OPENXTALK USER GUIDE
User Guide v1.1, by LiveCode Ltd, edited and debranded for open source as requested by LiveCode Ltd.

==========

BEGINNING CONCEPTS

CHAPTER 1) INTRODUCTION

Welcome, System Requirements, Documentation, Dictionary, Start Center, Forums.

1.1) Welcome

Thank you for choosing OpenXTalk. Turn your ideas into desktop applications using a graphical user interface designed to be easier and faster than most other coding languages.

Coding beginners will find the environment easy to learn, and experienced programmers will find the environment powerful and productive. The coding language is object-based which makes it easy to write modules of code attached directly to individual objects. Unlike other languages, you run and edit an application live without a compile and debug cycle saving time.

If coming from another language, you'll appreciate that the language is typeless, with data automatically stored in the most efficient format and converted. Memory management is fully automatic. Apps are not interpreted in the traditional sense, so they provide excellent performance. A graphical app will feel more responsive than a Java app, and take less time to write.

Apps are cross-platform. Run an app on Windows, Mac, and Linux. Unlike other cross-platform frameworks, it will look and feel native on each platform. No need to learn how to access individual programming interfaces for each operating system. This saves time and the effort of learning thousands of platform specific interface calls.

The goal of this manual is to provide an accessible, comprehensive, useful guide with the depth to cover advanced features while remaining accessible to beginners. We hope you find this manual useful and enjoy coding.

1.2) System Requirements

Memory and disk requirements below are for the development environment. Apps created will require sufficient system resources to load, display, process and interact with the content. Test your application to determine the minimum requirements. Most apps will run on a moderately powerful computer and require fewer resources than those listed for the development environment.

See Help > Release Notes for the latest system requirements.

1.3) Using The Documentation

This documentation contains examples of coding syntax. A syntax description uses standard conventions:
[ ] Square brackets enclose optional portions.
{ } Curly braces enclose sets of alternatives from which to choose.
| Vertical bars separate different alternatives.
\ Line continuation character – this line continues to the next line.
File > Open means from the "File" menu choose "Open".

Many menu items have keyboard equivalents, accessed by holding down a modifier key and pressing another key. Modifier keys

are usually the controlKey or on Mac the commandKey.

o Windows, and Linux keyboard shortcuts: Control, Alt, Right-click.
o Mac equivalent: Command, Option, Control-click.

1.4) Navigating The Documentation.
Documentation is spread across Dictionary, Start Center, Forums, and this User Guide. Access from the Help menu.

Dictionary, Start Center, and Forums
The Dictionary contains the coding syntax which can be searched or filtered. When the selected filter is "All" the keyword index will contain all the entries in the Dictionary. The documentation for a Dictionary entry can be viewed by clicking on the entry header in the keyword index.

The Start Center contains further documentation and a jump list to recently opened stacks.

The Forums are a good way to get online technical help. https://www.openxtalk.org/forum
==========

CHAPTER 2) GETTING STARTED

Event Driven Programming, Object-Based Programming, Run & Edit Modes, Structuring An Application, Structure Of A Stack, Opening A Stack, MainStack & SubStacks, Stacks & Memory, Media & Resources, External Files.

Creating a simple graphical app can take just minutes. First create a user interface Stack window. Then populate it with objects like push buttons, check boxes, text fields, or menus. Finally, use the English-like coding language to tell the app how to behave including palettes and dialogs.

2.1) Event Driven Programming
An app is driven by user actions. It constantly watches the computer for common actions, such as clicking on a button, typing into a field, sending data across a network, or quitting an application. Whenever an event occurs, it sends a message. You decide what messages you want your program to respond to. Each message is automatically sent to the most relevant object. For example, if a user clicks a button, a message is sent to the button. You place code within the button that tells it how to respond to being clicked.

There are a wide range of possible events. When a user clicks on a button, a series of events are sent to the button. When the mouse first moves within the border of the button a mouseEnter message is sent. Then a series of mouseMove messages are sent as the mouse moves over the button. When the mouse button is depressed a mouseDown message is sent. When the mouse is released a mouseUp message is sent. You don't have to respond to all of these events. Simply place code within an object to handle the events you want your application to respond to.

Events that are not handled by individual objects can be handled in a number of ways at different levels of your application, in libraries, or they can be ignored. The rules that govern what happen to events that are not processed by an object are described in the section The Message Hierarchy.

2.2) Object-Based Programming
Any graphical app will be based on objects. Create the objects of the application before writing any code. Start by drawing the buttons, text fields, and other objects that make up your application. Like other layout, drawing, or application development environments, select objects by clicking them, move by dragging them, resize, and change their layer to move them closer or further from the top of the interface. Once the objects are in place, attach code to each object to respond to the events you want.

The graphical development environment makes it easy to create and edit any kind of user interface. There are objects for all basic operating system elements including buttons, checkboxes, text fields, menus, and graphics. Create and customize your own

objects that look and behave the way you want. If writing a non-graphical application, create objects to assist in organizing your code into sections and load these objects off screen, or write code in a text file and run the text file directly. This method is commonly used to communicate with Apache and other web browsers when building server-side or network applications.

2.3) Run And Edit Modes
Run mode: Left Arrow Browse tool at the top of the tools palette. Edit > Browse Tool.
Edit mode: Right Arrow Pointer tool at the top of the tools palette. Edit > Pointer Tool.

Unlike most other development systems, an app can be created, edited, debugged, and run live. When in run mode, objects receive all the normal messages that drive an app. For example, clicking on a button in run mode will cause a mouseUp message to be sent to it, causing the button's script to run if you've designed it to respond to the mouseUp message. When in edit mode, objects do not receive messages when clicked on so you can move, resize, or edit its properties.

View and edit properties and code in either mode. The app does not stop running while changes are made to it. Only mouse interaction with objects is suspended in edit mode. Easily make simple changes and watch each change take effect as you make it, resulting in a more productive and satisfying development experience.

Tip: To temporarily stop all messages while editing, choose Development > Suppress Messages.

2.4) Structuring An Application
Stacks & Cards & Files
The first step in creating an application is creating a window, which is called a Stack. Each window you see is a stack. Palettes, dialog boxes, and standard windows are all stacks. Each stack contains one or more sets of information called Cards. Each card can have a different appearance or all the cards in a stack can look the same. By going from card to card in a stack, you change what's being displayed in that stack's window.

Think of a stack as a deck of playing cards you can flip through, but only one card is visible at a time. A stack can have a single card or many cards. All user interface objects are created by double-clicking them on the Tools palette or dragging them from the Tools palette to a card.

Objects can be grouped together to operate as a set. For example, a set of navigation buttons that go from card to card in a stack. A groups can appear on more than one card, so navigation buttons or a background image can appear on each card of a stack.

A collection of stacks can be saved together in a single file. This file is known as a Stack File. The first stack in a stack file is called the MainStack and will be loaded automatically when an app is opened.

2.5) Structure Of A Stack File
Each stack file contains one or more stacks: either a single mainstack, or a mainstack and one or more substacks. Since each stack is a window (including editable windows, modeless and modal dialog boxes, and palettes), a single stack file can contain multiple windows. Use this capability to bundle several related stacks into a single file for easy distribution, to organize stacks into categories, or to allow several stacks to inherit properties from the same mainstack.

2.6) Opening A Stack File
To open a stack file, choose File > Open Stack. The stack file's mainstack opens automatically to its first card. A stack file is saved as a whole, all stacks in a stack file are saved at the same time. Substacks in a stack file do not open automatically, they must be opened by code in a message handler or the Message Box.

2.7) MainStacks & SubStacks
The first stack created in a stack file is called the MainStack. Any other stacks created in the same stack file are called SubStacks

of the mainstack. The mainstack is part of the object hierarchy of all other stacks in the same stack file. The mainstack contains its substacks. Events that are not handled by a substack are passed on to the mainstack's script. Color and font properties of substacks are inherited from the mainstack. Dialog boxes and palettes are commonly substacks of the mainstack. Because of message hierarchy, code used by substacks can be stored in the mainstack's script.

2.8) Stacks & Stack Files & Memory
A stack file can be loaded into memory without actually being open. A stack whose window is closed (not just hidden) is not listed in the openStacks function. However, it takes up memory, and its objects are accessible to other stacks. For example, if a closed stack loaded into memory contains an image, the image can still be used as a button icon in another stack. If one stack in a stack file is loaded into memory, so are all other stacks in the same stack file.

A stack can be loaded into memory without being open when:
o Code in another stack reads or sets a property within the closed stack. This automatically loads the referenced stack into memory.
o The stack is in the same stack file as another open stack.
o The stack was opened and then closed, and its destroyStack property is set to false. This prevents removing the stack from memory.

Tip: To manipulate a stack window in an external, use the windowID property.

2.9) Media & Resources
When planning a project, consider the types of media needed and how to access that media. A wide range of media formats are supported. Media can be accessed using built-in support or with an external library. Built-in support can consistently display or play media on all platforms without having to check that any 3rd party component has been installed. Since each loaded stack file takes up as much memory as the size of all its stacks, place large seldom-used objects in external files. Color pictures, sound, and video in external files can be bundled with your app and loaded into memory only when needed.

Built-in media support allows embedding media directly within your stack file, or to reference it externally storing it in a data folder or online.

Embedding media within a mainstack:
o Allows distribution of a single-file app for easy, reliable distribution.
o Requires importing media whenever it is updated.
o Requires enough memory to load all the media.
o Allows using the built-in editing capabilities directly.
o Is less practical for creating large themed or localized applications where one set of media is replaced with another set.

Referencing media externally:
o Requires distributing the media files separately.
o Allows editing media files directly, they update automatically.
o Makes it easy to load and unload media to reduce memory requirements.
o Requires media import and export to use the built-in editing capabilities.
o Makes it easy to create themed or localized apps by linking to a different directory.

Tip: When importing images, use the Image Library, or create a card that contains all the originals, then reference those images throughout the stack.

2.10) How To Use External Files
There are three main ways to use external files:
o Keep media such as images, sounds, and video in external files, in the appropriate format, and use referenced objects to display the contents of the files. When you use a referenced object, the image, sound, or video file is loaded into memory only when a card that contains the referenced control is being displayed. The memory needed for the image, sound, or video is

therefore only used when actually needed. By using relative file paths, the application and its data files can be moved within any system.

o Keep portions of an app in a separate stack file, and refer to the stacks in that stack file as necessary. The stackFiles property simplifies referring to external stack files. When you set a stack's stackFiles property to include one or more file paths, the stacks at those locations become available to your stack by referring to the stacks by name.

o Keep portions of an app on a server, and download them using the built-in URL commands.

Note: To create a referenced object, choose File > New Referenced Control, or create an empty image or player object, then set the object's fileName property to a filepath for the file you want to reference.
==========

CHAPTER 3) INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)
Menu Bar, File, Edit, Tools, Object, Text, Development, View, Window, Help, Project Browser, Properties Inspector, Code Editor, Debugger, Message Box, Find & Replace.

The integrated development environment (IDE) contains features needed to quickly create a professional application. The Project Browser allows easy access to all areas of the app. The Properties Inspector allows setting appearance and behavior. The Code Editor allows adding code to each object in the app. The Message Box provides a mini command-line that allows direct development of the app and to quickly test code and functionality.

3.1) The Menu Bar
File > New Mainstack, New Substack, Open Stack..., Open Recent Stack, Close, Close and Remove From Memory..., Import As Control, New Referenced Control, Save, Save As..., Move Substack to File..., Revert to Saved..., Standalone Application Setup..., Save as Standalone Application..., Page Setup..., Print Card..., Print Field..., Exit.

Edit > Undo, Cut, Copy, Paste, Paste Unformatted, Clear, Duplicate, Select All, Deselect All, Invert Selection, Select Grouped Controls, Intersected Selections, Find and Replace..., Preferences.

Tools > Browse Tool, Pointer Tool, Tools Palette, Paint and Draw Tools, Project Browser, Message Box, Extension Manager, Extension Builder, Menu Builder.

Object > Object Inspector, Card Inspector, Stack Inspector, Object Script, Card Script, Stack Script, Group Selected, Edit Group, Remove Group, Place Group, new Card, Delete Card, New Control, New Widget, Flip, Rotate, Reshape Graphic, Align Selected Controls, Send to Back, Move Backward, Move Forward, Bring to Front.

Text > Plain, Bold, Italic, Underline, Strikeout, Box, 3D Box, Link, Subscript, Superscript, Font, Size, Color, Align.

Development > Object Library, Image Library, Plugins, Test, Test Target, Script Debug Mode, Clear All Breakpoints, Message Watcher, Suppress Errors, Suppress Messages, Suspend Development Tools.

View > Go First, Go Prev, Go Next, Go Last, Go Recent, Toolbar Text, Toolbar Icons, Palettes, Rulers, Grid, Backdrop, Show IDE Stacks In Lists, Show Invisible Objects.

Help > Dictionary, User Guide, Forums, Sample Stacks, Tutorials, Release Notes, About.

3.2) Project Browser
The Project Browser contains a list of all open stacks, the cards in each stack, and the objects on each card. Navigate to any card, open or close a stack, select, open the property Inspector for, or edit the script of any object. Choose Tools > Project Browser.

3.3) Properties Inspector

The Properties Inspector allows you to view and edit the properties for any selected object. Properties control how an object looks and some aspects of an object's behavior. Choose Object > Object Inspector, Card Inspector, Stack Inspector, or by double clicking on a selected object.

3.4) Code Editor
The Code Editor includes features to help make code more understandable. These include code indentation and color coded syntax highlighting, as well as other integrated tools such as a Debugger and syntax Dictionary. Choose Object > Object Script, Card Script, Stack Script.

The code editor will colorize and format scripts automatically. When typing press tabKey to manually format the script. Pressing the Apply button to instantly compile the script. Any syntax errors will appear in the Error Display and the script will not compile. Click next to a line of script to set or remove a breakpoint. A breakpoint specifies a point at which script execution should pause and the debugger be loaded. A compiled script is not saved until the stack that contains it is saved. Choose File > Save, or ctrl-s. It is not necessary to close the code editor to run a script after compiling. None of the message handlers within the script will be available if there is a script error during compilation.

Compile and execution errors will appear in the Error Watcher. Double click the icon next to the error details to highlight the line the error occurred on in the script. The Handler List displays all the functions, commands and handlers which are part of the current script. When a handler name is clicked, the script will jump to that handler. The code editor also has a built-in syntax Dictionary. When this tab is active a summary of the Dictionary entry for the keyword currently typed will be displayed. The full Dictionary entry can be viewed by clicking Launch Documentation or by toggling the Full Document check box at the bottom of the documentation pane.

Code Editor Menubar
File > Compile, Save, Close, Print, Quit.
Edit > Revert, Comment, Uncomment, Quick Find, Find and Replace
, Go, Variable Checking.
Debug > Show Next, Step Into, Step Over, Step Out, Stop, Run, Toggle Breakpoint, Clear All Breakpoints, Add Variable Watch, Variables, Breakpoints, Entry Point..., Script Debug Mode.
Handler > Go to Handler..., Add Default Handler..., Show Default Handlers.
Help > Documentation Pane, Dictionary, Resource Center, Start Center, User Guide, About.

3.5) Debugger
The Debugger helps track bugs and understand code by allowing to pause the script execution or step through line by line. You cause execution to pause by setting breakpoints. When a script runs, execution will pause at a breakpoint to observe or edit variable values, continue execution, or set another breakpoint later in the code.

When in debug mode, the Continue button will start running the code from the current position. The Program will run until the end is reached or another breakpoint is found. When not in debug mode, the continue button can be used to execute a handler. When pressed a dialog will appear asking which handler to call and, if applicable, which parameters to pass. On clicking OK the code editor will call the handler you specified. It will also remember the handler for next time. To change the handler called later, choose Entry Point
from the Debug menu.

o Stop Debugging: halt execution at the current point and stop debugging. You will then be able to edit the script.
o Show Next Statement: return the debugger to the currently executing statement if you have switched tabs or scrolled during debugging.
o Step Into Next Statement: execute the next statement. If the next statement is a command or function, Step Into will jump to that command or function to execute it line by line.

o Step Over Next Statement: execute a command or function call without stepping through it line by line. The code within the handler will run at full speed and execution will pause on the line after the handler call.

o Step Out: exit a command or function that previously stepped into. When selected, the rest of the current handler will run and execution will pause on the line after the handler call. This is useful to avoid stepping through a long command or function line by line when the error is not within that command or function.

Debug Context: path of execution up to the current statement paused on in the debugger. This identifies where you are in the Program and which handlers called the current handler. Change the debug context to view variables from different parts of the Program. Use this feature to find an erroneous command call or a call where wrong parameters were used.

Variable Watcher: examine the value of variables within a program while it is executing. As you step through code these values will be updated. The variable name is shown in the left column and its value adjacent on the right. If a variable's value is too large to be displayed in the pane it will have a magnifying glass next to it. Clicking this will bring up a watch window which allows the variable's value to be changed. Variables can also be edited by double clicking on their value in the variable watcher.

Use breakpoints to specify where to pause execution in the debugger. Set breakpoints by clicking in the gutter, or by right clicking in the Breakpoint Manager at the bottom of the code editor and selecting New Breakpoint. Each breakpoint is associated with a line of code and execution is paused when that line is reached. Alternatively, a condition can be assigned to the breakpoint where execution will only pause if the condition is true when the line is reached.

To see a list of all the breakpoints within a stack, including the script they are in, which line they are on, and whether there is a condition attached, click the Breakpoints Manager at the bottom of the code editor. The check box to the left of each breakpoint can be used to enable or disable the breakpoint. Double click a breakpoint to go to that line in the associated object's script. To edit a breakpoint, including adding a condition or changing the line number, click the pencil icon. This can also be done by right-clicking on the breakpoint, either in the Breakpoints Manager or the code editor's gutter.

The Breakpoint Manager also allows you to set watches. These are associated with variables rather than lines of code and will pause execution when the value of the variable changes. Alternatively, if the watch has a condition attached, execution will only pause if the variable's value is changed and the condition is true. To add a watch, right click in the Breakpoint Manager and choose New Breakpoint.

3.6) Message Box
The Message Box is a command line tool that can run short scripts, test parts of a program, be a convenient output window for debugging, and edit or set properties. The message box is useful when starting out coding. Try out script examples by copy-pasting them into the message box.

There are seven different modes.
o Single Line: execute a single line or a few lines with ";" semicolon between commands, returnKey to run.
o Multiple Lines: execute multiple line scripts, ctrl/cmd-returnKey to run.
o Global Properties: view and edit global properties.
o Global Variables: view and edit global variables.
o Pending Messages: view, cancel and edit pending messages.
o Front Scripts: view and edit frontScripts.
o Back Scripts: view and edit backScripts.

3.7) Find & Replace
The Find And Replace dialog allows to search the entire app, a portion of the app, multiple files in a directory, or stacks specified in the application's stackFiles property. Search for object names, scripts, field and button text, custom properties, and other properties. After performing a search you can replace the search term with a replacement term, either in all of the results or on a selection of the results.

Search options.
o Current selection: search the currently selected objects.
o This card: search the current card of the front most editable stack.
o This stack: search the front-most editable stack.

o This stack file: search the mainStack and all of its subStacks.
o This stack file and its stack files: includes the referenced stacks.

Other Options
Case Sensitive, Regular Expression, Obey dontSearch, Search marked or unmarked cards, Name, Script, Custom, Field Text, Button Text, All Other.
==========

CHAPTER 4) BUILDING A USER INTERFACE

Creating & Organizing Objects, Object Types, Stack, Window, Palette, Dialog Box, Drawer, Decorations, Button, Field, Data Grid, Card, Group & Background, Graphic, Image, Player, Audio Clip, Video Clip, Menu, Other Objects, Menu Builder, Geometry Manager, User Interface Design Tips.

The user interface for an application is often one of its most important features. Building a clear, logical, aesthetically pleasing user interface will make all the difference to the success of your app. This chapter tells how to create and layout objects, which objects to use, how to build custom objects, and tips for good user interface design.

4.1) Creating And Organizing Objects
Creating Objects With The Tools Palette.
The Tools palette allows you to change between Edit and Run mode, create objects, create bitmap images with the paint tools, and create vector graphics with the graphic tools.

When in run mode, objects receive all the normal messages that drive an app. Clicking on a button in run mode will cause a mouseUp message to be sent to it. When in edit mode, objects do not receive messages when clicked on so they can be moved, resized, or their properties edited.

o Double-click an object on the Tools palette, or drag an object from the Tools palette onto a stack to create a new object.
o Use Paint tools to create and edit bitmap graphics. Create an image object and paint within that area, or modify an existing image.
o Use Vector Graphics tools to create smooth scalable vector graphics. There are options for fill color, line color, line thickness, and shapes. Click and drag in a stack to create the new graphic.

Tip: To open a system-standard color chooser, click on a color chooser popup menu at the bottom of the Vector Graphics or Bitmap Graphics sections of the tools palette.

Alignment & Layering
Options in the Position tab of a property inspector.
o Lock size and Position: lock the object so its size and position cannot be adjusted with the mouse when in edit mode. This also prevents images, groups and players from automatically resizing to display their entire content whenever the card that they are on is reopened.
o Width & Height: set the width and height of the object(s) currently being operated on by the Property Inspector. Objects are resized from their center.
o Fit Content: automatically size the object large enough to display its content. In the case of buttons, the content is the text and any icon. For images, this is the native width and height of the original image before any scaling. See formattedWidth and formattedHeight in the Dictionary.
o Location: set the object's position (the center of the object) relative to the top left of the card.
o Left, Top, Right, Bottom: set the position of one of the object's edges.
o Layer: set the layer of the object. The buttons with arrows allow you to send an object to the back, move an object back one layer, bring an object forward one layer and bring an object to the front. Layer determines which objects are displayed in front or behind others, as well as the object's number and tabbing order. Note that you cannot relayer objects that are grouped unless you are in edit background mode, or have the relayerGroupedControls set to true.

Use the Align Controls tab to resize objects relative to other objects, to reposition objects and/or to relayer objects. To open the Align Controls tab, select multiple objects, then open the Property Inspector and choose Align Controls tab from the menu at the

top of the Inspector. The Align Controls tab resizes objects relative to each other. Always select the object to use as a reference first, then select other objects to make the same as that object next. To distribute objects, select them in the order to be distributed.

Options in the Align Controls Tab of a property inspector.
o Equalize: make objects the same width, height, or have exactly the same rectangle.
o Align: align objects by their left, right, top or bottom edges, or by their horizontal or vertical center, using the first object selected as a reference.
o Distribute: distribute objects with an equal difference between them, using the order they were selected. First To Last selected will distribute objects evenly between the edges of the first and last objects selected. Edge To Edge will distribute objects so their edges touch each other exactly. Across Card will distribute objects evenly over the entire card area.
o Nudge: nudge the selected object the number of pixels specified in the center of the arrows. To change the number of pixels, click on the number.
o Relayer: First To Last Selected will relayer objects in the order they were selected. Last To First will relayer objects in reverse order. Use these buttons to set the tab order of a set of objects.

