"
Expression ::= SimpleExpression [RelOperator SimpleExpression]
SimpleExpression ::= ["-"] Term { AddOperator Term }
Term ::= Factor { MulOperator Factor }
Factor ::= NumericConstant | String | NOT Factor | VariableAccess | RoutineCall |
 MethodCall | "(" Expression ")" | ObjectSpecifier | ConstantIdentifier
Assignment ::= VariableIdentifier ":" Expression
IfStatement ::= "IF" Expression "THEN" Statement ["ELSE" Statement]
RepeatStatement ::= "REPEAT" StatementSequence "UNTIL" Expression
WhileStatement ::= "WHILE" Expression "DO" Statement

ForToStatement ::= "FOR" VariableIdentifier ":" Expression "TO" Expression
 DO Statement
 ForEachStatement ::= "FOR" "EACH" VariableIdentifier ["OF" "TYPE" Factor]
 "IN" Expression DO Statement
 ReturnStatement ::= "RETURN" [Expression]
 Statement ::= [Assignment | IfStatement | RepeatStatement | WhileStatement |
 ForToStatement | ForEachStatement | RoutineCall | MethodCall |
 ReturnStatement | CompoundStatement]
 StatementSequence ::= Statement { ";" Statement }
 CompoundStatement ::= "BEGIN" StatementSequence "END"
 MainBlock ::= [VariableDefinitionPart]
 CompoundStatement
 Script ::= ScriptHeader ";" MainBlock