Keyboard Focus
The focus is an indication to the user of which control will receive a keystroke. Exactly which objects are capable of receiving the keyboard focus depend on the current operating system, and the properties applied to the control. Edit fields can receive the focus, as can all other objects on Windows and Linux, and many objects on Mac.

The order in which the user moves through objects that can receive the keyboard focus is determined by the object's layer. When a card is opened, automatic keyboard focus is made to the first object on the card capable of receiving the keyboard focus. Turn on the ability of an object to get keyboard focus by checking the Focus With Keyboard option in the object's Inspector, or by setting its traversalOn property by script.

On some platforms objects give an indication that they have the focus by displaying a border. You can specify whether a border should be displayed by setting an object's Show Focus Border option in the Inspector, or setting its showFocusBorder property by script.

Object Types

4.2) Stacks
For displaying windows, palettes and dialog boxes. Each window is a stack. This includes editable windows, modeless and modal dialog boxes, and palettes, as well as sub-windows available on some operating systems, such as sheets and drawers. Create a new stack by choosing File > New Mainstack. Edit stack properties by choosing Object > Stack Inspector. Caution: Do not start a stack name with "rev". Stacks with names starting with "rev" are reserved by the IDE.

4.3) Window Types And The Mode Of A Stack
The nature of a stack's window depends on the stack's style property and on what command was used to open the stack. Either specify the window type when opening the stack, or allow the stack's style property to determine the type of window it is displayed in.

Stack windows have four types: editable topLevel windows, modeless dialog boxes, modal dialog boxes, or palette windows. You will normally create a new stack and edit it while it is in editable topLevel mode. To create a dialog, first create the stack like any other stack, then when opening it specify in a script that it should be displayed as a dialog or a palette. The commands for doing this are detailed with each type of window below. Test out these commands as you work on the window layout and scripts, using the Message Box or the window context menu.

Most stack windows are editable topLevel windows, the default mode for stacks. When opening a stack using the go command or File > Open Stack menu item, the stack is displayed as an editable topLevel window unless its style property specifies another window type.

4.4) Editable TopLevel Windows - for documents.
An editable window has the appearance and behavior of a standard document window. It can be interleaved with other windows, and you can use any of the tools to create, select, move, or delete objects in the stack.

To display a stack in an editable window, you use the topLevel or go commands:
topLevel stack "My Stack"
go stack "My Stack" --topLevel is the default mode
go stack "My Stack" as topLevel

Stacks whose style property is set to "topLevel" always open as editable windows, regardless of what command you use to open them. If the stack's cantModify property is set to true, the window retains its standard appearance, but tools other than the Browse tool can no longer be used in it. Every tool behaves like the Browse tool when clicking in an unmodifiable stack's window.

4.5) Modeless Dialog Boxes - for alerts and settings.
Modeless dialog boxes are similar to editable windows. Like editable windows, they can be interleaved with other windows in the application. Their appearance may differ slightly from the appearance of editable windows, depending on the platform. Like unmodifiable stacks, modeless dialog boxes do not allow use of tools other than the Browse tool. Use modeless dialog boxes in your application for windows such as a Preferences or Find dialog box, that do not require the user to dismiss them before doing another task.

To display a stack in a modeless dialog box, you use the modeless or go commands:
modeless stack "My Stack"
go stack "My Stack" as modeless

Stacks whose style property is set to "modeless" always open as modeless dialog boxes, regardless of what command you use to open them.

4.6) Modal Dialog Boxes - for dialogs.
A modal dialog box is a window that blocks other actions while the window is displayed. You cannot bring another window in the application to the front until the dialog box is closed, nor can you edit the stack using the tools in the Tools palette. While a modal dialog box is being displayed, the handler that displayed it pauses until the dialog box is closed.

Modal dialog boxes do not have a close box. Usually, they contain one or more buttons that close the window when clicked. If you mistakenly open a modal dialog box without having included a button to close the window, use a keyboard shortcut to display a context menu:
(Ctrl-Shift-RightClick for Linux/Windows, Cmd-Ctrl-ShiftClick for Mac. Choose toplevel to make the stack editable).

To display a stack in a modal dialog box, use the modal or go commands:
modal stack "My Stack"
go stack "My Stack" as modal

Stacks whose style property is set to "modal" always open as modal dialog boxes, regardless of what command you use to open them.

4.7) Palettes - for accessory and tool windows.
A palette has a slightly different appearance, with a narrower title bar than an editable window. Like dialog box windows, a palette does not allow use of tools other than the Browse tool. A palette floats in front of any editable or modeless windows that are in the same application. Even when you bring another window to the front, it does not cover any palettes.

On some Linux systems, palettes have the same appearance and behavior as ordinary windows and do not float. On Mac systems, palette windows disappear when their application is in the background and another application is in front.

To display a stack in a palette, you use the palette or go commands:
palette stack "My Stack"
go stack "My Stack" as palette

Stacks whose style property is set to "palette" always open as palettes, regardless of what command you use to open them.

4.8) Built-In Dialogs
Ask Question Dialog - for asking a question.
The ask question dialog is a special type of window that is designed to make it easy to ask the user a question. It includes special syntax for opening the dialog with question text and returning any answer to the script that called it. You can also specify the window title and an icon to be displayed in the window. The font, object positions, button order, and icon will automatically change to reflect the operating system. If more flexibility is needed, create your own modal dialog box instead.

To display the ask dialog, use the following commands:
ask "What is your name?"
ask question "What is the answer?" titled "Quiz" --specify icon: question, information, error, warning

The result is returned in the special variable it.
if it = "Joe" then DoSomeThing

Answer Alert Dialog - for displaying a dialog.
Like the ask dialog, the answer dialog is a special dialog that has been designed to make it easy to display information in a dialog on screen and optionally allow the user to make a choice from a list of up to seven choices. The answer command opens the dialog, lets you specify the text and the button choices, and returns the button clicked to the script that called it. You can also specify the window title and an icon to be displayed in the window. As with the ask dialog, the font, object positions, button order and icon will automatically change to reflect the operating system. If more flexibility is needed, create your own modal dialog box instead.

To display the answer dialog, use the following commands:
answer information "Hello World!" with "OK" --specify icon: question, information, error, warning
answer question "What city is the capital of Italy?" with "Paris" or "London" or "Rome" titled "Multiple Choice"

The result is returned in the special variable it.
if it is "Rome" then answer information "That was the correct answer." with "OK"

Tip: If not sure what a stack's name is, use the mouseStack function to find out. In Message Box: put the mouseStack, then place mouse over the stack window and press returnKey.

File Selector Dialogs - display system dialogs.
The file selector dialogs display the system standard dialogs. These dialogs allow the user to select a file or a set of files, select a directory, or specify a name and location to save a file. The syntax for bringing up the file selector dialogs mirrors the syntax for the alert and question dialogs. However, unlike these dialogs, the file selector dialogs are displayed using the system standard dialogs where available. The notable exception is Linux platform, where a built-in dialog is used instead due to more limited OS support.

Answer File Dialog - for selecting a file.
answer file "Please select a file:"
answer file "Select an image file:" with type "All Files"

The file path to the file selected by the user is returned in the special variable it. If the user canceled the dialog, the special variable it will be empty and "cancel" will be returned by the result function.

Ask File Dialog - for saving a file.
ask file "Save this document as:" with "Untitled.txt"

ask file "Export picture as:" with type "JPEG File|jpg|JPEG"

The file path to the file saved by the user is returned in the special variable it. If the user canceled the dialog, the special variable it will be empty and "cancel" will be returned by the result function.

Answer Folder Dialog - for selecting a folder.
answer folder "Please choose a folder:"
answer folder "Please choose a folder:" with "/root/default folder"

The file path to the folder selected by the user is returned in the special variable it. If the user canceled the dialog, the it variable will be empty and "cancel" information will be returned by the result function.

Answer Color Dialog - for choosing a color from the operating system's standard color picker dialog.
answer color
The color chosen is returned in the special variable it. If the user canceled the dialog, it will be empty and "cancel" will be returned by the result function.

Answer Printer Dialog - for the standard printer dialog.
answer printer
Use the answer printer command to display a standard printer dialog prior to printing. If the user cancels the dialog, "cancel" will be returned by the result function.

Alpha Blend Windows - for Enhanced Tooltips and Multimedia.
Use the Shape option in the Stack Inspector to set a stack's windowShape property to the transparent, or alpha channel of an image that has been imported together with its alpha channel (i.e. in either PNG or GIF format). This creates a window with "holes" or a window with variable translucency. Apply a shape to any type of stack, regardless of the mode it is opened, allowing such a window to block as a dialog, float as a palette, etc. The border and title bar of a stack are not shown if the stack's windowShape is set. This means you will need to provide methods of dragging and closing the stack window if you want the user to be able to do these tasks.

Change the windowShape property dynamically by script to a series of images to create an animated translucent window.

System Palettes - for utilities floating above applications.
A system palette is like a palette, except that it floats in front of all windows on the screen, not just the windows in its application. Use system palettes for utilities you want the user to see and access in every application. To display a stack as a system palette, turn on the check box "Float Above Everything" in the Stack Inspector. Using this feature overrides the stack's style or mode. The system palette style is currently not supported on Linux. See systemWindow in the Dictionary.

Sheet Dialog Boxes - Mac OS X only.
A sheet is like a modal dialog box, except that it is associated with a single window, rather than the entire application. A sheet appears within its parent window, sliding from underneath the title bar. While the sheet is displayed, it blocks other actions in its parent window, but not in other windows in the application. To display a stack in a sheet dialog box, use the sheet command:
sheet "My Stack" --appears in defaultStack
sheet "My Stack" in stack "My Document"

The answer, answer file, answer folder, ask, ask file, and ask folder commands all include an ...as sheet form to display these dialog boxes as sheets on Mac OS X. You can safely use the 'as sheet' form on cross-platform applications. On systems other than Mac OS X, the sheet command displays the stack as an
ordinary modal dialog box.

4.9) Drawers - Mac OS X only.
A drawer is a subwindow that slides out from underneath one edge of a window, and slides back in when you close it. You usually use a button in the main window to open and close a drawer. To display a stack as a drawer, you use the drawer command:
drawer "My Stack" at left --of defaultStack

drawer "My Stack" at bottom of stack "My Document"

On systems other than Mac OS X, the drawer command displays the stack as an editable window. Because this does not map well to other platforms, only use drawers for applications only being developed for Mac OS X. Use drawers to hold settings, lists of favorites, and similar objects that are accessed frequently but don't need to be constantly visible.

Stack Menus - for displaying non-standard menus.
Usually a menu is implemented as a button, as these will automatically be drawn with the native theme on each platform. It is also possible to display a stack as a pulldown, popup, or option menu. Stack menus are used when a menu needs to contain something other than just text. For example, a popup menu containing a slider, or an option menu consisting of icons instead of text, can be implemented as a stack menu.

To display a stack menu, create a button and set its menuName property to the stack's name. When the user clicks the button, the stack is displayed with the behavior of a menu. Internally, the menu is implemented as a window, and can use the popup, pulldown, or option command to display any stack as one of these menu types.

4.10) Stack Decorations - for window appearance.
Stack decorations specify how the title bar and border of a stack window will be drawn. Access stack decorations options in Object > Stack Inspector. Apart from the differences in appearance between different window modes, the appearance and functionality of a window can vary depending on the stack's properties and the platform it is being displayed on. A window's title bar displays the window's title, a close box, minimize or collapse box, and maximize or zoom box. These properties can also be set by script. See decorations in the Dictionary.

On Mac OS X, a stack's shadow property controls whether the stack window has a drop shadow. Set this property to false to create a window with no shadow.

While the stack's mode is separate from its decorations, the mode may affect whether these properties have an effect. If the decorations property is set to default, it displays the appropriate decorations for the current window type on the current platform.

4.11) Button Objects - for performing actions.
A button is a clickable object that is typically for allowing a user to perform an action by clicking.

Check boxes and radio buttons are used to allow the user to make choices. Radio buttons are used when only one option for a set of options may be selected at any time. Check boxes are used where some options may be turned on and others may be off. The rule of highlighting one radio button at a time are automatically enforced if the radio buttons are grouped.

All button objects are highly flexible and customizable. Common settings include the ability to show and hide the border or fill, and to display an icon. Icons provide a wide range of functionality. For example, create a roll over effect by setting a hover icon, or create a custom check box by setting an icon and a highlight icon. Doing so will replace the system standard check box and display the custom icon for each state.

Button icons are not limited in width or height. They can be animated by using an animated GIF. In fact, an icon can reference any image contained within a stack file. Referencing an image saves disk space and allows updating all icons in a stack by updating a single image.

4.12) Text Field Objects - for displaying or entering text.
Fields display text. Fields can optionally allow the user to edit the text. Fields support multiple fonts, styles, colors, images, and a subset of basic HTML tags. Fields can directly accessing a database with the database library. They can display and render XML using the XML library. List fields allow the user to select one or a set of choices. Table fields allow display of data similar to a spreadsheet. Other types of field can be created including tree views, or any hybrid between these types with a little scripting.

List fields display a set of choices. Users cannot edit list fields. Specify whether the user is allowed to make a single selection or multiple selections. Table fields display data in cells, and optionally allow the user to edit the cells. Table fields are ideal for displaying basic tabular data. For more complex data display use the Data Grid object.

4.13) Data Grid Object - for presenting complex data.
Data grids display data in either the default Table or Form styles. Customize a data grid to include any other object. Data grids can provide a view into source data and display large data sets.

What Is A Data Grid?
A data grid integrates tables and lists (forms) in a stack. They combine nested groups and button behaviors to provide a flexible stylish way to display data. Data Grids are referred to as group or grp.
```
set the dgProp["column labels"] of grp "DataGrid 1" to "State" &cr& "Code"  --set data grid table column labels
put the dgText["true"] of group "DataGrid 1" into field "Text"  --data grid table text including headers
put the dgText of group "DataGrid 1" into field "Text"  --data grid table text without headers or data grid form text.
```

When the first data grid is dragged from the Tools palette to a stack, a substack is also created with two cards; card id 1002 and a second card containing a "row template" group for layout, and a "row behavior" script (if form) for controlling that data grid.

A new second data grid dragged from the Tools palette creates another new card on the same substack, the new card also containing a row template group, and row behavior script (if form) for the second data grid. There is a unique substack card for each data grid table and form in the main stack. Even though data grid tables don't require substack cards, one is created anyway in case controls like buttons or images are used or it's converted to a data grid form.

Data Grid Table
The default data grid table is an object similar to a table field that displays tabbed data in a stylish table complete with headers, sorting, and column alignment. A column can contain text by default, and graphics, images, objects with a customized column template and column behavior.

```
put the dgText["true"] of grp "DataGrid 1" into tData  --true/false, true includes headers in line 1
set the itemDel to tab  --tab delimited data
--make changes to tData here
set the dgText["true"] of grp "DataGrid 1" to tData
send "RefreshList" to grp "DataGrid 1"
```

Data Grid Form
The data grid form is a complex object similar to a list field that displays records the user can select. The data grid form uses a group as a template for each record. A record can contain text by default, and graphics, images, objects with a customized row template and row behavior. Data grid forms do not have headers. A data grid form is very fast because it only draws the records currently visible on screen, so large record sets can be used. Nearly any list can benefit from being displayed in a data grid form.

o A default data grid form requires non-tabbed line list data.
o A data grid form customized in its row template and row behavior will accept tabbed line list data.

How To Create A Data Grid Table
Data grid tables are used to display rows of data in structured columns.
1. Drag a data grid from the Tools palette to a stack.
2. Select it, Object > Object Inspector (or double-click it), and give it a name.
3. In the Columns pane, add column headers using the "+" button, renaming Col 1 to "State", and Col 2 to "Code".
4. In the Contents pane, paste or enter a tab delimited line list of text. Then ctrl-s to save.
```
Alabama	AL
Alaska	AK
Arizona	AZ
Arkansas	AR
```

The tab delimited text entered in the field will appear as columns in the data grid table. The Inspector automatically creates a data grid column for each text column entered. Clicking a triangle in the header row sorts that column. Column widths can be resized by dragging the column dividers in the header, and the entire data grid can then be resized in edit mode.

Populate Data Grid Using dgText Property
Default data grids can also be populated by code with the dgText property. For example:

```
on mouseUp
  put "State" &tab& "Code" &cr& \
  "Alabama" &tab& "AL" &cr& \
  "Alaska" &tab& "AK" &cr& \
  "Arizona" &tab& "AZ" &cr& \
  "Arkansas" &tab& "AR" into tText
  set the dgText["true"] of group "MyDataGrid" to tText    --true/false, true includes headers in line 1
end mouseUp

set the dgText["true"] of grp "DataGrid 1" to tData  --insert data in default data grids
put the dgText["true"] of grp "DataGrid 1" into tData  --extract data from any data grid
```

To insert data into a customized data grid, the dgData and an array must be used.

Populate Data Grid Using dgData Property
Data grids can also be populated by code using array data with the dgData property. This is more complex. For example:

```
on mouseup
  put "Alabama" into tData[1]["State"]
  put "AL" into tData[1]["Code"]
  put "Alaska" into tData[2]["State"]
  put "AK" into tData[2]["Code"]
  put "Arizona" into tData[3]["State"]
  put "AZ" into tData[3]["Code"]
  put "Arkansas" into tData[4]["State"]
  put "AR" into tData[4]["Code"]
  set the dgData of group "MyDataGrid" to tData
end mouseUp
```

How To Create A Data Grid Form
Data grid forms are complex but less rigid than columns in a table. They offer control over the display of each record. Data grid forms don't have columns or column headers.
1. Drag a data grid from the Tools palette to a stack.
2. Select it, Object > Object Inspector (or double-click it), give it a name, and change the style to Form.
3. In the Contents pane, paste or enter a line list of text (no tabs for default form). Then ctrl-s to save.

```
Alabama-AL
Alaska-AK
Arizona-AZ
Arkansas-AR
```

To customize a data grid form, the "Row Template" layout and "Row Behavior" button script must be edited. Fortunately, some of the layout and much of the script is pre-written requiring some custom modification. This is the complex portion of a data grid form and not for beginning coders. Refer to the Data Grid Manual for detailed instructions.

Row Template Layout
Once programmed, the data grid form must be populated with data. A customized data grid form will accepts tab-delimited text items in return-delimited lines in the contents section of its properties Inspector, or populates by script with either that data or data converted to an array. Customize the row template layout with a checkbox button or additional field added to the "gray"

group at the top left. Each object added corresponds to a data item.

Row Behavior Button Script
Once populated, the data grid form may respond to user interaction with a mouseUp handler in the row behavior script.

```
on mouseUp pMouseBtn  --user click
  local tData
  if pMouseBtn = "1" then  --left click
    put the dgDataOfLine[the dgHilitedLines of me] of me into tData  --array
    set the dgHilitedLines of me to empty  --remove selection
    answer tData["Label1"] with "OK" titled "DataGrid Line Text"  --text of row clicked, default field "Label" in array
tData["Label1"]
  end if
  pass mouseUp
end mouseUp
```

Customize A Data Grid Form
Enable word wrap for a data grid form with four steps.
1. In the data grid form's Inspector, uncheck Fixed Control Height.
2. In "Row Template" layout, in msg box: set the dontWrap of fld "Label" to "false" --allow wrap
3. In "Row Behavior" script, in the LayoutControl handler, in the empty line above "set the rect of graphic" add two lines:

```
put (the top of fld "Label" of me + the formattedHeight of fld "Label" of me -5) into item 4 of theFieldRect  --new bottom
set the rect of fld "Label" of me to theFieldRect  --resize
```

4. Click Apply, save with ctrl/cmd-S. In msg box:

```
set the dgText of grp "DataGrid 1" to fld "MyLineList"  --populate data grid form
```

Data Grid Tips
o Don't paste a data grid. Instead drag a new data grid from the Tools palette and paste in the template and script from the original. An alternative is pasting the card containing the data grid. Pasting a data grid on the same stack or a different stack doesn't create a new card in the data grid substack, the original's substack card is used instead.

o If the mainStack script has preOpenStack, openStack, preOpenCard, or openCard handlers, use "if the owner of this stack = empty then" to stop a data grid substack from triggering commands meant for the mainStack. An alternative is to place the "open" handlers in the first card's script.

o RefreshList and ResetList are data grid only commands used following changes. A data grid looks like a field but is a group.

```
send "RefreshList" to group "DataGrid 1"  --following changes to data grid's data.
send "ResetList" to group "DataGrid 1"  --following changes to data grid's row template group objects.
```

o The LayoutControl handler in the row behavior script is a common source of errors. It uses item 1,2,3,4 of the rect property to align the row template controls. That's Left,Top,Right,Bottom of the control rectangle.

```
set the left of fld "Label" of me...  --re-position only
set the rect of fld "Label" of me...  --resize

on LayoutControl pControlRect  --rect=L,T,R,B
  local theFieldRect
  put the rect of fld "Label" of me into theFieldRect
  put item 1 of pControlRect into item 1 of theFieldRect  --left
  put item 3 of pControlRect  into item 3 of theFieldRect  --right
  set the rect of fld "Label" of me to theFieldRect
  set the rect of graphic "Background" of me to pControlRect  --resize
end LayoutControl
```

o Put a mouseUp handler in the row behavior script for a data grid to respond to a user click.

o The order of items in the dgText[ ] property is determined by the alpha order a-z of the data grid keys/columnHeaders, not by the row template or the row behavior script.

o Change the contents of a data grid with the dgText[ ] property. It has tab-delimited items and return-delimited lines. dgText["true"] includes the data grid keys/columnHeaders in line 1. The dgText and the dgText["false"] do not include keys/headers.

o Change the properties of a data grid with the dgProp["xx"] custom property. One useful property to change from default true to false is "scroll selections into view". In msg box:
set the dgProp["scroll selections into view"] of grp "DataGrid 1" to "false" --stop scroll jump

o When deleting a data grid, a dialog asks to leave or delete its data grid substack card "record template". Default "No" leaves the orphan card in the data grid substack. "Yes" deletes the card, and the substack too if it's the only card. Choose "Yes" option.

4.14) Cards
Each stack contains one or more separate screens of objects. Each screen is known as a card. Each card can have an entirely different appearance, or all the cards in a stack can share some or all elements, by using shared groups, known as backgrounds. Choose Object > New Card to create a new card within the current stack. The new card will either be completely blank, or will contain any shared groups from the previous card.

4.15) Groups & Backgrounds - for organizing and sharing objects.
Groups are used for several purposes: radio button clusters, menu bars, creating scrollable areas within cards, and as backgrounds for displaying sets of objects that are shared between cards. Groups can also be used for creating a simple card and stack database by holding fields that contain a different record on each card.

A group is a single object that holds a set of objects. Objects are grouped by selecting them, then using the group command or choosing Object > Group Selected. Once grouped, it becomes an object itself. Select, copy, move, and resize the group, and all the objects in the group come with it. The objects in the group still maintain their own identities. Add objects to the group or delete them, but the objects are owned by the group instead of the card. A group has its own properties and its own script. Groups can be any size, can be shown or hidden, and can be moved to any location in the stack window, just like any other object. Groups can be layered in any order with the other objects on the card. Groups can also display a border and name.

Unlike other objects, groups can appear on more than one card. Place a group on a card using the place command or Object > Place Group. A group shared among cards appears at the same location on each card. A change made to the position of a shared group on one card is reflected on all the other cards that share the group.

Group vs Background
The terms group and background are interchangeable in some circumstances yet mean different things in others. In general, the term group refers to groups that are placed on a card, while the term background refers to all the groups in a stack that are available for use as backgrounds The expression the number of groups evaluates to the number of groups on the current card. The expression the number of backgrounds evaluates to the number of background groups in the current stack, including groups that are not placed on the current card.

Tip: When referring to a group by number, if you use the word group, the number is interpreted as referring to the groups on the referenced card, in order by layer. If you use the word background, the number is interpreted as referring to the groups in the stack, in the order of their creation.
put the name of group 1 --name of the lowest-layered group on the current card
put the name of background 1 --name of the first group that was created in the stack

The term background can be also used to refer to the set of cards that share a particular group.
go card 3 of background "Navigation" --goes to third card with group "Navigation" on it.

Nested Groups

Since a group is an object, it can be contained in another group. To create a nested group, select the objects to group including the existing group, then choose Object > Group Selected. The existing group becomes a member of the new group.

Selecting And Editing Groups
To select a group, click on one of the objects contained within it. To select an object within the group instead of the group itself, Edit > Select Grouped Controls, or turn on the Select Grouped option on the toolbar. This causes groups to be ignored when selecting objects.
allowing you to select objects inside a group as if the group didn't exist.

Alternatively go into edit group mode, a special mode that only displays the objects within that group. Select the group, then choose Object > Edit Group, or press Edit Group on the toolbar. When finished, choose Object > Stop Editing Group. Toggle this mode by script with the start editing and stop editing commands.

Tip: If a group's border has been set, an outline appears at the group's edges. However, clicking within or on the border does not select the group. To select the group, click one of its objects.

Placing And Removing Backgrounds
Display a group on any or all cards in the stack. First, ensure the group's Behave As Background option has been set in its Inspector, then navigate to a card and choose Object > Place Group. Choose the group to place on the current card. Control these features by script with the backgroundBehavior property and place command.

When creating a new card, any groups on the current card whose Behave As Background has been set are automatically placed on the new card. To make it easy for all the cards in a stack to share a single group, create the group on the first card and set this property to true before creating more cards.

To remove a group from the current card without deleting it from the stack, select the group and choose Object > Remove Group. The group disappears from the current card, but it's still placed on any other cards that share the group. Remove a group from a card by script with the remove command. Completely delete a group in the same way as you delete any other object, by selecting the group and choose Edit > Clear, or press the backspaceKey. Deleting a background group removes it from all the cards it appears on, and from the stack itself.

Tip: Use the start editing command from the Message Box to edit a group that has not been placed on any card. Since the group is not on any card, you must refer to it using the term background instead of the term group.

To dissolve a group back into its component objects, select the group and choose Object > Ungroup. Ungroup by script using the ungroup command. Ungrouping deletes the group object and its properties (including its script) from the stack, but does not delete the objects in it. Instead, they become card objects of the current card. The objects disappear from all other cards the group was on.

Tip: If you ungroup, then select the objects and regroup them before leaving the current card, the group is restored as it was. However, leaving the current card makes the ungrouping permanent and deletes the group from all other cards it was on.

4.15) Graphics, Images, Players, Audio & Video Clip Objects - for multimedia.
A wide range of media formats are supported to produce rich media apps. The image object imports or references images, manipulates images by script or interactively with the paint tools, and saves them out in different formats with variable compression options. Support extends to alpha channeled PNG images and animated GIF images. Images can be imported and reused within a stack to create custom interactive interface elements.

Image formats supported include GIF, JPEG, PNG, BMP, XWD, XBM, XPM, PBM, PGM, and PPM files. On Mac OS Classic systems, PICT files can also be imported (but cannot be displayed on Linux or Windows systems).

Import images using File > Import As Control > Image File. Reference an image using File > New Referenced Control > Image File. Paint tools can only be used on images that have been imported.

Graphic, Image, And Player Objects
Vector graphics can be created and manipulated with the graphic tools and by script. Supported are variable fills, gradients, blended, and anti-aliased graphics. Use graphic objects to create interactive interfaces, graphs, charts or games. Use the player object to display and interact with any media formats supported by AVFoundation on Mac OS X and DirectShow on Windows. Turn on and off tracks within a movie, pan, zoom, or change location within a movie, set callback messages that trigger scripts at specific points in the movie, and stream movies from a server.

Audio Clip & Video Clip
The Audio Clip and Video Clip objects embed audio or video clip data within a stack. Some audio clip formats can be played back directly. They do not have any visual representation and can be accessed by script or in the Project Browser.

4.16) Menu Objects - for displaying choices.
Menus are used to display a list of choices. The pulldown menu displays a standard pulldown menu, and can be automatically inserted into the main menu bar on Mac OS X systems. The option menu allows a choice from a list. The combobox allows the user to type an option or choose from a list. Popup menus can be displayed underneath the cursor and used to provide context sensitive options anywhere in your application. The tabbed panel displays different groups of objects.

Menu contents can be defined using a list of item names and special characters to indicate where shortcuts and check-marks should be placed. This is the most common type of menu and is known as a contents menu. They are automatically drawn using the native system look on each platform. Choosing an item from a contents menu will send a menuPick message along with the name of the item chosen. Sub-menu items can be created in list-based content menus.

Alternatively, menus may be constructed from a stack panel, giving complete control over the menu contents and allowing the display of any object type. When choosing an item from a stack panel menu, the individual object within the menu will receive a mouseUp message. Note that panel menus cannot be displayed within the main menu bar on Mac OS X systems. The cascade menu is a special type of control that is only used when building a stack panel menu with sub-menu items.

The tabbed panel is a type of menu with a list of tabs to be displayed, and receive a menuPick message when the user clicks on a tab in the same way as other menus. There are two common techniques for implementing a tabbed interface: group the objects for each tab together and show or hide the appropriate group when the tab is changed, or place the tab object into a group which is then placed as a background on multiple cards.

4.17) Other Objects
A scrollbar can be of style scrollbar (scrolling bar), progress bar (display a value), slider (choose a value), or little arrows (scroll objects). They are all referred to as scrollbar or sb. Fields and groups can display a built-in scrollbar. Sliders and scrollbars can be displayed horizontal or vertical depending on how they are resized.

```
on mouseUp  --or scrollbarDrag pNew
  set the thumbPosition of me to the thumbPosition of me  --snap to integer
  answer "You chose level" && the thumbPosition of me &"." with "OK" titled "Scrollbar"
end mouseUp
```

Valid abbreviations for objects: btn/button, fld/field, grp/group, grc/graphic, img/image, ac/audioClip, vc/videoClip, sb/scrollbar, msg/msg box/message box.

4.18) Using The Menu Builder
The Menu Builder can create and edit a standard menu bar that will work correctly regardless of the platform. On Windows and Linux, menus built with the Menu Builder will appear in the top of the window. On Mac they will be displayed in the main menu bar. It is also possible to generate a menu bar by script.

Tip: Set As Menu Bar On Mac OS X option previews the menu bar in the main menu bar. To bring back the IDE menu bar when this option turned on, click on an IDE window such as the Tool Bar.

Menu Bar Settings
This area specifies the main settings for the menu bar. Use the New button to create a new menu bar in the current top most editable stack. Enter the name for the menu bar in the text area. Delete will permanently delete the menu bar. Use the Edit button to load an existing menu bar from the top most editable stack to edit in the area below.

Menu Edit Area
Select a menu to work on from the scrolling list. At a minimum the application should have a File, Edit, and Help menu. These menus are created automatically when you create a new menu bar. To create a new menu, move the orange divider bar to the position in the menu bar you want to create the new menu, and press New Menu. Disable the currently selected menu by checking Disabled. Choose the keyboard shortcut (the portion of the name that is underlined), using the Mnemonic popup menu (Windows & Linux only). To move a menu in the list, select it then press the up or down arrows (to the right of the name area).

Menu Content
o Select a menu item to work on from the scrolling list.
o To create a new menu item, move the orange divider bar to the position in the menu bar you want to create the new menu, and press New Item.
o Disable the currently selected item checking Disabled.
o Choose the keyboard shortcut (the portion of the name that is underlined, for use when the menu is open), using the Mnemonic popup menu (Windows & Linux only).
o To move a menu item up or down the list, select it then click the up or down arrows (to the right of the name area).
o To move items into a submenu, click the Right arrow, or click the Left arrow to move a submenu item back into the main menu bar.
o To insert a divider, position the orange divider bar where you want the divider, then click the blue divider button (top right).
o To make the menu item a Checkbox or Diamond option, choose the appropriate option from the Mark popup menu.
o To create a control key shortcut for the item, click the Shortcut check box and enter the letter you want to use for the shortcut.
o To understand the symbols that are created next to the menu items, see the chapter on Programming A User Interface.

Scripting
Auto Script places a basic script within the currently selected button with spaces to insert actions for each of the menu items within that item. Edit Script opens the Code Editor for the currently selected menu to complete final coding.

4.19) Using The Geometry Manager
Use the Geometry Manager to specify how objects should be scaled and positioned when the user resizes the window.

Scale Or Position Selector
Choose whether the control should to be scaled or positioned when the stack is resized. Scaling will change the dimensions of the control as the stack resizes. Positioning will move the control without changing its dimensions. It is possible to scale an object in the horizontal plane and have it position in the vertical by selecting Scale and set the options for one axis in the Linking area then select Position and set the options for the other axis. If options in both the Scale and Position modes are set for both axis, the Scale options will be ignored.

Linking Area
Use the linking area to specify the relationship between the control and the stack window or other objects as the stack is resized. In Scale mode link each edge of the object to the window or another object. In Position mode, link only the X and Y axis of an object. Click the gray bars to create a link. A single click results in an Absolute link, a second click will create a wavy Relative link. An Absolute link keeps the object the current number of pixels away from what it is being linked to. A Relative link calculates the distance as a percentage of the total width of the card. The object will remain the same percentage away from the edge of the card, the exact number of pixels will vary.

When linking to another control, be sure to link to a control that is moved relative to the stack window, or by a script in a resizeStack handler. When using the Geometry Manager with an existing resizeStack handler, be sure to pass the resizeStack message, otherwise the Geometry Manager will not be able to take effect. To force the Geometry Manager to manually update, call revUpdateGeometry.

Tip: Use the Geometry Manager to scale objects with a "divider" bar. Create and script the bar to move, link the edges of the objects to it, then call revUpdateGeometry each time it moves to have the objects scale automatically.

Clipping Settings
Turn on Prevent Object Clipping Text to prevent the control getting too small to display its label when the window is resized. If the control is a field, turn on the option to display scroll bars if the text within the field does not fit.

Limit Settings
Set the minimum and maximum possible widths and heights for the object.

Remove All Settings
Removes all geometry settings from the control. Use this option if the settings applied do not give the desired effect and want to start over.

Geometry Card Settings
The Geometry Card Settings options can be accessed from within the Card Property Inspector. Use these options to determine how geometry is applied to the objects within the current card.

Add To Cards Virtual Width Or Height
Use this option to implement a layout that allows a section of optional objects to be folded out. The Geometry Manager will ignore the extra height or width pixels specified in this area, resizing objects as if that area of the card has not been "expanded". Normally these values are set by script as the window is resized to fold out additional objects. To set these properties by script, set the cREVGeneral["virtualWidth"] or cREVGeneral["virtualHeight"] card properties.

Update Before Opening Card
Causes the objects to be resized when navigating to the card if the window has been resized while on another card. This option is not needed if the objects are contained within a background that has already been correctly resized to the current window dimensions.

4.20) User Interface Design Tips
If creating a simple utility for yourself, a handful of other people, or as a research project, the design of the interface is less important. However, if you are creating software for a wide group of users, take time to carefully design the user interface. As computer software has become more mature, users expect clear, uncluttered interfaces which are visually appealing. Getting this right is part art and part science.
o Less is more. The fewer choices a user has the easier the app will be to learn.
o Get defaults right. Preferences are great for power users but the majority of users never adjust defaults.
o Make a clean layout. Convey professionalism and make the app usable. Line objects up carefully.
o Screen resolution. Decide if the interface will be resizable early on.
o Consider program flow to guide a user step by step through a complex task.
o Test on real users. Don't interfere or help out, just get them going and take notes.
o Don't use an object with a well understood function to perform a different task.
==========

CHAPTER 5) CODING
Script Structure, Handler, Compiling, Events, Message Path, Command & Function, Variable, Constant, Container, Operator, If-Then-Else, Switch, Extending Message Path, Code Library, BackScript, FrontScript, Stack Script, Send Message, Behavior Button, Timer Based Message, Code Tips.

Writing code is how you give your application functionality. Writing the right code means your app will do what you want it to do. The English-like syntax is easy to read and write.

5.1) Script Structure

Every object can contain a script, which tells it what to do. Edit the script of an object using the Code Editor. A script is organized into
a set of individual message handlers. Each handler can respond to a different event or contain a specific piece of functionality. Scripts can send messages to each other. In a well organized application, a script will regularly pass data between different types of message handlers organized to deliver functionality with a minimum of duplication. A script can contain comments and sections of code that are not executed.

5.2) Handler
A handler is a complete section of code. Each handler can be executed independently. There are four types of handlers: message, function, getProp, and setProp handlers.

Message Handler
Each Message handler begins with the on control structure followed by the name of the message the handler responds to. The handler ends with the end control structure and the name of the message. A message handler is executed when the object whose script contains the handler receives the message.

```
on mouseUp  --comment
  beep
end mouseUp
```

Function Handler
Each Function handler begins with the function control structure followed by the name of the function the handler computes. The handler ends with the end control structure and the name of the function. Function handlers are typically called by another handler and return a value using the return control structure. A Function handler is executed when a handler in the same script or message hierarchy calls the function.

```
on mouseUp
  put currentDay into field "Day"  --call the function
end mouseUp

function currentDay  --called from a Message handler
  return item 1 of the long date  --sunday, monday...
end currentDay
```

GetProp Handler - for custom properties.
Each getProp handler begins with the getProp control structure followed by the name of the custom property the handler corresponds to. The handler ends with the end control structure and the name of the custom property. GetProp handlers return a value using the return control structure. A getProp handler is executed whenever the value of the corresponding custom property is requested by a code statement.

```
getProp cMyCustomProperty
  return the scroll of me + 20
end cMyCustomProperty
```

SetProp Handler - for custom properties.
Each setProp handler begins with the setProp control structure followed by the name of the custom property the handler corresponds to. The handler ends with the end control structure and the name of the custom property. A setProp handler is executed whenever the value of the corresponding custom property is changed by the set command.

```
setProp cMyCustomProperty newValue  --newValue is a parameter containing "true" or "false"
  set the hilite of me to newValue
  pass cMyCustomProperty  --allow pass along message hierarchy
end cMyCustomProperty
```

If the custom property is in a custom property set, use a setProp handler with array notation in the first line. The following setProp handler responds to changes in the custom property "cMyCustomProp", which is a member of the custom property set "mySet".

```
setProp mySet[tProperty] newValue
  if tProperty = "cMyCustomProp" then put newValue into me
end setProp
```

Note: Either set the property explicitly in a setProp handler, or include the pass control structure for the Engine to set the custom property.

Comments

Comments are helpful remarks to read that are not executed. Comments can be placed anywhere within a script. A line or portion of a line that starts with two dashes (--) or a hash mark (#) is a comment. Placing these characters at the beginning of a line is called "commenting out" the line. Temporarily remove a command line statement, or even an entire handler, by commenting it out.

```
on mouseUp  --this a comment
  beep
  --this is a comment too
end mouseUp
```

To comment out several lines at once, select them and choose Script > Comment. Comments that start with -- or # only extend to the end of the line. To create a multi-line comment, or block comment, surround it with /* and */ instead. Block comments are handy to temporarily remove a section of code while debugging a handler. Place /* at the start of the section and */ at the end, to prevent the section from being executed.

```
on openCard
  /* This is a multi-line comment that might contain extensive information about this handler,
  such as the author's name, a description, or the date the handler was last changed. */
  show image "My Image"
  pass openCard /* You can also make one-line comments. */
end openCard
```

5.3) Compiling A Script

A script is compiled by clicking Apply in the Script Editor. While compiling the entire script is analyzed. If a compile error is found the entire script is unavailable for execution until the error is corrected and the script re-compiled. This is only for compile errors. An execution error during the execution of a handler does not affect the ability to use other handlers in the same script. A script cannot be edited while a handler in it is executing.

5.4) Events

Coding is based upon events. Every action a script takes is triggered by an event, which is sent in the form of a message. Events include user actions (typing a key or clicking the mouse button) and program actions (completing a file download or quitting the application). Events send a message to the appropriate object.

When a tool other than the Browse tool is active, the IDE traps the built-in messages that are normally sent when clicking so you can use the Pointer tool to select and move a button without triggering its mouseUp handler. These messages are referred to as built-in messages, and include mouseDown, mouseUp, keyDown, openCard, and all the other messages described in the Dictionary. Built-in commands are executed directly by the engine and also don't result in sending a message.

To respond to a message, write a message handler with the same name as the message. To respond to a keyDown message sent to a field (sent when the user presses a key while the insertion point is in the field), place a keyDown handler in the field's script:

```
on keyDown theKey -- responds to keyDown message
  if theKey is a number then beep
end keyDown
```

5.5) The Message Path

The message path is the set of rules that determine which objects, in which order, have the opportunity to respond to a message. The message path is based on the object hierarchy.

FrontScript(s) > Object > Group > Card > BackgroundGroup1...> Stack > MainStack > Library Stack(s) > BackScript(s) > Engine

Each object is part of another object, of a different object type. For example, each card is part of a stack, each grouped control is part of a group, and so on. This object hierarchy defines the ownership and inheritance relationship between objects. Font, color, and pattern properties are inherited from the object's owner if they are not set. If the textFont of a stack is set, all the objects within that stack without their textFont property explicitly set will use that text font.

When a message is sent to an object, it is often handled directly by a message handler in that object. However if no handler is present, the message will continue along a path until it finds a message handler that can respond to it. This makes it possible to group similar functionality together at different levels within the application. This applies both to event messages sent as a result of a user action, and custom messages sent by script. This makes it possible to write libraries of common functions. The object hierarchy is closely related to the path that a message travels on. In most cases, when an object passes a message on, the message goes to the object's owner in the object hierarchy.

Suppose the user clicks a button in a mainstack sending a mouseUp message to the button. If the button's script does not contain a handler for the mouseUp message, the message is passed along to the card the button is on. If the card's script contains a mouseUp handler, the handler is executed. But if the card does not handle the mouseUp message, it is passed on to the card's stack. If the stack script contains a mouseUp handler, the handler is executed. But if the stack does not handle the mouseUp message, it is passed on to the engine. The engine is the end of the message path, and if a message reaches it, the engine takes any default action (like inserting a text character into a field or highlighting a button), then throws the message away.

If a message corresponding to a custom command or a custom function call reaches the end of the message path without finding a handler, instead of being thrown away, it causes an execution error.

Tip: In order to be considered a background along the message pathway, a group must have its backgroundBehavior property is set to true.

Message Target

The object that a message was originally sent to is called the message's target. You can get the target from within any handler in the message path by using the target function.

Click a button causing a mouseUp message to be sent, and if the button's script contains a mouseUp handler, then the target function returns the button's name. However, if the button doesn't handle the mouseUp message, it's passed to the card, and if the card has a mouseUp handler, it is executed in response to the message. In this case, the card's script is executing, but the target is not the card -- it's the button that was originally clicked, because the mouseUp message was first sent to the button.

Tip: To get the name of the object whose script is currently executing, use the me keyword.

```
on mouseUp
  beep
  put the target  --name of button first clicked returned to the Message Box
  put ","&& me after msg  --could be name of button or card
end mouseUp
```

Handlers With The Same Name

If two different objects in the message path each have a handler with the same name, the message is handled by the first object

that receives it and has a handler for it. Suppose a button's script includes a mouseUp handler and so does the stack script. Click the button and a mouseUp message is sent to the button because the button's script has a mouseUp handler and is first in the message pathway. The button handles the message, and doesn't pass it any further. The message is never sent to the stack script, so for this click event the stack script's mouseUp handler is not executed.

If an object's script has more than one handler with the same name, the first one is executed when the object receives the corresponding message. Other handlers in the same object's script with the same name are never executed.

Trapping Messages
When an object receives a message and a handler for that message is found, the handler is executed. Normally, a message that's been handled does not go any further along the message path. Its purpose having been served, it disappears. Such a message is said to have been trapped by the handler. To prevent a message from being passed further along the message path, but don't want to do anything with it, an empty handler for the message will block it from being passed on. Use the same technique to block custom function calls, getProp calls, and setProp triggers.

```
on mouseDown  --mouseDown message blocked with empty handler
end mouseDown
```

Blocking System Messages
Block system messages while a handler is executing (like those sent when navigating to another card) by setting the lockMessages property to true. If a handler opens another stack, openCard and openStack messages are sent to the stack. If the stack contains handlers for these messages that would cause unwanted behavior , use the lockmessages command before going to the stack in order to temporarily block these messages. When the handler finishes executing, the lockMessages property is automatically reset to its default value of false, and normal sending of system messages resumes.

Tip: To block navigation messages while testing or debugging a stack, choose Development > Suppress Messages, or press Suppress Messages on the toolbar.

Passing A Message To The Next Object
To let a message pass further along the message path, use the pass control structure. The pass control structure stops the current handler and sends the message on to the next object in the message path, just as though the object hadn't contained a handler for the message.

```
on openCard
  SetupThisCard  --custom command
  pass openCard  --let stack get the message too
end openCard
```

Selectively Passing Or Trapping Messages
Some built-in messages, such as keyDown, trigger an action, so trapping the message prevents the action from being performed at all or only a selective action.

```
on keyDown pKey  --allow only numbers typed
  if pKey is a number then pass keyDown
end keyDown
```

Placed in a field's script, this handler allows the user to enter numbers, but any other keystrokes are trapped by the keyDown handler and not allowed to pass. A similar principle applies to setProp triggers. If a setProp handler does not pass the setProp trigger, the custom property is not set.

Groups, Backgrounds, & The Message Path
A group's position in the message path depends on whether the "Behave As Background" check box has been set by script or the backgroundBehavior property. If the backgroundBehavior is false, the group is in the message path for all objects it owns, but is not in the message path of any other object. If the background behavior is true, the group is also in the message path for any cards it is placed on. If you send a message to an object, the message passes through the object, then the card, then any

background groups on the card in order of number, then the stack.

Object > Group > Card > Stack --backgroundBehavior false
Object > Card > BackgroundGroup1...> Stack --backgroundBehavior true

Since a group owns any objects that are part of the group, if you send a message to an object within a group, the group is in the message path for its own objects, regardless of whether its backgroundBehavior is true or false. If a group has already received a message because it was originally sent to one of the objects in the group, the message is not sent through the group again after the card has handled it.

Tip: If you want a handler in a group's script to affect only the objects in the group, place the following statement at the beginning of the handler to filter out objects not in the group:

```
on mouseUp
  if the owner of the target is not me then pass mouseUp
  --more code
end mouseUp
```

5.6) Commands And Functions
Use commands and functions to perform the actions of your application. Commands instruct the application to do something such as play a movie, display a window, or change a property. Functions compute a value such as add a column of numbers, get the fifteenth line of list, or find out whether a key is being pressed.

Using Built-in Commands And Functions
There are over one hundred and fifty built-in commands, and over two hundred built-in functions, all documented in the Dictionary.

Commands
A command is an instruction to to do something. A command is placed at the start of a command line statement in the Script Editor. The command is followed by any parameters that specify the details of what the command is to do. Here are examples of how built-in commands are used:

```
go next card  --go command
beep  --beep command
set the hilite of me to "true"  --set command
get line 2 of field 1  --get command
```

Functions
A function call is a request for information. A function is stated using the name of the function, followed by opening and closing brackets that may contain any parameters that specify the details of what the function is to act on. When you use a build-in function in a command line statement, the function call computes the specified parameters and substitutes that information in the script in place of the function call. The info returned can be put into a container by using the put command. A function can have one parameter, several, or none. The parameters are enclosed in parentheses and, if there's more than one, they're separated with commas. Here are examples of how built-in functions are used:

```
put round(22.3) into field "Number"  --22, round function
put specialFolderPath("desktop") &"/MyText.txt" into tPath  --specialFolderPath function
put date() into field "Today"  --date function
put length(item 2 of it) into the message box  --length function
get average(10,2,4)  --average function
```

Tip: A function call is not a complete statement. It must be used in conjunction with a command or control structure.

If a built-in function has no parameter or one parameter, it can be written without parentheses. If the function has no parameters, it's written as: the functionName. If it has one parameter, it's written as: the functionName of parameter. Use either form interchangeably. The Dictionary entry for each built-in function shows how to write both forms.

```
put the date into field "Today"  --date function
put the length of item 2 of it into the message box  --length function
```

Tip: Use the "the" form for built-in functions, not for custom functions you write.

Custom Commands And Functions

Use custom commands and custom functions the same as any other command or function. Execute a custom command by typing the name of the command in a command line statement in the Script Editor.

Custom Commands

```
on mouseUp
  MyCommand  --custom command
end mouseUp
```

Respond to a custom command's message in the same way, by writing a message handler with the name of the command.

```
on MyCommand  --message handler for custom command
  beep 3
  go next card
  add 1 to field "Card Number"
end MyCommand
```

If you don't specify an object, the message is sent to the object whose script is being executed, and then passes up the message hierarchy as normal. Like a built-in command, a custom command is an instruction to do something. You can include parameters with a custom command by passing them after the name: MakePrefsFile pFileLoc, pPrefsField

Built-in commands can have very flexible syntax:

```
go to card 3 of stack "Dialogs" as modal
group image "Help Icon" and field "Help Text"
hide button 1 with visual effect dissolve very fast
```

The syntax of custom commands is more limited. A custom command can have several parameters separated by commas. But custom commands cannot use words like "and" and "to" to join parts of the command together, the way built-in commands can. Because of this, custom commands cannot be as English-like as built-in commands:

```
MyNextCommand myFile,newPath,"downloadComplete"
ExecuteNow myDatabase,field "Query",myRecords
```

Custom Functions

Like a built-in function, a custom function call is a request for information. Here are a few examples of made-up custom functions:

```
get formattedPath("/Disk/Folder/File.txt")
put summaryText(field "Input") into field "Summary"
if handlerNames(myScript, "getProp") is empty then beep
```

Custom functions don't have a "the" form, and are always written with parentheses following the function name. If the function has parameters, they are placed inside the parentheses, separated by commas. If the function doesn't have parameters, the parentheses are empty.

Here is custom function "fileHeader" in a command line statement. The custom function handler is executed, and the function call is replaced by the returned value:

```
on mouseUp
  put fileHeader(it) into tFileHeader  --custom function call
end mouseUp
```

```
function fileHeader theFile  --custom function handler with 1 parameter
  put char 1 to 8 of URL ("file:" & theFile) into tempVar
  return tempVar
end fileHeader
```

Passing Parameters

A value passed from one handler to another is called a parameter. In the example below, the following statement sends the "alertUser" message with a
parameter:

```
on mouseUp
  alertUser "You clicked a button!"
end mouseUp
```

The parameter "You clicked a button!" is passed to the "alertUser" handler, which accepts the parameter and places it in a parameter variable called "theMessage". The "alertUser" handler can then use the parameter in its statements:

```
on alertUser theMessage
  beep
  answer theMessage -- uses the parameter "theMessage"
end alertUser
```

Another example using two parameters, "theMessage" and "numberOfBeeps":

```
on mouseUp
  seriouslyBugUser "Hello!",5
end mouseUp

on seriouslyBugUser theMessage,numberOfBeeps
  beep numberOfBeeps  --5 beeps
  answer theMessage  --Hello
end seriouslyBugUser
```

In the example above, "theMessage" and "numberOfBeeps" are the parameter variables. Declare parameter variables in the first line of a handler. When the handler begins executing, the values passed are placed in the parameter variables. Use parameter variables the same as ordinary variables, put data into them and use them in expressions. Parameter variables are local variables, so they disappear as soon as the handler stops executing.

Parameter Variable Names

Give a parameter any name that's a legal variable name. The names of variables must consist of a single word and may contain any combination of letters, digits, and underscores (_). The first character must be either a letter or an underscore. It is not the name, but the order of parameters that is significant.

Empty Parameters

A handler can have any number of parameter variables regardless of the number of parameters passed to it. If there are more parameter variables than there are values to put in them, the remaining parameter variables are left empty. Consider the following handler, which has three parameter variables:

```
on processOrder itemName,itemSize,numberOfItems
  put itemName into field "Name"
  put itemSize into field "Size"
  if numberOfItems is empty then put "1" into field "Number"
  else put numberOfItems into field "Number"
end processOrder
```

The following statement places an order for one sweater:
processOrder "sweater","large"

The statement only passes two parameters, while the "processOrder" handler has three parameter variables, so the third parameter variable, "numberOfItems", is empty. Because the handler provides for the possibility that "numberOfItems" is empty, you can pass either two or three parameters to this handler. To use a default value for a parameter, check whether the parameter is empty. If it is, then no value has been passed, and put the desired default into the parameter.

Implicit Parameters
If a statement passes more parameters than the receiving handler has parameter variables to hold them, the receiving handler can access the extra parameters with the param function:

```
function product firstFactor,secondFactor
  put firstFactor * secondFactor into theProduct
  if the paramCount > "2" then
    repeat with x = 3 to the paramCount
      multiply theProduct by param(x)
    end repeat
  end if
  return theProduct
end product
```

The function above assumes that two parameters will be passed to be multiplied, but can multiply more numbers by using the param function to access parameters beyond the first two. The following statement uses the "product" custom function above to multiply four numbers together:

```
on mouseUp
  answer product(22,10,3,7)
end mouseUp
```

When the custom function handler executes, the first two parameters, 22 and 10, are placed in the parameter variables "firstFactor" and "secondFactor". The third parameter, 3, is accessed with the expression param(3), and the fourth parameter, 7, is accessed with the expression param(4).

Passing Parameters By Reference
Normally, the name of a parameter variable is not changed by anything the called handler does. The variable name is not passed, only its contents. Passing parameters in this way is called "passing by value". This is the most common method of passing parameters.

But if the name of a parameter variable is preceded with the "@" character, that parameter's value becomes a variable name. This way of passing parameters is called "passing by reference", because you pass a reference to the variable instead of its value. This is a rarely used alternative method of passing parameters. For example, the following handler takes a parameter and adds 1 to it:

```
on setVariable @incomingVar
  add 1 to incomingVar
end setVariable
```

The following handler calls the "setVariable" handler above:

```
on mouseUp
  put "8" into tVariable
  setVariable tVariable  --call by reference
  answer "tVariable is now:" && tVariable  --9
```

```
end mouseUp
```

Executing this mouseUp handler displays a dialog box that says "tVariable is now: 9". This is because, since "tVariable" was passed by reference to the "setVariable" handler, its value was changed when "setVariable" added 1 to the corresponding parameter variable.

Pass parameters by reference to any custom command or custom function by preceding the parameter name with the "@" character in the first line of the handler, as in the "setVariable" example handler above. Do not use the "@" character when referring to the parameter elsewhere in the handler. If a parameter is passed by reference, you can pass only variable names for that parameter. You cannot pass string literals or expressions using other containers such as fields. Trying to use the "setVariable" command described above using the following parameters will cause an execution error:
setVariable 2-- can't pass a literal by reference
setVariable field 2 -- can't pass a container
setVariable line 1 of someVariable -- can't pass a chunk

Tip: If a handler defines a parameter as being passed by reference, you must include that parameter when calling the handler. Omitting it will cause an execution error.

Returning Values
Once a function handler has calculated a value, it needs a way to send the result back to the message handler that called the function. And if an error occurs during a message handler, it needs a way to send an error message back to the calling handler.

The return control structure is used within a custom function handler to pass the resulting value back to the calling handler. The returned value is substituted for the function call in the calling statement, just like the value of a built-in function.

```
function expanded theString
  repeat for each character i in theString
    put i & space after expandedString
  end repeat
  delete char -1 of expandedString  --last space
  return expandedString
end expanded
```

In the custom function example above, the return control structure sends the spaced-out string back to the mouseUp handler that called it. The return control structure stops the handler, so it's usually the last line.

When used in a message handler, the return control structure sets the result function for the calling handler. This is a rarely used alternative use of the return control structure. Here's an example of a message handler that displays a dialog box:

```
on echoAMessage
   ask "What do you want to show?"
   if it is empty then return "No message!"  --return "no message" in the result
   else answer it with "OK"
end echoAMessage

on mouseUp
   echoAMessage
   if the result is not empty then answer "The result =" && the result with "OK"
end mouseUp
```

The handler asks the user to enter a message, then displays that message in a dialog box. If the user doesn't enter anything or clicks Cancel, the handler sends an error message back to the calling mouseUp handler in the result function where it can check to see whether the command successfully displayed a message.

A more common method without using a return control structure for error:

```
on mouseUp
  ask "What do you want to show?"
  if it is empty then answer "No message!" with "OK"
  else answer it with "OK"
end mouseUp
```

Tip: The result function is also set by many built-in commands in case of an error. If checking the result, do it right after the command or put the result into a temporary container before it is lost or changed.

Command And Function Summary

o A command instructs the application to do something, while a function requests the application to compute a value.
o Create a custom command or custom function by writing a handler for it.
o Values passed to a handler are called parameters.
o To pass a parameter by reference, precede its name with an "@" sign in the first line of the handler.
o The return control structure in a function handler returns a value.
o The return control structure in a message handler returns an error message to the result function.

5.7) Variables

A variable is a place to store data that you create, which has no on-screen representation. Variables can hold any data you want to put into them. One way to think of a variable is as a box with a name on it. You can put anything you want into the box, and take it out later by providing the variable's name:

```
put "1" into tThing  --local variable container named "tThing"
put tThing into field 2
put "Hello Again!" into line 2 of tThing
```

But unlike some other types of containers, variables are non-permanent and aren't saved with the stack. Instead, variables are automatically deleted either when their handler is finished running or when you quit the application, depending on the variable's scope. You can also use the delete variable command to delete a variable before it would normally disappear. When a variable is deleted, not only the content of the variable disappears, but also the variable itself, the "box".

Tip: To save a variable's value, set a custom property of the stack, to the value of the variable in your application's closeStackRequest or shutDown handler. To restore the variable, put the custom property into a new variable in the application's startUp or openStack handler.

The scope of a variable is the part of the application where the variable can be used. If you refer to a variable in a handler that's outside the variable's scope, you'll get either an execution error or an unexpected result. There are three levels of variable scope: local, script local, and global. Every variable is one of these three types. The difference between these scopes is where they can be used and how long their value lasts.

Local Variables

A local variable can be used only in the handler that creates it. Once the handler finishes executing, the variable is deleted. The next time you execute the handler, the variable starts from scratch: it does not retain what you put into it the last time the handler was executed.

To create a local variable, put something into it. If you use the put command with a variable name that does not yet exist, the variable is automatically created as a local variable.

```
put true into myNewVar -- creates variable named "myNewVar"
```

Tip: While you can use any word for a variable name that isn't a language word, known as a reserved word, develop the habit of naming variables logically and consistently. Alternatively, you can create a local variable explicitly by declaring it using the local command inside a handler.

If a handler with a local variable calls another handler with the same local variable name, a new second variable is automatically

created for the second handler to prevent conflicts. In the following example, the two handlers each have a local variable named "myVar", but one won't affect the other.

```
on mouseUp
  put "1" into myVar -- create a local variable
  doCalledHandler  --custom command
  answer myVar  --displays 1, not 2
end mouseUp

on doCalledHandler
  local myVar  -- explicitly create a different local variable with the same name
  put "2" into myVar
end doCalledHandler
```

One common source of bugs involves misspelling a local variable name. Normally, doing so doesn't produce an execution error, because using a variable that doesn't exist yet creates it automatically. This means if you misspell a variable name, a new variable with the misspelled name is created. Such a bug may be difficult to track down because a variable with a wrong value doesn't trigger an error message.

To prevent this problem, you can require all local variables be declared with the local command. You do this by choosing Script > Variable Checking in the Script Editor menu bar. If this option is on, trying to use a local variable that doesn't exist will cause an execution error instead of automatically creating it. Any misspelled variable names will trigger an obvious execution error when their handler is executed, making them easier to find.

Script Local Variables

A script local variable can be used in any handler in an object's script. You cannot use the same script local variable in handlers in other objects' scripts. Unlike a local variable, a script local variable retains its value even after a handler finishes executing. To create a script local variable, use the local command in the script, but outside any handler, usually declared at the top of a script so they are in one place and easy to find. A script local is available to all handlers below it in the same script.

```
local mySharedVariable  --script local
on mouseDown
  put "2" into mySharedVariable
end mouseDown

on mouseUp
  answer mySharedVariable with "OK" titled "Test Script Local"  --displays 2
end mouseUp
```

Tip: Putting the local command within a handler instead of outside a handler creates a local variable instead. The local command creates a script local variable only if placed outside the handler. The delete variable command can delete a script local variable too.

Global Variables

A global variable can be used in any handler, anywhere in the application. Unlike a local variable, a global variable retains its value even after the handler that created it finishes executing. Unlike a script local variable, a global variable can be used by any handler in any object's script. The same global variable can be used by any stack during a session.

To create a global variable, declare it using the global command. You must also declare the global at the start of a handler to make an existing global variable available to that handler. While a global variable can be used by any handler, you must do this in any handler the global is used it in. If you don't declare a global variable before using it, the handler will instead create a local variable with the same name.

Use the global command inside a handler or outside any handler at the top of a script, like a script local. If declared in a handler, the global variable can be used by any statement in that handler. If declared in a script outside any handler, the global variable is

available to every handler in that script. In this example, the global variable is declared outside any handler, so available to all handlers and individual handlers don't need to declare it again.

```
global myGlobal  --declare global for whole script
on mouseDown -- can use "myGlobal"
  put "1" into myGlobal
end mouseDown

on mouseUp  --can use "myGlobal"
  add "2" to myGlobal
  answer myGlobal with "OK" titled "Test Global" -- displays "3"
end mouseUp
```

To use the same global variable in a handler in a different object's script, you must place the global declaration within the handler.

```
on mouseUp  --in another button's script
  global myGlobal
  add "5" to myGlobal
  answer myGlobal with "OK" titled "Test Global In Btn2"  --displays 8
end mouseUp
```

Click the first button, then the second. The second button displays the number 8. As with script local variables, place global declarations in scripts at the top of the script, making the global variable easy to find later.

Tip: Get a list of existing global variables with the globalNames function, or choose Development > Variable Watcher to see a list of global variables and change their values, or get the value using the Message Box. Global variables are automatically deleted when the application is quit. You can also use the delete variable command to delete a global variable.

Parameter Variables

In a handler for a custom command or custom function, define parameters on the first line of the handler. For example, the following custom command handler defines two parameter variables named "thisThing" and "thatThing". A custom command or custom function can pass values to the handler using the parameter variables.

```
on myHandler thisThing,thatThing  --custom command handler
  add thisThing to thatThing
  subtract thatThing from field 2
end myHandler

on mouseUp
  myHandler 15,4+1  --puts "15" into parameter "thisThing", puts "5" into parameter "thatThing"
end mouseUp
```

When named parameters are used in a handler, they are called parameter variables. Within the handler, the parameters can be used in the same way as local variables: get their value, use them in expressions, and put data into them. Like local variables, parameter variables persist only as long as the handler is executing.

Variable Names

The names of variables must consist of a single word and may contain any combination of letters, digits, and underscores (_). The first character must be either a letter or an underscore. You cannot use any language word as the name of a variable. Accepted practice is to preface each variable with a letter that indicates its use. t: temporary local, s: script local, g: global, c: custom property, p: parameter, k: constant.

Good variable names:

someVariable

tVariable

Picture3
my_new_file
_Output

Bad variable names:
3rdRock --illegal, starts with digit
this&That --illegal, "&" cannot be used
My Variable --illegal, more than one word

Tip: Avoid giving a variable the same name as a custom property, the contents of the variable will be used to rename the custom property producing unexpected results. Global variables whose names begin with "gRev" are reserved by the development environment and should not be used.

Special Variable Types
Most of the time you create the variable, using the local or global commands, or putting a value into a new variable to create it. The Engine also creates certain types of special variables automatically: environment variables, command-line variables, and the special variable it.

Environment Variables
Most operating systems supported provide information about the operating environment in environment variables. Access environment variables by prepending the "$" character to the variable's name. For example, the following statement gets the contents of the LOGNAME environment variable, which holds the current user's login name:
get $LOGNAME

See your operating system's technical documentation to find out what environment variables are available. Create your own environment variables by prepending the "$" character to the new environment variable's name:
put field 3 into $MYENVIRONMENTVAR

Tip: Environment variables behave like global variables and can be used in any handler, but are not declared before using them. The environment variables that created this way are available to the application, and are also exported to processes started up by the shell function or the open process command.

Command-Line Argument Variables
If you start up the application from a command line, the command name is stored in the variable $0 and any arguments passed on the command line are stored in numbered variables starting with the "$" character. For example, Start the application by typing the following shell command:
myrevapp -h name

Then the variable $0 contains "myrevapp" (the name of the application), $1 contains "-h", and $2 contains "name".

Tip: Command-line argument variables behave like global variables and can be used in any handler, but are not declared before using them.

The Special Variable "It"
The it variable is a special local variable the Engine uses to store certain results. The commands that use the it variable are: answer, answer color, answer effect, answer file, answer folder, ask, ask file, ask password, clone, convert, copy, create, get, paste, post, read from driver, read from file, read from process, read from socket, request, and request appleEvent. See the entry for it in the Dictionary. The following example shows how the answer command uses the it variable.

```
on mouseUp
  answer "Go where?" with "Backward" or "Forward" titled "Navigate"  --clicked button name is put into the it variable.
  if it = "Backward" then go back  --previous card
  else if it = "Forward" then go next  --next card
end mouseUp
```

Tip: Use the it variable the same as any other local variable, using the put command to put things into it and using the variable in expressions to work with the contents.

Array Variables
A variable can hold more than a single value. A variable that holds more than one value is called an array, and each of the values it holds is called an element. Each element has its own name-key. If you think of a variable as a box with a name, think of an array as a box with compartments inside it where each compartment is an element with a name-key.

Specify an element of an array variable by using the array variable name along with the element's name-key. Enclose the key in square brackets. The key may be a name, number, or variable. Here's an example that shows how to put data into one element of an array called myArray:
```
put "ABC" into myArray["myNameKey"]
```

Tip: If a key is not a number or variable, enclose the key in double quotes. This prevents problems in case there is a variable or reserved word with the same name.

Use any element of an array variable to put data into the element (or before or after it) and find out what data it contains. Any element of an array variable is like a variable in and of itself. Array elements may contain nested or sub-elements, making them multi-dimensional. This type of array is ideal for processing hierarchical data structures such as trees or XML. To access a sub-element, declare it using an additional set of square brackets:
```
put "ABC" into myArray["myNameKey"]["aSubElement"]  --nest elements within themselves to any number of levels.
```

Deleting Elements Of An Array
Use the delete variable command to remove one element from an array variable, in the same way you delete a variable. To delete an element, specify both the array variable and the element's key:
```
delete variable myArray["myKey"]
delete variable myArray["myKey"]["sub1"]
```

The first statement deletes the element named "myKey" from the array variable "myArray", but does not delete the other elements in the array. The second statement deletes the subElement named "sub1" from the array variable "myArray", but does not delete the other elements in the array.

Tip: To delete the contents of an element without deleting the element itself, put empty into the element:
```
put empty into myArray["myKey"]
```

Listing The Elements In An Array
Use the keys function to list the elements in an array variable. The keys function returns a list of elements, one per line:
```
put the keys of myArray into tlistOfElements  --line list of keys in an array
```

Listing Nested Elements Within An Element
Use the keys function to list the child elements of an element within an array variable. The keys function returns a list of elements, one per line:
```
put the keys of myArray["myKey"] into listOfElements  --subElements of element myKey.
```

Split a List of Data Into An Array - text to array.
The split command separates the text parts of a variable into elements of an array. Specify what character in the list you want the data to be split by. The data will be converted into a set of elements named according to where it is split. Split by, split with, and split using are all valid. An array cannot be directly displayed in a field or as text. see split in the Dictionary. For example:

```
on mouseUp
  put "A apple,B bottle,C cradle" into myArray  --myArray is a variable containing text
  split myArray by comma and space  --myArray is now an array
end mouseUp
```

Combine An Array Into A List - array to text.

The combine command combines the elements of an array into a single text variable. After the command is finished executing, the variable specified by array is no longer an array. Combine by, combine with, and combine using are all valid. See combine in the Dictionary. The following example will combine the contents of the each element of the array on a separate line.

```
on mouseUp
  combine myArray by cr and comma  --myArray is now a variable containing text, and no longer an array
  put myArray into field 1
end mouseUp
```

Field 1 contents:
A,apple
B,bottle
C,cradle

Nesting An Array
Place an entire array as a child of an element by putting an array variable into an element of another array. For example:
put tMyArray into tBigArray["node50"]

This will result in the entire array tMyArray being placed as a child of key node50 within tBigArray.

5.8) Constants
A constant is a value that has a name. Constants can be built-in or user-defined in a script. Unlike variables, constants cannot be changed. When you use a constant, the Engine replaces the constant's name with its value.

Built-In Constants
The Dictionary defines all built-in constants, such as cr, return, space, and comma, for characters that have special meaning in scripts that can't be entered in any other way.
put slash after field 1 --insert slash's value "/"
put comma &"Hello" after field 1 --insert comma's value ","
put cr &"Hello" after field 1 --insert return's value, a carriage return
---
User-Defined Constants
Create a new constant using the constant command. You cannot put anything into a constant once it's been created.

```
constant myName = "Joe Smith"  --insert "Joe Smith" where used in script
put myName &comma&& tPhone &cr after field 1

constant kFactor = "20"  --insert "20" where used in script
put tAmount * kFactor into tSubTotal
```

Like variables, constants can have different scope depending on how they are created. A constant can be defined as either a local constant or a script local constant. If placed within a handler, the constant can be used only in that handler. If place in a script outside any handler, the constant can be used by any handler in the script.

5.9) Containers, Operators, & Sources of Value
Container
Containers are sources of information that can be edited using chunk expressions. There are seven container types: variables, fields, buttons, images, URLs, the selection, and the message box. Fields, buttons, and imported images are objects that display their content on the screen in different ways, and their contents are saved when you save the stack they are in. URLs refer to external resources (either files on the system, or items on an Internet server). The message box is a special container that's part of the development environment.

Setting And Retrieving Data From Containers

Use the put command to put data into a container, or to put data from one container into another container or variable. Containers support the use of chunk expressions, the ability to specify a portion of a container by referring to it in English.

Sources Of Value
Sources of value are like containers. They can be retrieved using the get command. However unlike containers, sources of value cannot be set using the put command. Sources of value include properties, function calls, literal strings, and constants.

Getting And Setting Properties
Use the get command to get data from properties and the set command to set data of properties. Chunk expressions may be used to get data. You cannot set a property with the put command.

Literal Strings
A literal string is a string of characters whose value is itself. If the string is a number, the value is that number. Literal strings may be quoted. To use a literal string in an expression, simply enter it on a command line.
put "Hello World!" into field 1
get 1 + 2 + 3 --puts 6 into it
put 1 - 4.234 into field "Result"

Quoting Strings
Literal strings that consist of more than one word or are reserved words must be enclosed in double quotes:
put "This is a test" into myVar --works
put This is a test into myVar --doesn't work, not quoted
put That into myVar --works
put This into myVar --doesn't work, reserved word

In some contexts, using an unquoted one-word literal string will not cause a script error. However, make a practice of always quoting literal strings (other than numbers), because it ensures the statement will continue to work properly even if the string becomes a reserved word in the future. If the Script > Variable Checking option is set to true, compiling a script that contains an unquoted literal string causes a script error.

5.10) Operator
Use operators to put together, compare, or perform an operation on data. Use a String Operator to combine data. Use a Numeric Operator to perform a calculation. Use a Logical Operator to return true or false.

Numeric Operators
Numeric operators produce a number as their result. Numeric operators include the arithmetic operators (+, -, *, /, mod, div, and ^) and the bitwise operators (bitAnd, bitOr, bitXOr, and bitNot). For example:
put "1+2 =" && 1+2 into field 1 --displays: 1+2 = 3

String Operators
String operators produce a string of characters as their result. String operators are the concatenation operators (&, &&, and ,). For example:
put "Hello" && tName &","&& "how are you?" into field 2 -- displays: Hello Tom, how are you?

Logical Operators
Logical operators produce either "true" or "false" as their result. Logical operators include the comparison operators (=, <>, <, >, <=, >=), existence operators (there is a, there is no, is in, is not in, is among, is not among, contains), data type operators (is a, is not a), geometry operators (is within, is not within), and basic logical operators (and, or, not). For example:
if the platform is "MacOS" and field "Time" < zero then...
if tAmount >= "20" and tToday is a date then...

Unary versus Binary Operators
Operators can use either one argument (a unary operator) or two arguments (a binary operator). The bitNot, there is a, there is no, is a, is not a, and not operators are unary operators. All other operators are binary operators.

there is a field "C" --"there is a" is unary
"a" & "b" --"&" is binary

Conversion Of Values
Values in expressions are converted to whatever type of data is needed for the operation. This conversion happens automatically. For example:
put char 2 of "123" + char 3 of "456" into field 3 --26

Character 2 of the literal string "123" is the single-character string "2", and character 3 of the literal string "456" is the single-character string "6". The + operator automatically converts these strings to numbers so they can be added together, then converts the resulting number back to a string to be placed in a field as text.

Operator Precedence
Precedence determines the order of calculations. If an expression contains more than one operator, the operators with higher precedence are calculated before operators with lower precedence.

o Any part of the expression in parentheses is evaluated first. If parentheses are nested, the innermost values are evaluated first.
o Next, unary operations (that act on only one operand) are done. This includes unary minus (which makes a number negative).
o Exponentiation operations are done next.
o Multiplication and division are done next. These are numeric operators and result in a number.
o Addition and subtraction are done next. These are numeric operators and result in a number.
o Operations that join two strings are done next. These are string operators and result in a string.
o Operations that compare two values are done next. These are logical operators and result in either true or false.
o Operations that compare two values for equality are done next. These are logical operators and result in either true or false.
o bitAnd operations, then bitXOr operations, then bitOr operations are done next.
o And operations are done next.
o Or operations are done last.
o Functions are evaluated after all possible operators in the function's parameters are evaluated.

The Grouping Operator ()
The grouping operator () has the highest precedence. Use it to change the order in which operators are evaluated. Enclosing part of an expression in parentheses forces any operations inside the parentheses to be evaluated first. If parentheses are nested, the expression within the innermost set of parentheses is evaluated first.
put 3 + 4 * 2 into field 1 --11, 3 + 8 = 11
put (3 + 4) * 2 into field 2 --14, 7 * 2 = 14

Factors And Expressions
An expression is any source of value, or combination of sources of value. Any of the sources of value discussed above – containers, properties, function calls, literal strings, and constants – are simple expressions. You use operators to combine sources of value to produce more complex expressions.

Defining Factors
A factor is the first fully resolvable portion of an expression. (All factors are expressions, but not all expressions are factors.) A factor can be either a source of value, or an expression that combines sources of value but doesn't include any binary operators outside parentheses. Enclosing an expression in parentheses turns it into a factor. These examples show some expressions, along with the first factor in each expression.

Expression --First Factor
3 + 4 --first factor is 3
(3 + 4) --first factor is (3 + 4)
(3 + 4)/field 4 --first factor is (3 + 4)
field 6 * pi --first factor is field 6
sin of pi/4 --first factor is sin of pi
sin(pi/4) --first factor is sin(pi/4)
sin(pi/4) * exp2(12) --first factor is sin(pi/4)

```
whole world  --first factor is whole
"whole world"  --first factor is "whole world"
```

The distinction between factors and expressions matters when using the "the" form of built-in functions, URL, and referring to objects. If using the "the" form of a built-in function that has a parameter, the parameter must be a factor, not an expression:

```
get the sqrt of 4 + 5  --7, 2+5
get sqrt(4+5)  --3, square root of 9
get the sqrt of (4 + 5)  --3, square root of 9
```

In the first example above, although we intended to get the square root of 9, we actually ended up with the square root of 4. This is because the expression 4 + 5 is not a factor (because it contains a binary operator that's not inside parentheses). The first factor in the expression 4 + 5 is 4, so the first example gets the square root of 4 (which is 2), then adds 5 to that result.

The second example avoids this problem because it doesn't use the "the" form of the function. Since the parameter is enclosed in parentheses, use either a factor or an expression, and obtain the intended result, the square root of 9.

The third example turns the expression 4 + 5 into a factor by surrounding it with parentheses. Since the parameter is now a factor, we can get the correct result, even using the "the" form of the function.

When referring to URLs, the URL must be a factor:

```
get URL "file:myfile.txt"  --works
get URL "file:" & "myfile.txt"  --doesn't work
get URL ("file:" & "myfile.txt")  --works
```

In the first example, the URL specified is a factor because it is a simple string with no operators. The URL in the second example is not a factor, because it includes the binary operator &, so the get command tries to get the URL "file:", which is nonexistent, and concatenate the content of that URL with the string "myfile.txt". The third example turns the URL into a factor by surrounding it with parentheses, providing the expected result.

When referring to cards or backgrounds, the name, number, or ID of the object is an expression:

```
go card 1 + 1  --goes to card 2
go card 3 of background "Data" && "Cards"  --goes to first card with the group "Data Cards"
```

However, when referring to objects (including groups) or stacks, the name, number, or, ID of the object is a factor:

```
answer field 1 + 10  --displays field 1 content + 10
answer field (1 + 10) -- displays field 11 content
select button "My" && "Button"  --doesn't work
select button ("My" && "Button")  --works
```

5.11) If-Then-Else Control Structure

Make decisions using the if control structure or choose from a list of options using the switch control structure.

Use the if control structure to execute a statement or list of statements under certain circumstances. It always begins with the word if. There are four forms of the if control structure:

```
if condition then statement [else statement]

if condition then
  statementListA
[else
  statementListB]
end if

if condition
```

```
then statementA
[else
  statementListB
end if]

if condition then
  statementListA
else statementB
```

The condition is any expression that evaluates to true or false. The statement consists of a single statement. The statementList consists of one or more statements, and can include if, and, or, switch, try, and repeat control structures.

If the condition evaluates to true, the statement or statementList is executed. If the condition evaluates to false, the statement or statementList is skipped. If the if control structure contains an else clause, the elseStatement or elseStatementList is executed if the condition is false.

If one if control structure is nested inside another, use of the second form to prevent ambiguities.

The if control structure is most suitable to check a single condition. To check for multiple possibilities and doing something different for each one, use a switch control structure instead.

5.12) Switch Control Structure

Use the switch control structure to choose among several possible values for an expression and then execute a set of statements that depends on the value.

```
switch [switchExpression]
  case {caseValue | caseCondition}
    [statementList]
    break
  [default
    defaultStatementList]
end switch
```

Here are two examples:

```
on mouseUp
  put any item of "3,4,5,6" into tVar
  switch tVar  --if switchExpression then use equal caseValue
    case "3"
      answer "You have a 3."
      break
    case "4"
      answer "You have a 4."
      break
    default
      answer "You have a 5 or 6."
  end switch
end mouseUp

on mouseUp
  put random(25) into tVar  --1 to 25
    switch  --if no switchExpression then use true caseCondition
      case tVar <= "10"
        answer "You're 10 or under."
        break
```

```
    case tVar >= "11" and tVar <= "20"
      answer "You're from 11 to 20."
      break
    default
      answer "You're from 21 to 25."
  end switch
end mouseUp
```

The switch control structure begins with the word switch on a single line, with an optional switchExpression. The switch line is followed by one or more case sections. Each case section begins with the case keyword, followed by either a caseValue (if a switchExpression was included on the switch line) or a caseCondition (if no switchExpression was included).

If the caseValue is equal to the switchExpression, or the caseCondition evaluates to true, the following statements are executed. The case sections may be followed by an optional default section. If no break statement has been encountered yet in the switch control structure, the statements in the default section are executed. The switch structure ends with an end switch statement.

Each statementList consists of one or more statements, and can include if, and, or, switch, try, and repeat control structures.

Tip: When a matching condition is found, all statements following it are executed, even statements in another case section, until either a break statement is encountered or the switch control structure ends. In most cases, place a break statement at the end of each statementList to ensures only one statementList is executed and the rest are skipped.

This also means that you can attach more than one caseValue or caseCondition to the same statementList, by placing one case line above the next. The following example beeps if the current card is either card 1 or the last card, and goes to the next card otherwise. Remove the break statement and it always goes to the next card.

```
on mouseUp
  switch (the number of this card)
    case 1
    case (the number of cards)  --both cases execute the next statement
      beep
      break
    default
      go next card
  end switch
end mouseUp
```

An equivalent if statement would be:

```
on mouseUp
  if the number of this card = 1 or the number of this card = the number of cards then beep
  else go next card
end mouseUp
```

5.13) Extending The Message Path

This section deals with how to extend the message path, either by adding code libraries to the message path, or by sending messages directly to objects that are not currently in the message path.

Creating A Code Library

A library is a set of custom commands and custom functions for a specific application or a specific area of functionality. Libraries are typically used to store routines common across an application. Useful libraries can be shared with other developers.

To create an object code library, place the handlers used by any object in a stack in an object's script. The object is now a code library. Then use the insert script command to add that object's script to the message path. For example:

```
insert the script of button "Message Library" into back  --handlers now available to all objects in stack.
```

To create a stack code library, place the handlers in the stack's script. The stack is now a code library. Then use the start using command to add the stack's script to the message path.
start using stack "MyLibrary"

Run one of these commands as the application is starting up, so all the scripts can access the libraries required. Libraries do not need to be in the same stack or even stack file; you can load any stack on disk and then load the libraries within it to make them available to all running stacks. This makes it easier to design an app in modules, share code with other developers, or update application libraries without modifying the application. A standalone app can be designed to work the same way, updating a library with a small patch utility without having to reinstall the entire standalone.

Using BackScripts
To make the script of an object available to any other handler:
insert script of card "Library" into back

The script of an object that's been inserted into the back is called a backScript. Such an object is placed last in the message path of all objects. It receives messages after any other object, and just before the engine receives the message. The backScript object remains in the message path until the end of the session or until removed with the remove script command. Because a backScript receives messages after all other objects in the message path, it eventually receives all messages, no matter what object sent them, unless another object handles them first.

Using FrontScripts
The message path can also be extended by placing objects into the front of the message path, before any other object. The script of such an object is called a frontscript. Use the insert script command to place the object in the front:
insert script of button "Interceptor" into front

Because the object is in the front of the message path, it receives messages even before the target of the message. For example, if you click a button, any objects in the front of the message path receive the mouseUp message first, before the button. After placing a handler in a frontscript, it receives all the corresponding messages before any other object can handle them.

Use a frontscript to handle a message even if the target object has a handler for the message. For example, the development environment displays a contextual menu when you Control-Shift-RightClick an object. It does this with a mouseDown handler in a frontscript. Whenever an object is clicked, the frontscript receives the mouseDown message first, and checks whether the needed keys are being pressed. If they are, the handler displays the contextual menu, the mouseDown message is trapped and goes no further. Otherwise, the handler passes the message to the next object in the message path, which is the object clicked.

Using A Stack's Script With Start Using
The start using command is similar to the insert script command, but is used only to place stacks in the message path. The start using command inserts the stack at the end of the message path, after the object's stack and mainstack, but before objects that have been inserted into the back with insert script.

Sending Messages Directly To Objects
To use a handler not in the message path, an alternative to inserting a code library is to use the send command to directly send a message to the object whose script contains the handler.

```
on mouseUp  --in button script
  send "mouseUp" to this stack  --send a message directly to an object further up the hierarchy.
end mouseUp

on mouseUp  --in button script
  send "mouseUp" to button "AnotherButton"  --send a message directly to an object outside the hierarchy.
end mouseUp
```

For example, to send a message directly to a button whose script contains the handler:
send "myCommand" to button "Stuff" of card "Stuff Card" --custom command handler in button "Stuff"

Tip: The message path for a sent message starts with the target object, not the sending object.

To use a custom function whose function handler is not in the message path, use the value function to specify the object. The value function can be thought of as an equivalent of the send command for function calls. For example, if card 1 script contains a custom function handler called "myFunction", to use the function from within the script of card 3:

```
get value("myFunction(22)",card 1)  --myFunction handler in script of card 1
```

The Send Command Versus The Call Command

The call command is similar to the send command. Like the send command, the call command sends a message to the specified object to use handlers not in the message path. The difference between send and call is how they handle object references in the triggered handler. Send uses the context of the object the handler is in. Call uses the context of the object that called the handler.

```
on showCard
  answer the number of this card  --send returns card of handler, call returns card of calling object.
end showCard
```

Writing Reusable Code Using A Behavior Button

A behavior button is a method to create common functionality between objects without duplicating the script. Objects with a behavior set will use the script of the button as if it was their own script. Twenty buttons with similar function could all use one script in a behavior button. This makes it easier to edit the script. If multiple objects share the same behavior, each can have its own set of script local variables. Any references to me and the owner of me will resolve to the child object currently executing. Set an object's behavior in its properties inspector, or set the behavior by code using the long id of the button containing the behavior script.

```
set the behavior of button 2 to the long id of button 1  --button 1 is the behavior button
```

5.14) Timer Based Messaging

Use timers to schedule events to happen in the future. Use timers for updating the display at regular intervals, processing data in chunks, playing animations, displaying status bars, or to schedule events. Messages can be scheduled with millisecond precision and fire many times a second to create an animation, or scheduled to arrive hours later. All user events continue as normal. This makes timer based messaging ideal for a responsive user interface while data processing or updating the display.

Delivering A Message In The Future

To deliver a message after a specified time period, use the in time form of the send command.

```
send "updateStatus" to me in 20 seconds
send "updateAnimation" to me in 33 milliseconds
```

Repeating A Timer Message

To send a message repeatedly is more complicated and requires some planning. For example, to continuously draw frames in an animation, send the same message again at the end of the message handler.

```
on mouseUp
  updateAnimation
end mouseUp

on updateAnimation
  --insert code to update animation
  --insert code to exit message loop
  send updateAnimation to me in 33 milliseconds
end updateAnimation
```

Animation starts when the button is clicked, then updates the frame every 33 milliseconds (30 frames per second). The message will continue in a loop indefinitely. It is important to ensure there is a condition in the message handler to exit when the task is done, stopping the message loop. An alternative is to directly cancel the message to stop the loop.

Tip: To create a smooth script driven animation, allow for the time it takes to redraw the display and for any interruptions. Send the message a little more frequently than needed to redraw the screen. Check the current frame each time the message is activated to exit the message loop.

```
local sTotalFrames, sStart  --script local variables available to all handlers below
on mouseUp
  put the milliseconds into sStart  --start time
  put "100" into sTotalFrames  --number of frames in animation
  updateAnimation
end mouseUp

on updateAnimation
  local tElapsedTime, tFrameNumber
  put the milliseconds - sStart into tElapsedTime  --milliseconds since start
  put round(tElapsedTime / 33) into tFrameNumber
  if tFrameNumber > sTotalFrames then  --current frame number
    set the currentFrame of image 1 to sTotalFrames  --draw last frame
    exit updateAnimation  --exit message loop
  end if
  --insert code for drawing the screen
  set the currentFrame of image 1 to tFrameNumber  --animated gif
  send updateAnimation to me in 22 milliseconds  --continue message loop
end updateAnimation
```

Canceling A Timer Message

When a timer message is sent, the result function contains the ID of the message generated. Use the cancel command to cancel that message, preventing it from being delivered. In the following example, a dialog will be displayed 10 seconds after the user moves the mouse over a button. If the mouse moves out of the button before the 10 seconds has elapsed, the dialog will not be displayed.

```
local sTimerID  --script local available to all handlers below
on mouseEnter
  send "displayDialog" to me in 10 seconds
  put the result into sTimerID
end mouseEnter

on mouseLeave
  cancel sTimerID
end mouseLeave

on displayDialog
  answer "The mouse was over this button for 10 seconds." with "OK" titled "Timer Test"
end displayDialog
```

Important: Ensure a way to cancel any pending timer messages. Typically ensure a timer is canceled when the card is closed. For example, a timer message that draws an animation on the current card will generate a script error if the card is changed and the message is still sent, as the script will no longer be able to find the objects.

Displaying A List Of Pending Timer Messages

Get a list of all the currently pending timer based messages using the pendingMessages function.

```
put the pendingMessages into tMessagesList
```

tMessagesList will now contain a list of messages, one per line. Each line consists of four items, separated by commas: message ID, time the message is scheduled for, message name, the long ID property of the object the message will be sent to.

Tip: Get a list of all pending messages using the Message Box's pending messages tab. See pendingMessages in the Dictionary.

Tip: To cancel all currently pending messages, use the following repeat loop:
repeat for each line i in the pendingMessages
  cancel (item 1 of i)
end repeat

5.15) Tips For Writing Good Code
It's worth taking time to establish some conventions in how you write code.
o Consistently named variables are easier to debug when identified as local, script local, global....
o Comments make it easier to understand code written six months ago.
o Appropriate use of functions and libraries make a program modular and easier to modify later.
o Use the appropriate variable for the task. Don't use a global when a script local or handler local will do.
o Consider explicit variables (in Script Editor, Script > Variable Checking) which requires declaring variables and quoting literal strings.
o Read other people's code for ideas to improve your coding skills.
o Develop a consistent coding style and stick to it.

For an in-depth look at this subject, read Fourth World's Scripting Style Guide at:
http://www.fourthworld.com/embassy/articles/scriptstyle.html
==========

CHAPTER 6) PROCESSING TEXT AND DATA

Chunk Expressions, Character, Word, Item, Line, Compare & Search, Regular Expressions (Regex), Filter, Unicode, Array, Encode & Decode, Styled Text, URL, Binary, Encryption, Sorting.

Process text and data with unique chunk expressions using English-like statements like "word 3 to 5 of line 7 of field 1", and other powerful features including regular expressions, XML processing, associative arrays, data encoding and decoding functions, and compression and encryption algorithms.

6.1 Using Chunk Expressions
Chunk expressions are the primary method of working with text. A chunk is an English-like way of describing an exact portion of text. Use chunks to retrieve a portion of text and to edit text. This topic defines the types of chunks and the syntax for specifying them.

Types Of Chunks
The common types of chunks are the character, word, line, or item. An item can be delimited by any character(s) specified. In addition, the token chunk is useful when parsing script data. Here is an example of a chunk expression using the word chunk:
put word 1 to 3 of line 1 of field "Text" into myVariable

Using Chunks With Containers
Use a chunk of a container anywhere an entire container is used. Use chunk expressions with the put command to replace or with the delete command to remove any portion of a container.
add 1 to word 3 of field "Numbers"
put "Hello" into line 2 of field 1
delete character -1 of tList

Using Chunks With Properties
Use chunk expressions to read portions of a property, such as the script property, with the put command. However to change a property, first put the property value into a variable, use the chunk expression to change the variable, then use the set command to set the property to the variable's contents. The following example shows how to change the third line of an object's script property:

```
on mouseUp
  put the script of me into tScript
  put "--Last changed by Jane" && date() into line 3 of tScript
  set the script of me to tScript
end mouseUp
```

The Character Chunk

A character is a single character, which may be a letter, digit, punctuation mark, or control character. Use the abbreviation char as a synonym for character in a chunk expression. The character chunk corresponds to the notion of grapheme in the Unicode standard.

The Word Chunk

A word (segment) is a string of characters delimited by space, tab, or return characters, or enclosed by double quotes.

The Item Chunk And The ItemDelimiter Property

By default, an item is a string of characters delimited by commas. Items are delimited by a string specified in the itemDelimiter property. Change the default comma to create your own chunk type by setting the itemDelimiter property to any string.

The Line Chunk And The LineDelimiter Property

By default, a line is a string of characters delimited by the return character. Lines are delimited by a string specified in the lineDelimiter property. Change the default return to create your own chunk type by setting the lineDelimiter property to any string.

Other Chunks

There are also some chunks which are useful for more advanced text processing. These are paragraph, sentence, trueWord, codepoint, codeunit, byte, and token. See chunk types in the Dictionary.

The paragraph chunk is currently identical to the existing line chunk, however in the future paragraph chunks will also be delimited by the Unicode paragraph separator.

The sentence and trueWord chunk expressions facilitate the processing of text, taking into account the different character sets and conventions used by various languages. They use the ICU library, which uses a large database of rules for its boundary analysis, to determine sentence and word breaks.

The codepoint chunk type allows access to the sequence of Unicode codepoints which make up the string. The codeunit chunk type allows direct access to the UTF-16 code-units which notionally make up the internal storage of strings. The codeunit and codepoint chunk are the same if a string only contains unicode codepoints from the Basic Multilingual Plane.

The byte chunk is an 8-bit unit and should be used when processing binary data.

A token is a string of characters delimited by certain punctuation marks. The token chunk is useful in parsing statements, and is generally used only for analyzing scripts.

Specifying A Chunk

The simplest chunk expression specifies a single chunk of any type. The following statements all include valid chunk expressions:

```
get char 2 of "ABC"  --B
get word 4 of "This is a test"  --test
get line 7 of myTestData
put "A" into char 2 of myVariable
```

Use the ordinal numbers first, last, middle, second, third, fourth, fifth, sixth, seventh, eighth, ninth, and tenth to designate single chunks. The special ordinal any specifies a random chunk.

```
put "7" into last char of "1085"  --1087
put "7" into any char of "1085"  --7085, random
```

Negative Indexes In Chunk Expressions
To count backwards from the end of the value instead of forward from the beginning, specify a negative number. For example, the number -1 specifies the last chunk of the specified type, -2 specifies the next-to-last chunk, and so forth. The following statements all include valid chunk expressions:
get item -1 of "feather,ball,cap" --cap
get char -3 of "ABCD" --B

Complex Chunk Expressions
More complex chunk expressions can be constructed by specifying a chunk within another chunk. For example, the chunk expression word 4 of line 250 specifies the fourth word of line 250. When constructing a complex chunk expression, specify the chunk types in order: char, word, item, line. The full hierarchy is: byte > codeunit > codepoint > character > token > trueWord > word > item > line > sentence > paragraph.

The following statements are valid chunk expressions:
get char 7 of word 3 of line 8 of myValue
get word 9 of item 2 of myValue
get last char of word 8 of line 4 of myValue

These are not valid chunk expressions:
get word 8 of char 2 of myValue --chars can't contain words
get item 9 of first word of myValue --words can't contain items
get line 3 of last item of myValue --items can't contain lines
---
Using Parentheses In Chunk Expressions
Use parentheses in chunk expressions for the same reasons they're used in arithmetic, to make a complex expression clearer and to change the order in which the parts of the expression are evaluated. For example, in the following statement:
put item 2 of word 3 of "a,b,c i,j,k x,y,z" --won't work, words can't contain items
put item 2 of (word 3 of "a,b,c i,j,k x,y,z") --works, word 3 is "x,y,z" and item 2 of "x,y,x" is "y".

By adding parentheses around (word 3 of "a,b,c i,j,k x,y,z"), that part of the chunk expression is evaluated first. As with arithmetic expressions, the parts of a chunk expression that are in parentheses are evaluated first. If parentheses are nested, the chunk expression within the innermost set of parentheses is evaluated first. The part that is enclosed in parentheses must be a valid chunk expression, as well as being part of a larger chunk expression:
put line 2 of word 1 to 15 of myValue --won't work
put line 2 of word (1 to 15 of myValue) --won't work
put line 2 of word 1 to 15 (of myValue) --won't work
put line 2 of (word 1 to 15 of myValue) --works!

Nonexistent Chunks
If requesting a chunk number that doesn't exist, the chunk expression evaluates to empty.
get char 7 of "AB" --yields empty

If attempting to change a chunk that doesn't exist, it depends on what kind of chunk is specified:
o Nonexistent character or word: Putting text into a character or word that doesn't exist appends the text to the end of the container, without inserting any extra spaces.
o Nonexistent item: Putting text into an item that doesn't exist adds enough itemDelimiter characters to bring the specified item into existence.
o Nonexistent line: Putting text into a line that doesn't exist adds enough lineDelimiter characters to bring the specified line into existence.

Specifying A Range
To specify a portion larger than a single chunk, specify the beginning and end of the range. These are all valid chunk expressions:
get char 1 to 3 of "ABCD" --ABC
get word 2 to -1 of myValue --second word to last word

put it into line 7 to 21 of myValue --replace

The start and end of the range must be specified as the same chunk type, and the beginning of the range must occur earlier in the value than the end. Char -4 comes before char -1. The following are not valid chunk expressions:
char 3 to 1 of myValue --won't work, use char 1 to 3 or char 3 to -1
char -1 to -4 of myValue -- won't work, use char 1 to -4 or char -4 to -1

Counting The Number Of Words, Lines, or Items
The number function returns the number of chunks of a given type in a value. For example, to find the number of lines in a variable:
get the number of lines in myVariable
get the number of chars of item 10 in myVariable

6.2) Compare And Search
There are a number of ways to compare and search text. Using chunk expressions are easy and convenient. For complex searching use Regular Expressions.

Checking If A Part Is Within A Whole
Use the is in operator to check if text or data is within another piece of text or data. Use the reverse is not in operator to check if text or data is not within another piece of text or data.
"A" is in "ABC" --true
"123" is in "13" --false
"123" is not in "13" --true

Use the is in operator to check whether text or data is within a specified chunk of another container.
"A" is in item 1 of "A,B,C" --true
"123" is in word 2 of "123 456 789" --false
"123" is not in word 2 of "123 456 789" --true

Case Sensitivity
Comparisons are case insensitive by default (except for Regular Expressions, which have their own syntax for specifying whether or not a match should be case sensitive). To make a comparison case sensitive, first set the caseSensitive property to true. The caseSensitive property returns to default false after the handler finishes. See caseSensitive in the Dictionary.
set the caseSensitive to true
"A" is in "A,B,C" --true
"a" is in "A,B,C" --false

Check If Text Is True, False, Number, Integer, Point, Rectangle, Date, Or Color
Use the is a operator to check whether the user has entered data correctly and for validating parameters before sending them to a handler. The is an operator is equivalent to the is a operator.

A value is a:
o boolean if it is true or false.
o integer if it consists of digits with optional leading minus sign.
o number if it consists of digits, optional leading minus sign, optional decimal point, and optional "E" or "e" (scientific notation).
o point if it consists of two numbers separated by a comma.
o rectangle if it consists of four numbers separated by commas.
o date if it is in one of the formats produced by the date or time functions.
o color if it is a valid color reference.

The text being checked can contain leading or trailing white space characters in all the types except boolean. The is a operator is the logical inverse of the is not a operator. When one is true, the other is false. For example:
" true" is true --false, leading space
"1/16/98" is a date --true

1 is a boolean --false
45.4 is an integer --false, but is a number
"red" is a color --true

Tip: To restrict a user to typing numbers in a field, use the following script
on keyDown pKey
  if pKey is a number then pass keyDown
end keyDown

Check If A Word, Item, or Line Matches Exactly
The is among operator tells whether a whole chunk exists exactly within in a larger container. For example, to find if the whole word "free" is within a larger string, use the is among operator:
"free" is among the words of "Live free or die" --true
"free" is among the words of "Unfree world" --false, free is not an entire word
"free" is in "Unfree world" --true

Check If One String Starts Or Ends With Another
To check if one string begins with or ends with another string, use the begins with or ends with binary operators. For example:
"foobar" begins with "foo" --true
"foobar" ends with "bar" --true
line 5 of tList begins with "the"

Replace Text
To replace one string with another, use the replace command. To search a string with a regular expression, use the replaceText command.
replace "A" with "N" in thisVariable --change all A's to N's

To delete text using replace, replace a string with the empty constant. See replace and replace in field in the Dictionary.
replace return with empty in field 1 --run lines together

To preserve styling when replacing one string with another in a field, use the preserving styles clause.
replace "foo" with "bar" in field "myStyledField" preserving styles --replace "foo" with "bar", "bar" will retain the text styling of "foo"

Retrieve The Position Of A Matching Chunk
The offset, itemOffset, tokenOffset, trueWordOffset, wordOffset, lineOffset, sentenceOffset, and paragraphOffset functions locate the position of chunks within a larger container. To check if an item, line or word matches exactly using offset, set the wholeMatches property to true.
get offset("C","ABC") --3, "C" is the third char in "ABC"
put lineOffset(tMonth,monthNames()) into tMonthNumber --convert full month name to number, January = 1

Chunks Summary
o A chunk expression describes the location of a piece of text in a longer string.
o Chunk expressions can describe characters, items, tokens, trueWords, segments, lines, sentences, and paragraphs of text.
o To count backward from the end of a string, use negative numbers: word-2 = second-to-last word.
o Combine chunk expressions to specify one chunk contained in another chunk: word 2 of line 3 of myVariable.
o For a range of chunks, specify the start and end points of the range: line 2 to 5 of myVariable.
o To check if a chunk is within another, use the is in operator.
o To check if a chunk is a specified type of data, use the is a operator.
o To check if a chunk starts or ends with another uses the begins with or ends with operators.
o To check if a chunk is contained exactly within a string use the is among operator.
o To get an index specifying where a chunk can be found in a container, use the offset functions.
o To match only a complete chunk within a string, set the wholeMatches to true before using the offset functions.

6.3) Regular Expressions
Regular expressions check if a pattern is contained within a string. Use regular expressions to search for a pattern, replace a pattern, or filter the lines in a container depending on whether or not each line contains the pattern. RegEx is case sensitive. Regular expressions use PERL compatible or "PCRE" syntax. For more details on the supported syntax, see the PCRE manual at http://www.pcre.org/man.txt
---
Search Using A Regular Expression
Use the matchText function to check whether a string contains a specified pattern.
matchText(string,regularExpression[,foundTextVarsList])

The string is any expression that evaluates to a string. The regularExpression is any expression that evaluates to a regular expression. The optional foundTextVarsList consists of one or more names of existing variables, separated by commas. See matchText in the Dictionary.
matchText("Goodbye","bye") --true, lowerCase "bye" is in "Goodbye"
matchText("Goodbye","Bye") --false, upperCase "Bye" is not in "Goodbye"
matchText("Goodbye","^Good") --true
matchText (tPhoneNumber,"([0-9]+)-([0-9]+-[0-9]+)",areaCode,phone)

To retrieve the positions of the matched substrings in the optional foundTextVarsList, use the matchChunk function instead of the matchText function. These functions are otherwise identical.

Replace Using A Regular Expression
Use the replaceText function to search for a regular expression and replace the portions that match. To replace text without using a regular expression, use the replace command instead.
replaceText(stringToChange,matchExpression,replacementString)

The stringToChange is any expression that evaluates to a string. The matchExpression is a regular expression. The replacementString is any expression that evaluates to a string. See replaceText in the Dictionary.
replaceText("malformed","mal","well") --wellformed
replaceText(field "Stats",return,comma) --make line list a comma-delimited list
replaceText("abc35","[0-9]","") --abc, replace numbers with empty
replaceText("abc35","[^0-9]","") --35, replace all except numbers with empty
---
RegEx Expression Syntax
[chars] --match any one of the chars inside the brackets. A[BC]D matches "ACD".
[^chars] --match any single char not inside the brackets. A[^BC]D does not match "ACD".
[char-char] --match the range from first char to second char. [0-9] matches any number, (char1 ASCII value < char2 ASCII value).
. --match any single char (except a linefeed). A.C matches "ABC" or "AGC", but not "AC" or "ABDC".
^ --match the following char at the beginning of the string. ^A matches "ABC" but not "CAB".
$ --match the preceding char at the end of a string. B$ matches "CAB" but not "BBC".
* --match zero or more occurrences of the preceding char or pattern. ZA*B matches "ZB" or "ZAB" or "ZAAB", but not "ZXA" or "AB".
+ --match one or more occurrences of the preceding char or pattern. ZA+B matches "ZAB" or "ZAAB", but not "ZB" or "ZXA" or "AB".
? --match zero or one occurrence of the preceding char or pattern. ZA?B matches "ZB" or "ZAB", but not "ZAAB".
**** --match either the pattern before or the pattern after the \. A\B matches "A" or "B". [ABC]\[XYZ] matches "AY" or "CX", but not "AA" or "ZB".
\ --match the following char literally, even in a regular expression. A\.C matches "A.C", but not "A\.C" or "ABC".
any other char --match itself. ABC matches "ABC".

Filter Using A Wildcard Expression
Use the filter command to remove lines or items in a container that do or do not match a pattern. See filter in the Dictionary.
filter [lines of | items of] container {with | without | [not] matching} [wildcard pattern | regex pattern] filterPattern [into targetContainer]

The container is any expression that evaluates to a container. If lines or items are not specified, the all lines are filtered. The with and matching forms retain the matching lines or items. The without and not matching forms discard the matching lines or items. The wildcard, regex, and filter patterns are used to match certain lines. If no targetContainer is specified, the container contents is replaced with the filtered result.
filter lines of tList without empty --delete empty lines
filter lines of tList without "gray*" --delete all lines beginning with "gray..."
filter items of tList where each begins with "b" --keep only items beginning with "b"
filter items of field 1 with regex pattern "b.*" --keep only items containing "b"
filter lines of tList not matching "foo*" into tFilteredList
filter field "MyLines" matching wildcard pattern "[" & tMySelection & "]"
filter myVariable without "*[a-zA-Z]*"

6.4) Unicode And International Text Support
All text processing capabilities extend seamlessly to international text. This includes the ability to render and edit Unicode text and convert between various encoding types.

What Are Text Encodings?
Computers use numbers to store information, converting those numbers to text to be displayed on the screen. A text encoding describes which number converts to a given character. There are many different encoding systems for different languages. The following are examples of common encodings.

o ASCII, Single byte - English. ASCII is a 7-bit encoding, using one byte per character. It includes the full Roman alphabet, Arabic numerals, Western punctuation and control characters. See http://en.wikipedia.org/wiki/ASCII for more information.

o ISO8859, Single byte. ISO8859 is a collection of 10 encodings. They are all 8-bit, using one byte per character. Each shares the first 128 ASCII characters. The upper 80 characters change depending on the language to be displayed. For example ISO8859-1 is used in Western Europe, whereas ISO8859-5 is used for Cyrillic. NB: only ISO8859-1 is supported. Use Unicode to represent other languages, converting if necessary (see below).

o Windows 1252, Single byte - English. This is a super-set of ISO8859-1 which uses the remaining 48 characters not used in the ISO character set to display
characters on Windows systems. For example curly quotes are contained within this range.

o MacRoman, Single byte - English. MacRoman is a super-set of ASCII. The first 128 characters are the same. The upper 128 characters are entirely rearranged and bear no relation to either Windows-1252 or ISO8859-1. However while many of the symbols are in different positions many are equivalent so it is possible to convert between the two.

o UTF-16, Double byte. Any UTF-16 typically uses two bytes per code point (character) to display text in all the world's languages (see Introduction to
Unicode, below). UTF-16 will take more memory per character than a single-byte encoding and so is less efficient for displaying English.

o UTF-8, Multi-byte. Any UTF-8 is a multi-byte encoding. It has the advantage that ASCII is preserved. When displaying other languages, UTF-8 combines
together multiple bytes to define each code point (character). The efficiency of UTF-8 depends on the language you are trying to display. If displaying Western European it will take (on average) 1.3 bytes per character, for Russian 2 bytes (equivalent to UTF-16) but for CJK 3-4 bytes per character.

What Are Scripts?
A script is a way of writing a language. It takes the encoding of a language and combines it with its alphabet to render it on screen as a sequence of glyphs. The same language can sometimes be written with more than one script (common among languages in India). Scripts can often be used to write more than one language (common among European languages).

Scripts can be grouped together into four approximate classes. The "small" script class contains a small alphabet with a small set

of glyphs to represent each single character. The "large" script class contains a large alphabet with a larger set of glyphs. The "contextual" script class contains characters that can change appearance depending on their context. And finally the "complex" script class contains characters that are a complex function of the context of the character - there isn't a 1 to 1 mapping between code point and glyph.

o Roman, Small script. The Roman encoding has relatively few distinct characters. Each character has a single way of being written. It is written from left to right, top to bottom. Every character has a unique glyph. Characters do not join when written. For example: The quick brown fox.

o Chinese, Large script. The Chinese encoding has a large number of distinct characters. Each character has a single way of being written.

o Greek, Contextual script. Every character except sigma has a unique glyph. Sigma changes depending on whether it is at the end of a word or not. Characters do not join when written. The text runs left to right, top to bottom.

o Arabic, Contextual script. The glyph chosen is dependent on its position in a word. All characters have initial, medial and terminal glyphs. This results in a calligraphic (joined up) style of display. The text runs right to left, top to bottom.

o Devanagari, Complex script. In this script there is no direct mapping from character to glyph. Sequences of glyphs combine depending on their context. The text runs from left to right, top to bottom.

Introduction To Unicode

Traditionally, computer systems have stored text as 8-bit bytes, with each byte representing a single character (for example, the letter 'A' might be stored as 65). This has the advantage of being very simple and space efficient while providing 256 different values to represent all the symbols that might be provided on a typewriter. The flaw in this scheme becomes obvious fairly quickly: there are far more than 256 different characters in use in all the writing systems of the world, especially when East Asian ideographic languages are considered. But, in the pre-internet days, this was not a big problem.

This product was first created before the rise of the internet, and adopted the 8-bit character sets of the platforms it ran on (which also meant that each platform used a different character set: MacRoman on Apple devices, CP1252 on Windows, and ISO-8859-1 on Linux and Solaris). These character encodings are "native" encodings.

In order to overcome the limitations of 8-bit character sets, the Unicode Consortium was formed. This group aims to assign a unique numerical value ("codepoint") to each symbol used in every written language in use (and in a number that are no longer used!). Unfortunately, this means that a single byte cannot represent any possible character.

The solution is to use multiple bytes to encode Unicode characters and there are a number of schemes for doing so. Some of these schemes can be quite complex, requiring a varying number of bytes for each character, depending on its codepoint.

The Engine was modified to handle Unicode text transparently throughout. All standard text manipulation operations work on Unicode text without any additional effort on your part. Unicode text can be used in menus and to name controls, stacks, and other objects. Anywhere text is used, Unicode should work.

Creating Unicode Apps

Creating stacks that support Unicode is no more difficult than creating any other stack but there are a few important differences. In previous versions, text and binary data could be used interchangeably; doing this with Unicode may not work as expected.

When text is treated as binary data (written to a file, process, socket, or other object outside of the Engine) it will lose its Unicode-ness: it will automatically be converted into the platform's 8-bit native character set. Any Unicode characters that cannot be correctly represented will be converted into question mark '?' characters.

Similarly, treating binary data as text will interpret it as native text and won't support Unicode. To avoid this loss of data, text should be explicitly encoded into binary data and decoded from binary data at these boundaries. This is done using the textEncode and textDecode functions (or their equivalents, such as opening a file using a specific encoding).

Unfortunately, the correct text encoding depends on the other programs that will be processing the data and cannot be automatically detected by the Engine. If in doubt, UTF-8 is often a good choice as it is widely supported by a number of text processing tools and is sometimes considered to be the "default" Unicode encoding.

The following are examples of the textEncode and textDecode functions for exporting and importing text that include unicode:

```
put specialFolderPath("resources") &"/Output.txt" into tPath  --stack folder
put tOutput into url ("file:" &tPath)  --export all-English text
put url ("file:" &tPath) into tInput  --import all-English text

put textEncode(tOutput,"utf8") into tEncoded
put tEncoded into url ("binfile:" &tPath)  --export any text including unicode

put url ("binfile:" &tPath) into tEncoded
put textDecode(tEncoded,"utf8") into tInput  --import any text including unicode
```

Things To Look Out For

o With binary data, use byte rather than char. The char chunk expression is intended for use with text data and represents a single graphical character rather than an 8-bit unit.

o Avoid hard-coding assumptions based on your native language when formatting numbers or the direction for text layout.

o Regardless of visual direction, text is always in logical order. Word 1 is always the first word whether it appears at the left or the right.

o Even English text can contain Unicode characters like curly quotation marks, long and short dashes, accents, and currency symbols.

6.5) Using Arrays

When To Use Arrays

Each element in an array can be accessed in constant time. This makes searching an array much faster than functions like the offset that look up information by counting through a variable from beginning to end.

Each element in an array can contain data of any length, making it easier to group together records that contain assorted lengths or delimiter characters. Arrays can contain nested elements. This makes them ideal for representing complex data structures such as trees and XML data that would be hard to represent as a flat structure. Each sub-array in an array can be accessed and operated on independently. This makes it possible to copy a sub-array to another array, get the keys of a sub-array, or pass a subarray as a parameter in a function call. There are functions for converting information to and from arrays, and for performing operations on the contents of arrays.

These characteristics make arrays useful for a number of data processing tasks, including tasks that involve processing or comparing large amounts of data. Arrays are ideal to count the number of instances of a specific word in a piece of text. A word count iterating through each word and checking if it is present in a list of words gets slower as the list of words gets longer. A word count using an array is much faster, where each element in an array is named using a text string.

```
on mouseUp
  repeat for each word i in field "Sample Text"  --cycle through each word adding each instance to an array
    add 1 to tWordCount[i]
  end repeat
  combine tWordCount using return and comma  --array to text
  sort tWordCount numeric descending by item 2 of each  --hi-low
  answer tWordCount
end mouseUp
```

Text in field "Sample Text:

Single Line – execute single line and short scripts

Multiple Lines – execute multiple line scripts

Global Properties – view and edit global properties
Global Variables – view and edit global variables

Resulting value of tWordCount:
Global,4
–,4
Line,3
and,3
Multiple,2
Properties,2
Single,2
Variables,2
edit,2
execute,2
scripts,2
view,2
Lines,1
short,1

Array Functions
The following is a list of all the syntax that works with arrays. Each of these functions can be used on subarrays within an array. Instead of referring to the arrayvariable, refer to x[x]. For a full description see the corresponding entry in the Dictionary.

o add: add a value to every element in an array where the element is a number.
o combine: convert an array into text, placing delimiters you specify between the elements.
o customProperty: return an array of the custom properties of an object.
o delete variable: remove an element from an array.
o divide: divide each element in an array where the element is a number. For example, divide tArray by 3
o element: keyword used in a repeat loop to loop through every element in an array.
o extents: find the minimum and maximum row and column numbers of an array whose keys are integers.
o intersect: compare arrays, removing elements from one array if they have no corresponding element in the other.
o keys: return a list of all the elements within an array.
o matrixMultiply: perform a matrix multiplication on two arrays whose elements are numbers and whose keys are sequential numbers.
o multiply: multiply a value in every element in an array where the element is a number.
o properties: return an array of the properties of an object.
o split: convert text to an array using delimiters that you define.
o sum: return the sum total of all the elements in an array where the element is a number.
o transpose: swap the order of the keys in each element in an array whose elements are numbers and whose keys are sequential numbers.
o union: combine two arrays, eliminating duplicate elements.

6.6) Encode & Decode

Styled Text
There are a number of built-in functions for encoding and decoding styled text in a variety of popular formats.

Encode and decode styled text as HTML and RTF. This feature is useful to adjust text styles programmatically, or import or export text with style information. HTML conversion support only extends to the styles that the field object is capable of rendering. See htmlText in the Dictionary.

To convert the contents of a field to HTML compatible tags, use the htmlText property. You can also set this property to display styled text in a field.

Get and set the htmlText property of a chunk within a field to view or change text attributes on a section of the text. For example, to set the text style of line 2 of a field to bold:

```
on mouseUp
  local tText
  put the htmlText of line 2 of field "Sample Text" into tText
  replace "<p>" with "<p><b>" in tText  --bold tag
  replace "</p>" with "</b></p>>" in tText  --end bold tag
  set the htmlText of line 2 of field "Sample Text" to tText
end mouseUp
```

Tip: While this is not as simple as directly applying the style to the text using:
set the textStyle of line 2 of field "Sample Text" to "bold"

It does allow search and replace, and multiple complex changes to the text based on pattern matching. Performing a series of changes on the HTMLText in a variable then setting the text of a field once is faster than updating the style repeatedly directly on the field.

Use the HTML keyword with the Drag and Drop features and the Clipboard features to perform conversion of data to and from HTML when exchanging data with other applications.

Use the rtfText property and RTF keyword to work with the RTF format. Use the unicodeText property and Unicode keyword to work with Unicode.

URL

To encode and decode URLs, use the urlEncode and urlDecode functions. The urlEncode function will make text safe to use with a URL. For example it will replace space with +. These functions are particularly useful posting data to a web form using the POST command, using the launch URL command or sending email using the revMail function. For more info see the Dictionary.

Text: "Jo Bloggs" <jo@blogs.com>
urlEncode function result: %22Jo+Bloggs%22+%3Cjo%40blogs.com%3E

Binary Data - Base64 (for MIME Email Attachments and Https Transfers)

To encode and decode data in Base64 (to create an email attachment), use the base64Encode and base64Decode functions. These functions are useful to convert binary data to text data and back. For more info see the Dictionary.

Binary Data - Arbitrary Types

Use the binaryEncode and binaryDecode functions to encode or decode binary data. For more info see the Dictionary.

Character To Number Conversion

To convert a character to and from its corresponding ASCII value use the codePointToNum and numToCodePoint functions.

```
put codePointToNum("Z")  --90, quotes optional, A-Z=65-90, a-z=97-122
put numToCodePoint("90")  --Z, quotes optional, enter decimal or hexadecimal number ("0x" prefix for hexadecimal)
```

Data Compression

To compress and decompress data using GZIP, use the compress and decompress functions. The following routine asks the user to select a file, then creates a GZip compressed version with a ".gz" extension in the same directory as the original.

```
on mouseUp
  local tFileCompressed
  answer file "Select a file:"
  if it is empty then exit mouseUp
  put it & ".gz" into tFileCompressed  --file path with extension
  put compress(URL ("binfile:" & it)) into URL ("binfile:" & tFileCompressed)  --compressed file next to original file
end mouseUp
```

Encryption
To encrypt or decrypt data use the encrypt and decrypt commands. See their documentation in the Dictionary.

Generating A Checksum
Use the MD5Digest function to generate a digest of some data. Use this function later to determine if the data was changed or to check that transmission of information was complete. In this example, save the MD5Digest of a field as a custom property when the user opens it for editing. In the field script:

```
on openField
  set the cDigest of me to md5Digest(the htmlText of me)  --cDigest is a custom property
end openField
```

If the field is modified (including if a text style is changed anywhere) then a subsequent check of the MD5Digest will return a different result. In the next example, check this digest to determine whether or not to bring up a dialog alerting the user to save changes:

```
on closeStackRequest
  if the cDigest of field "Sample Text" is not md5Digest(the htmlText of field "Sample Text") then
    answer "Save changes before closing?" with "No" or "Yes"
    if it is "Yes" then save this stack
  end if
end closeStackRequest
```

6.7) Sorting Data
Sorting data is a common and fundamental operation. Sort to display data in a user-friendly fashion or code a number of algorithms. The intuitive sort command has the power and flexibility to perform any kind of sorting required.

The Sort Container Command: Overview
To sort data, use the sort container command.
sort [{lines | items} of] container [direction] [sortType] [by sortKey]

o The container is a field, button, or variable, or the message box.
o The direction is either ascending or descending. If a direction is not specified, the default sort is ascending.
o The sortType is one of: text, numeric, or dateTime. If a sortType is not specified, the default sortType is text.
o The sortKey is an expression that evaluates to a value for each line or item in the container. If the sortKey contains a chunk expression, the keyword "each" indicates that the chunk expression is evaluated for each line or item. If a sortKey is not specified, the entire line (or item) is used as the sortKey.

The following examples sort the lines of a variable alphabetically:

```
sort lines of field "Sample Text" ascending text  --a-z
sort lines of tText descending text  --z-a
```

The following example sorts a collection of items numerically:

```
sort items of field "Sample Numbers" ascending numeric  --low-high
sort items of tItems descending numeric  --high-low
```

Using Sort Keys
The sortKey syntax to sort each line or item based on the results of an evaluation performed on each line or item. To sort the lines of a container by a specific item in each line:

```
sort lines of tContainer by the first item of each  --each line
sort lines of tContainer by item 3 of each  --each line
```

The sortKey expression will only be evaluated once for every element that is to be sorted. This syntax allows a variety of more

complex sort operations to be performed. The following example will extract the minimum and maximum integers present in a list:

```
on mouseUp
  local tMinInteger, tMaxInteger
  set the itemDelimiter to "."
  sort lines of fld 1 numeric by char 2 to -1 of the first item of each  --low-high
  put char 2 to -1 of the first item of the first line of fld 1 into tMinInteger
  put char 2 to -1 of the first item of the last line of fld 1 into tMaxInteger
  answer "tMinInteger =" && tMinInteger &","&& "tMaxInteger =" && tMaxInteger with "OK" titled "Sort"
end mouseUp
```

Original list:
F54.mov
M27.mov
M7.mov
F3.mov

Result: tMinInteger = 3, tMaxInteger = 54

Sorting Randomly
To sort randomly, use the random function to generate a random number as the sortKey for each line or item, instead of evaluating the line or item's contents. For example:

```
on mouseUp
  local tExampleList, tCount
  put field 1 into tExampleList
  put the number of lines of tExampleList into tCount
  sort lines of tExampleList ascending numeric by random(tCount)
  put tExampleList into field 1  --randomized line list
end mouseUp
```

Stable Sorts - Sort Multiple Times
To sort a list by multiple criteria, sort multiple times. The sort command uses a stable sort, if the same sortKey is used their relative order in the output won't change. Begin with the least important criteria and work up to the most important. For example, sort fld 1 multiple times using the same sortKey "of each":

```
sort lines of fld 1 ascending numeric by item 2 of each
sort lines of fld 1 ascending text by item 1 of each
```

Original list:
Oliver,1.54
Elanor,5.67
Tim,8.99
Elanor,6.34
Oliver,8.99
Tim,3.44

Result of stable multiple sorting: item 2 was sorted numeric low-high, then item 1 was sorted text a-z.
Elanor,5.67
Elanor,6.34
Oliver,1.54
Oliver,8.99
Tim,3.44
Tim,9.99

Sorting Cards
To sort cards, use the sort cards command.
sort [marked] cards [of stack] [direction] [sortType] by sortKey

o The marked is a subset of cards with their mark property set to true.
o The stack is a reference to any open stack. If a stack is not specified, the cards of the current stack are sorted.
o The direction is either ascending or descending. If a direction is not specified, the sort is ascending.
o The sortType is one of text, international, numeric, or dateTime. If a sortType is not specified, the sort is by text.
o The sortKey is an expression that evaluates to a value for each card in the stack. Any object references within the sortKey are treated as pertaining to each card being evaluated, so for example, a reference to a field is evaluated according to that field's contents on each card. Typically the sort command is used with background fields that have their sharedText property set to false so that they contain a different value on each card.

To sort cards by the contents of field "Last Name" on each card:
sort cards by field "Last Name"

To sort cards by the numeric value in a field "ZIP Code" on each card:
sort cards numeric by field "ZIP Code"

Tip: To sort cards by a custom expression that performs a calculation, create a custom function:
```
sort cards by myFunction()  --custom function
function myFunction
  put the number of buttons of this card into tValue
  --perform any calculation on tValue here
  return tValue  --sort will use this value
end myFunction
```
==========

CHAPTER 7) PROGRAMMING A USER INTERFACE

Referring To Objects, Properties, Global Properties, Text Properties, Creating & Deleting Objects, Property Array, Custom Property, Custom Property Set, GetProp & SetProp Handler, Virtual Property, Managing Windows & Dialogs, Menus & Menu Bar, Find, Drag & Drop.

The user interface for an application is often one of its most important features. In this shows how to edit or even build a user interface by code. Everything done using the built-in tools can also be done by code. Even create and modify a user interface at run time in a standalone application, or provide interactive methods for users to modify specific aspects of the application.

This set of capabilities can produce applications that construct their own interface using XML files or a custom data structure, programmatically construct aspects of complex interfaces, modify their look using user specified parameters, create themed or skinned interface options, and build interface editing tools that plugin to the IDE. You can also create custom objects and attach your own virtual behaviors and custom properties to them. However, first become familiar with building an interface using the tools in the IDE before creating or editing an interface by code.

7.1) Referring To Objects
Refer to any object by its name, number, or ID property.

Referring To Objects By Name
Refer to an object using its object type followed by its name. For example, to refer to a button named "OK", use the phrase button "OK":
set the loc of button "OK" to "32,104"

To change an object's name, enter a name in the object's property inspector, or use the set command to change the object's name property:
set the name of field "Old Name" to "New Name"

select after text of field "New Name"

Referring To Objects By Number
An object's number is its layer on the card, from back to front. A card's number is its position in the stack. A stack's number is the order of its creation in the stack file. A main stack's number is always zero. Refer to an object using its object type followed by its number. For example, to refer to the third-from-the-back field on a card, use the phrase field 3:
set the backgroundColor of field 3 to blue

To change the number of a card or object, change the Layer box in the Position tab of the object's property inspector, or use the set command to change the object's layer property:
set the layer of field "Backmost" to "1"

Tip: New objects are always created at the top layer. To refer to an object you've just created, use the ordinal last:
set the name of last button to "My New Button"

Referring To Objects By ID
Each object has a unique ID number. The ID property never changes (except for stack IDs), and is guaranteed unique within the stack: no two objects in the same stack can have the same ID property.

Refer to an object using its object type, then keyword ID, followed by its ID number. For example, to refer to a card whose ID property is 1154, use the phrase card ID 1154:
go to card ID 1154

An object's ID property cannot be changed (except for a stack).

Important: Wherever possible, name objects and refer to them by name instead of number or ID. Both the number and ID properties will change if objects are copied, pasted, or deleted. Additionally, the scripts will become difficult to read if there are many ID or numerical references to objects.

Referring To Objects By Ordinal
Refer to an object using its object type followed by the ordinal numbers first through tenth, or the special ordinals middle and last. To refer to a random object, use the special ordinal any. For example, to refer to the last card in the current stack, use the special ordinal last:
go to last card

The Special Descriptor 'This'
Use the this keyword to indicate the current stack, or the current card of a stack:
set the backgroundColor of this stack to white
send "mouseUp" to this card
set the textFont of this card of stack "Menubar" to "Sans"

Control References
A control is any object that can appear on a card. Fields, buttons, scrollbars, images, graphics, players, EPS objects, and groups are all controls. Stacks, cards, audio clips, and video clips are not controls. You can refer to an object of any of these object types using the word "control", followed by an ID, name, or number:
hide control ID 2566
send mouseDown to control "My Button"
set the hilite of control 20 to "false"

Tip: referring to an object using the word "object" causes an error.
hide object ID 2566 --doesn't work

When referring to a control by name, the reference is to the first control (with the lowest layer) that has that name.

When referring to a control by number using its object type, the reference is to the Nth control of that type. For example, the

phrase field 1 refers to the lowest field on the card. This may not be the lowest control, because there may be objects of other types underneath field 1. However, the phrase control 1 refers to the lowest control of any type on the card.

Tip: To refer to the object underneath the mouse pointer, use the mouseControl function.
edit the script of the mouseControl

Nested Object References
To refer to an object that belongs to another object, nest the references in the same order as the object hierarchy. For example, if there is a button called "My Button" on a card called "My Card", refer to the button like this:
show button "My Button" of card "My Card"

Mix names, numbers, ordinal references, and IDs in a nested object reference, and nest references to whatever depth is required to specify the object. The only requirement is the order of references be the same as the order of the object hierarchy, going from an object to the object that owns it. Here are some examples:
field ID 34 of card "Holder"
player 2 of group "Main" of card ID 20 of stack "Demo"
first card of this stack
stack "Dialog" of stack "Main" --"Dialog" is a substack

If a card is not specified in referring to an object that is contained by a card, it is assumed the object is on the current card. If a stack is not specified, it is assumed the object is in the current stack. Reference a control in another stack by either of the following methods:

o Use a nested reference that includes the name of the stack:
field 1 of stack "My Stack"
graphic "Outline" of card "Tools" of stack "Some Stack"

o Set the defaultStack property to the stack you want to refer to first:

```
on mouseUp
  local tSavedDefault, tSetting
  put the defaultStack into tSavedDefault  --original stack
  set the defaultStack to "Other Stack"  --all object references now to open stack "Other Stack"
  put the hilite of button "Me" into tSetting  --this button is in "Other Stack"
  set the defaultStack to tSavedDefault  --all object references back to original stack
  set the hilite of button "Me Too" to tSetting  --this button is in the original stack
end mouseUp
```

If an object is in a group, include or omit a reference to the group in a nested reference to the object. For example, suppose the current card contains a button called "Guido", which is part of a group called "Stereotypes". Refer to the button with any of the following expressions:
button "Guido"
button "Guido" of card 5
button "Guido" of group "Stereotypes"
button "Guido" of group "Stereotypes" of card 5

If there are no other buttons named "Guido" on the card, these examples are equivalent. If there is another button with the same name in another group (or on the card, but not in any group), you must either specify the group (as in the third and fourth examples) or refer to the button by its ID property, to be sure you're referring to the correct button.

Avoid Using Numbers As Object Names
It is dangerous to set the name property of an object to a number.

For example, create three fields on a new card, naming the first one "3", the second one "2", and the third one "1". In other words, the creation order of those three fields is the reverse of the numerical names you've given them. Now use the message

box to:
put "This is field 3" into field 3 --field 3 named "1" displays text

The third field created, the one named "1", displays the text. The number of the field, not the name set by you, was used as the object reference. This shows the perils of using a number as the name of an object. The built-in properties of objects (the layer and the number) are based on integers, so to use integers in the name property will cause problems.

This applies to cards as well. The card order supersedes any numerical card name.

However, a name which includes a number can be used safely. For example, naming fields "MyText1", "MyText2", and "MyText3" have no conflict with the control number:
put "This is field 3" into field "MyText3"

To construct numerical field names:

```
on mouseUp
  repeat with i = 1 to 3
    put i into field ("MyText" & i)
  end repeat
end mouseUp
```

7.2) Properties
A property is an attribute of an object. Each type of object has many built-in properties, which affect the object's appearance or behavior. You can also define custom properties for any object, and use them to store any kind of data.

This topic discusses how to use properties, how properties are inherited between objects, and how to create and switch between collections of property settings. To fully understand this topic, know how to create objects, how to use an object's property inspector, and how to write short scripts.

Using Object Properties
A property is an attribute of an object, and each object type has its own set of built-in properties. Making all the build-in properties of two objects identical, they'd be the same object type. Thus it is possible to describe an object entirely as an array of properties, or to export and import properties using text or XML files.

Built-in properties determine the appearance and behavior of stacks and their contents - fonts, colors, window types, size, and placement - as well as much of the behavior of an application. By changing properties, almost any aspect of an app can be changed. Combining the ability to change properties with the ability to create and delete objects by code, you can modify every aspect of an application.

Referring To Properties
Property references consist of the word the, the property name, the word of, and a reference to the object:
the armedIcon of button "My Button"
the borderWidth of field ID 2394
the name of card 1 of stack "My Stack"

Properties are sources of value, so you can get the value of a property by using it in an expression:
put the height of field "Text" into myVar
put the width of image "My Image" + 17 after field "Values"
if item 1 of the location of me > zero then beep

For example, to use the width property of a button as part of an arithmetic expression:
add the width of button "Cancel" to totalWidths

The value of the property, the width of the button in pixels, is substituted for the property reference when the statement is

executed.

Tip: To see a list of properties filtered to a particular object type: Help > Dictionary, Object > Button, sort the button list by clicking Type header. To return back: All, sort the list by clicking Keyword header.

Changing Properties
To change the value of a property, use the set command:
set the borderColor of group "My Group" to "red"
set the top of image ID 3461 to "100"

View and change many of an object's properties by selecting the object and choosing Object > Object Inspector.

Most built-in properties affect the appearance or behavior of the object. For example, a button's height, width, and location are properties of the button. Changing these properties in a handler causes the button's appearance to change. Conversely, dragging or resizing the button changes the related properties.

Read-Only Properties
Some properties can be read with the get command, but not set. These are called read-only properties. Trying to set a read-only property causes an execution error. To find out whether a property is read-only, check its entry in the Dictionary.

Changing A Part Of A Property
Properties are not containers, so you cannot use a chunk expression to directly change a part of a property. Put the property into a variable container, use a chunk expression to change a part of the property, then set the property back with its new change:
put the rect of me into tempRect --original property in container
put "10" into item 2 of tempRect --change with chunk expression
set the rect of me to tempRect --changed property

Custom Properties And Virtual Properties
A custom property is a property that you define. Create as many custom properties for an object as you want, and put any kind of data into them, including binary data or array data. Even store a file in a custom property.

Virtual properties are custom properties that trigger a custom script action when changed, allowing "virtual" object behaviors.

Property Inheritance
Most properties are specific to an object, and affect only that object. However, some properties of an object, such as its color and text font, take on the settings of the object above it in the object hierarchy. For example, if a field's background color property is empty, the field takes on the background color of the card that owns it. If no background color is specified for the card either, the stack's background color is used, and so on. Set a background color for a stack, and every object in it will automatically use that background color, without having to set it for each object.

This process of first checking the object, then the object's owner, then the object that owns that object, and so on, is called inheritance of properties. Each object inherits the background color of the object above it in the hierarchy. Similar inheritance rules apply to the foregroundColor, topColor, bottomColor, borderColor, shadowColor, and focusColor properties, to their corresponding pattern properties, and to the textFont, textSize, and textStyle properties.

Overriding Inheritance
Inheritance is used to determine an object's appearance only if the object itself has no setting for the property. If an inheritable property of an object is not empty, that setting overrides any setting the object might inherit from an object above it in the object hierarchy. For example, if a button's backgroundColor property is set to a color instead of being empty, the button uses that background color, regardless of the button's owners. If the object has a color of its own, that color is always used.

The Effective Keyword
If an inheritable property of an object is empty, use the effective keyword to obtain the inherited setting of the property. The effective keyword searches the object's owners to find out what setting is actually used. For example, suppose a field whose textFont property is empty and the textFont of the card the field is on is set to "Helvetica". The field inherits this setting and

displays its text in the Helvetica font. To find out what font the field is using, use the effective keyword:
```
get the textFont of field "My Field"  --empty
get the effective textFont of field "My Field"  --Helvetica
```

Use the effective keyword with any inherited property.

7.3) Global Properties
Global properties affect the overall behavior of the application. Global properties are accessed and changed the same way as object properties. They do not belong to any particular object, but otherwise behave like object properties.

Tip: To see a list of all global properties, open the Message Box, and choose the Global Properties icon. To see a list of all properties, including both global and object properties, use the propertyNames function. Check the list to avoid using a reserved word for your own custom properties.
```
put the propertyNames into tAllProps
```

A few properties are both global and object properties. For example, the paintCompression is a global property, and also a property of images. For these properties, the global setting is separate from the setting for an individual object. Some other global properties are affected by system settings. For example, the default value of the playLoudness property is set by the operating system's sound volume setting.

Referring To Global Properties
Refer to global properties using the and the property name:
```
the defaultFolder
the emacsKeyBindings
the fileType
```

Since global properties apply to the whole application, an object reference is not needed. Global properties are sources of value, so you can get the value of a global property by using it in an expression:
```
get the stacksInUse
put the recentNames into field "Recent Cards"
if the ftpProxy is empty then exit setMyProxy
```

Changing Global Properties
To change a global property, use the set command in the same way as for object properties:
```
set the itemDelimiter to "/"
set the grid to false
set the idleTicks to "10"
```

Some global properties can be changed by other commands. For example, the lockScreen property can either be set directly, or changed using the lock screen and unlock screen commands. The following two statements are equivalent:
```
set the lockScreen to false  --global prop
unlock screen  --command
```

Saving And Restoring Global Properties
Object properties are part of an object, so they are saved when the stack is saved. Global properties are not associated with any object, so they are not saved with a stack. A change to the value of a global property is lost when the application is quit.

To use a changed setting of a global property during a different session of the application, save the global setting in a Preferences file, in a custom property, or elsewhere in a saved file and restore it each time the application starts up.

7.4) Text Related Properties
Normally, properties are applied only to objects, and global properties are applied only to the application. However, a few properties also apply to chunks in a field or to single characters in a field.

Text Style Properties
Certain text-related properties can be applied either to an entire field or to a chunk of a field:
set the textFont of word 3 of field "My Field" to "Courier"
set the foregroundColor of line 1 of field 2 to "green"
if the textStyle of the clickChunk is "bold" then beep

The following field properties can be applied to either an entire field or to a chunk of the field: textFont, textStyle, textSize, textShift, backgroundColor, foregroundColor, backgroundPattern (Windows & Linux), foregroundPattern (Windows & Linux).

Each chunk of a field inherits these properties from the field, in the same way that fields inherit from their owners. For example, if a word's textFont property is empty, the word is displayed in the field's font. But if you set the word's textFont to another font name, that word alone is displayed in its own font.

To find the text style of a chunk in a field, whether that chunk uses its own styles or inherits them from the field, use the effective keyword:
get the effective textFont of word 3 of field ID 2355
answer the effective backgroundColor of char 2 to 7 of field "My Field"

Tip: If a chunk expression includes more than one style, the corresponding property for that chunk reports "mixed". For example, if the first line of a field has a textSize of "12", and the second line has a textSize of "24":
put the textSize of line 1 to 2 of field "My Field" --mixed

Formatted Text Properties
The htmlText, RTFText, and unicodeText properties of a chunk are equal to the text of that chunk, along with the formatting information that's appropriate for the property. For example, if a field contains the text "This is a test.", and the word "is" is boldfaced, the htmlText of word 2 reports "is".

For more information on these properties see their individual entries in the Dictionary.

The FormattedRect And Related Properties
The formattedRect, formattedWidth, formattedHeight, formattedLeft, and formattedTop properties report the position of a chunk of text in a field. These properties are read-only.

The formattedRect, formattedLeft, and formattedTop properties can be used for a chunk of a field, but not the entire field. The formattedWidth and formattedHeight apply to both fields and chunks of text in a field.

The ImageSource, LinkText, And Visited Properties
The imageSource of a character specifies an image to be substituted for that character when the field is displayed. Use the imageSource to display images inside fields:
set the imageSource of char 17 of field 1 to 49232 --id of image
set the imageSource of char thisChar of field "My Field" to "https://www.example.com/banner.jpg"

The linkText property of a chunk lets you associate hidden text with part of a field's text. Use the linkText in a linkClicked handler to specify the destination of a hyperlink, or for any other purpose.

The visited property specifies whether the user clicked on a text group during the current session. Get the visited property for any chunk in a field, but it is meaningless unless the chunk's textStyle includes "link".

The imageSource, linkText, and visited properties are the only properties that can be set to a chunk of a field, but not to the entire field or any other object. Because they are applied to text in fields, they are listed as field properties in the Dictionary.

7.5) Creating And Deleting Objects
Create and delete objects by code, and optionally specify all the properties for a new object before creating it.

The Create Object Command
Use the create command to create a new object. See create in the Dictionary.
create [invisible] type [name] [in group]

o Type is any object that can be on a card: field, button, image, scrollbar, graphic, player, or EPS.
o Name is the name of the newly created object. If a name is not specified, the object is created with a default name.

o In Group is any group that's on the current card. If a group is specified, the new object becomes a member of the group, and exists on each card that has the group. If a group is not specified, the object is created on the current card and appears only on that card.
create button "Click Me"
create invisible field in first group --or last group

Use the create widget command to create widgets of a given kind on a card. See create widget in the Dictionary.
create [invisible] widget [name] as widgetKind [in group|card]

o As WidgetKind is the quoted name of the module of any currently installed widget. Widget modules are reverse-addressed.
create widget "My Navbar" as "com.openxtalk.widget.navbar"

To Install A New Widget
1. Download an extension file (.ice) to the desktop: https://livecode.com/extensions
2. Select the .ice file: Tools > Extension Manager > Widget, "+" button at topRight. A copy is placed in the MyOpenXTalk > Extensions folder.
3. Create a new widget: Object > New Widget, choose from enabled Widgets list.

The Delete Object Command
Use the delete command to remove objects from the stack. See delete in the Dictionary.
delete {object}

o Object is any available object.
delete this card
delete button "New Button"
delete last button

Creating Objects Off-Screen Using Template Objects
The IDE uses template objects to specify the properties for an object before it is created. The template objects are off-screen models. There's one for each possible type of object: button, field, graphic, etc.

To create a new object and then set some properties on the object, it's more efficient to make changes to the template object, then create the object. There's no need to lock the screen, update or reposition the object after creating it, and then unlock the screen.

Set properties on template objects in the same way properties on normal objects are set. See template in the Dictionary.
set the {property} of the template{objectType} to {value}

For example, to create a button with the name "Hello World", positioned at 100,100:

```
on mouseUp
  set the name of the templateButton to "Hello World"
  set the location of the templateButton to "100,100"
  create button
  reset the templateButton
end mouseUp
```

After using the templateObject to create a new object, reset it. Resetting the templateObject sets all of its properties back to defaults.
reset the template[objectType]

For example, to reset the templateButton:
reset the templateButton

7.6) Property Arrays Using The Properties Property
In addition to retrieving individual object properties, you can retrieve or set an entire set of properties in an array. Use the properties property to edit, copy, export, or import properties as an entire set.

The properties of an object is an array containing that object's significant built-in properties.
put the properties of button 1 into myArray --array set of button 1's properties to container
set the properties of last player to the properties of player "Example" --last player now has properties of player "Example"

This example shows how to write the properties of an object and their values to a text file.

```
on mouseUp
  local tPropertiesArray
  put the properties of button 1 into tPropertiesArray
  combine tPropertiesArray using return and comma  --line list of property,value
  ask file "Save properties as:"
  if it is not empty then put tPropertiesArray into URL ("file:" & it)  --text file
end mouseUp
```

This example shows how to retrieve the properties of the array.

```
on mouseUp
   local tPropertiesArray
   put the properties of field 1 into tPropertiesArray
   combine tPropertiesArray with return and comma  --line list of property,value
   put myArray  --to message box
end mouseUp
```

7.7) Custom Properties
A custom property is a property you create for an object, in addition to its built-in properties. Define custom properties for any object, and use them to store any kind of data.

This topic discusses how to create and use custom properties, and how to organize custom properties into array sets. The following section covers how to create virtual properties and use getProp and setProp handlers to handle custom property requests.

Using Custom Properties
A custom property is a property that you define. Create as many custom properties for an object as you want, and put any kind of data into them, even binary data or a file.

Use a custom property to associate data with a specific object, save the data with the object in the stack file, and access the data quickly.

Creating A Custom Property
Create a custom property by setting the new property to a value. Setting a custom property that doesn't exist automatically creates the custom property and sets it to the requested value. Create a custom property in a handler or the message box, by using the set command. The following statement creates a custom property called "cEndingTime" for a button:

set the cEndingTime of button "Session" to the long time --cEndingTime custom prop of button with value

Create custom properties for any object. However, you cannot create global custom properties, or custom properties for a chunk of text in a field. Unlike some built-in properties, a custom property applies only to an object.

Important: Each object can have its own custom properties, and custom properties are not shared between objects. Creating a custom property for one object does not create it for other objects.

The Content Of A Custom Property
Set the value of a custom property by using the set command together with the property name, the same as built-in properties:
set the cMyCustomProperty of button 1 to "false" --cMyCustomProperty custom prop of button with value

See and change an object's custom properties in the Custom Properties tab of the object's property inspector. Click the custom property to change, then enter the new value.

Changing A Part Of A Property
Like built-in properties, custom properties are not containers, so you cannot use a chunk expression to change a part of the custom property. Instead, put the property's value into a variable container, change the variable, then set the custom property to the new variable contents:
put the cLastCall of this card into tMyVar --cLastCall custom prop of card with old value
put "March" into word 3 of tMyVar
set the cLastCall of this card to tMyVar --cLastCall custom prop of card with new value
---
Custom Property Names
The name of a custom property must consist of a single word and may contain any combination of letters, digits, and underscores (_). The first character must be either a letter or an underscore.

Avoid giving a custom property the same name as a variable. If you refer to a custom property in a handler, and there is a variable by the same name, the contents of the variable is used as the name of the custom property. This usually causes unexpected results.

Avoid giving a custom property the same name as existing Engine properties unless only used in a custom property set. This usually causes unexpected results.

Important: Custom property names beginning with "rev" are reserved for the IDE's own custom properties. Avoid giving a custom property a reserved name. This usually causes unexpected results in the development environment.

Tip: There is no customPropertyNames function for a list of custom properties in a stack. You are responsible for tracking your custom properties and what they are associated with: the stack, a card, or an object.

Referring To Custom Properties
Custom property references look just like built-in property references: the word the, the property name, the word of, and a reference to the object. For example, to use a custom property called "cLastCall" that belongs to a card:
put the cLastCall of this card into field "Date" --value in cLastCall put into field "Date"

Nonexistent Custom Properties
Custom properties that don't exist evaluate to empty. For example, if the current card doesn't have a custom property called "cFirstCall", the following statement empties the field:
put the cFirstCall of this card into field "Date" --field "Date" is now empty

Note: Referring to a nonexistent custom property does not cause a script error, so a misspelled custom property in a handler might initially go unnoticed.

Finding Whether A Custom Property Exists
The customKeys property of an object lists the object's custom properties, one per line:

put the customKeys of button 1 into field "Custom Props"

To find whether a custom property for an object exists, check in the Custom Properties tab of the object's property inspector, or check whether it's listed in the object's customKeys. The following statement checks whether a player has a custom property called "cTellAll":
if "cTellAll" is among the lines of the customKeys of player "My Player" then...

Custom Properties & Converting Text Between Platforms
Moving a stack developed on a Mac to a Windows or Linux system (or vice versa), automatically translates text in fields and scripts into the appropriate character set. However, text in custom properties is not converted. This is because custom properties can contain binary data as well as text, and converting them would garble the data.

Characters whose ASCII value is between 128 and 255, such as curved quotes and accented characters, do not have the same ASCII value in the Mac OS X character set and the ISO 8859-1 character set used on Linux and Windows systems. If such a character is in a field, it is automatically translated, but it's not translated if it's in a custom property.

If a stack displays custom properties to the user, for example in a field or dialog, and the text contains special characters, the text may be displayed incorrectly when the stack is moved between platforms. To avoid this problem, use one of these methods:

o Before displaying the custom property, convert it to the appropriate character set using the macToISO or ISOToMac function. The following example shows how to convert a custom property that was created on a Mac OS X system, when the property is displayed on a Linux or Windows system:
if the platform is "MacOS" then answer the cMyPrompt of button 1 with "OK" --cMyPrompt custom prop
else answer macToISO(the cMyPrompt of button 1) with "OK" --cMyPrompt text converted to ISO for windows and linux

o Instead of storing the custom property as text, store it as HTML, using the HTMLText property of fields:
set the cMyPrompt of button 1 to the HTMLText of field 1 --cMyPrompt text is HTML

Because the HTMLText property encodes special characters as entities, it ensures that the custom property does not contain any special characters - only the platform independent encodings for them. You can then display it for any platform:
answer the cMyPrompt of button 1 with "OK" --cMyPrompt text is HTML

Storing A File In A Custom Property
Use a URL to store a file's content in a custom property:
set the cMyStoredFile of stack "My Stack" to url ("binfile:" & "mypicture.jpg") --url is not a function

Restore the file by putting the custom property's value into a URL:
put the cMyStoredFile of stack "My Stack" into url ("binfile:" & "mypicture.jpg")

Because a custom property can hold any kind of data, store either text files or binary files in a custom property. Use this capability to bundle media files or other files in the stack.

Many Mac OS Classic files have a resource fork. To store and restore such a file, use the resfile URL scheme to store the content of the resource fork separately. To save space, compress the file before storing it:
set the cMyStoredFile of stack "My Stack" to compress(url ("binfile:" & "mypicture.jpg"))

When restoring the file, decompress it first:
put decompress(the cMyStoredFile of stack "My Stack") into url ("binfile:" & "mypicture.jpg")

Deleting A Custom Property
The customKeys property of an object is a list of the object's custom properties. Set the customKeys of an object to control which custom properties it has. There is no command to delete a custom property. Instead, place all the custom property names in a variable, delete the one you don't want from that variable, then set the object's customKeys to the modified contents of the variable. This removes the custom property whose name you deleted.

For example, the following statements delete a custom property called "cCustomPropToRemove" from button "My Button":

```
get the customKeys of button 1  --delete a custom prop
set the wholeMatches to "true"
delete line lineOffset("cCustomPropToRemove",it) of it
set the customKeys of button 1 to it
```

You can also delete a custom property in the Custom Properties tab of the object's Property Inspector. Select the property's name and click the Delete button to remove it.

7.8) Custom Property Sets
Custom properties can be organized into custom property array sets. A custom property set is a group of custom properties that has a name you specify. The currently-active custom property set is used by the Engine when referring to, creating, or setting a custom property. Only one custom property set is active at a time, but you can use array notation to get or set custom properties in other non-active sets.

The previous examples assume you haven't created any custom property sets. If creating a custom property without creating a custom property set for it, the new custom property becomes part of the object's default custom property set.

Creating Custom Property Sets
To make a custom property set active, set the object's customPropertySet property to the set you want to use. As with custom properties and local variables, if the custom property set specified doesn't exist, it's automatically created and becomes the currently-active set.

The following statement creates a custom property set called "Alternate" for an object which also makes it the active set:

```
set the customPropertySet of the target to "Alternate"  --creates new set, currently-active
```

View, create, and delete custom property sets in the Custom tab of the object's property inspector.

List all the custom property sets of an object using the customPropertySets property.

Tip: As with custom properties, create custom property sets for any object. But you can't create global custom property sets, or custom property sets for a chunk of a field.

Custom Property Set Names
The names of custom property sets should consist of a single word, with any combination of letters, digits, and underscores (_). The first character should be either a letter or an underscore. When using the Custom Properties tab in the property inspector to create a custom property set, you're restricted to these guidelines.

It is possible to create a custom property set with a name that has more than one word, or that otherwise doesn't conform to these guidelines. However, this is not recommended because such a custom property set can't be used with the array notation described below.

Referring To Custom Property Sets
To switch the currently-active custom property set, set the object's customPropertySet property to the name of the desired set:

```
set the customPropertySet of button 3 to "Spanish"  --Spanish set is currently-active
```

Any references to custom property refer to the currently-active custom property set. For example, suppose there are two custom property sets named "Spanish" and "French", and the French set includes a custom property called "Paris" while the Spanish set does not. Switch to the Spanish set and the customKeys of the object won't include "Paris". The current custom property set doesn't include that property.

If referring to a custom property that isn't in the current set, the reference evaluates to empty. If you set a custom property that isn't in the current set, the custom property is created in the set. There can be two custom properties with the same name in

different custom property sets, and they don't affect each other. Changing one does not change the other.

The customProperties property of an object includes only the custom properties in the currently-active custom property set. To specify the customProperties of a particular custom property set, which is an array, include the set's name in array square brackets:

```
put the customProperties["Spanish"] of this card into myArray1
get the customProperties["MyProps"] of me  --array in special variable it
```

Finding Out Whether A Custom Property Set Exists

The customPropertySets property of an object lists the object's custom property sets, one per line:

```
answer the customPropertySets of field "My Field"
```

The following statement checks whether an image has a custom property set called "Spanish":

```
if "Spanish" is among the lines of the customPropertySets of image ID 23945 then...
```

Also look in the Custom Properties tab of the object's property inspector, which lists the custom property sets in the "Set" menu half way down.

The Default Custom Property Set

An object's default custom property set is the set that's active if you haven't used the customPropertySet property to switch to another set. Every object has a default custom property set, you don't need to create it.

If you create a custom property without first switching to a custom property set, the custom property is created in the default set. If you don't set the customPropertySet property, all your custom properties are created in the default set.

The default custom property set has no name of its own, and is not listed in the object's customPropertySets property. To switch from another set to the default set, set the object's customPropertySet to empty:

```
set the customPropertySet of the target to empty  --switch to default customPropertySet, contents are not deleted
```

Using Multiple Custom Property Sets

Since only one custom property set can be active at a time, you can create separate custom properties with the same name but different values in different sets. Which value you get depends on which custom property set is currently active.

A Translation Example.

Suppose a stack uses several custom properties that hold strings in English, to be displayed to the user by various commands. The stack might contain a statement such as:

```
answer the standardErrorPrompt of this stack  --only in English
```

The statement above displays the contents of the custom property called "StandardErrorPrompt" in a dialog box. To translate the application into French, make the original set of English custom properties into a custom property set called "EnglishStrings", and create a new set called "FrenchStrings" to hold the translated properties.

Each set has the same-named properties, but the values in one set are in English and the other in French. Switch between the sets depending on what language the user chooses. Now the same statement provides either the English or French translations, depending on which custom property set is active.

```
set the customPropertySet of this stack to "FrenchStrings"  --currently-active custom prop set
answer the standardErrorPrompt of this stack  --in French
```

Copying Custom Properties Between Property Sets

When it's created, a custom property set is empty. There aren't any custom properties in it yet. Put custom properties into a new custom property set by creating the custom properties while the set is active.

1. Create a new set and make it active: set the customPropertySet of button 1 to "MyNewSet"
2. Now create a new custom property in the current set: set the myCustomProp of button 1 to "true"

Or use the customProperties property to copy custom properties between sets.
1. Create a new set and make it active: set the customPropertySet of button 1 to "MyNewSet"
2. Copy the properties in "MyOldSet" to the new set: set the customProperties["MyNewSet"] of button 1 to the customProperties["MyOldSet"] of button 1

Caution: The IDE uses custom property sets whose names start with "cRev". If changing custom property sets, avoid these sets. For example, when using a repeat loop that goes through all of an object's custom property sets, skip any whose names start with "cRev", to prevent accidental interfere with reserved custom properties.

Arrays, Custom Properties, And Custom Property Sets
All the custom properties in a custom property set form an array. The array's name is the custom property set name, and the elements of the array are the individual custom properties in that custom property set.

Referring To Custom Properties Using Array Notation
Use array notation to refer to custom properties in any custom property set, even if it's not in the current set.

For example, suppose a button has a custom property named "cMyProp" which is in a custom property set called "MySet". If "MySet" is the current set, refer to the "cMyProp" custom property like this:
```
get the cMyProp of button 1  --MySet is currently active
set the cMyProp of the target to "20"  --MySet is currently active
```

Or use array notation to refer to the "cMyProp" custom property, even if "MySet" is not the current set:
```
get the MySet["cMyProp"] of button 1  --MySet need not be currently active
set the MySet["cMyProp"] of the target to "20"  --MySet need not be currently active
```

Tip: Because the default custom property set has no name, you cannot use array notation to refer to a custom property in the default set.

Storing An Array In A Custom Property Set
When storing a set of custom properties in a custom property set, the set can be used like an array. Think of the custom property set as though it were a single custom property, and the properties in the set as the individual elements of the array. To store an array variable as a custom property set:
```
set the customProperties["MyPropertySet"] of me to tMyArray
```

The statement above creates a custom property set called "MyPropertySet", and stores each element in "tMyArray" as a custom property in the new set. To retrieve a single element of the array, use a statement like this:
```
get the MyPropertySet["MyElement"] of field "Example"
```

Deleting A Custom Property Set
The customPropertySets property of an object is a list of all the object's custom property sets. Set the customPropertySets of an object to control which custom property sets it has.

There is no command to delete a custom property set. Instead, place all the custom property set names in a variable, delete the one you don't want from that variable, then set the customPropertySets back to the modified contents of the variable. This removes the deleted custom property set. For example, the following statements delete a custom property set called "MySet" from the button "My Button":

```
get the customPropertySets of button "My Button"
set the wholeMatches to "true"
delete line lineOffset("MySet",it) of it  --delete MySet from list
set the customPropertySets of button "My Button" to it
```

You can also delete a custom property set in the Custom tab of the object's property inspector. Select the set's name from the Set menu, then click the Delete button to remove it.

7.9) Attaching Handlers To Custom Properties - setProp and getProp.
When a custom property is changed, a setProp trigger is sent to the object whose property is being changed. Write a setProp handler to trap this trigger and respond to the attempt to change the property. Place the setProp handler anywhere in the object's message path.

Similarly, when you get the value of a custom property, a getProp call is sent to the object whose property is being queried. Write a getProp handler to reply to the request for information. Like a function call, the getProp call also traverses the message path.

Use setProp and getProp handlers to:
o Validate a custom property's value before setting it.
o Report a custom property's value in a format other than what it's stored as.
o Ensure the integrity of a collection of properties by setting them all at once.
o Change an object's behavior when a custom property is changed.

Note: setProp triggers and getProp calls are not sent when a built-in property is changed or accessed. They apply only to custom properties.

Responding To Changing A Custom Property
When using the set command to change a custom property, a setProp trigger is sent to the object whose property is being changed. A setProp trigger acts very much like a message does. It is sent to a particular object. If that object's script contains a setProp handler for the property, the handler is executed; otherwise, the trigger travels along the message path until it finds a handler for the property. If it reaches the end of the message path without being trapped, the setProp trigger sets the custom property to its new value.

Include as many setProp handlers in a script for as many different custom properties needed.

The Structure Of A SetProp Handler
Unlike a message handler, a setProp handler begins with the word setProp instead of the word on. This is followed by the handler's name (which is the same name of the custom property) and a parameter that holds the property's new value. A setProp handler, like all handlers, ends with the word "end" followed by the handler's name. See setProp in the Dictionary.

The following example shows a setProp handler for a custom property named "cPercentUsed", and can be placed in the script of the object whose custom property it is:

```
setProp cPercentUsed newAmount  --responds to setting the cPercentUsed custom property
  if newAmount is not a number or newAmount < zero or newAmount > 100 then
    beep 2
    exit cPercentUsed  --prevent changing custom prop
  else pass cPercentUsed  --allow engine to change custom prop
end cPercentUsed
```

When setting the "cPercentUsed" custom property, the "cPercentUsed" setProp handler is executed:
set the cPercentUsed of scrollbar "Progress" to "90"

When this statement is executed, a setProp trigger is sent to the scrollbar. The new value of 90 is placed in the newAmount parameter. The handler makes sure the new value is in the range 0 to 100, or it beeps and exits the handler preventing the property from being set.

Passing The SetProp Trigger
If the setProp trigger reaches the Engine, the last stop in the message path, the custom property is set. If the trigger is trapped and doesn't reach the Engine, the custom property is not set.

To let a trigger pass further along the message path, use the pass control structure. The pass control structure stops the current handler and sends the trigger on to the next object in the message path, just as though the object didn't have a handler for the

custom property.

In the "cPercentUsed" handler above, if "newAmount" is out of range, the handler uses the exit control structure to trap the trigger and prevent it from reaching the Engine. If "newAmount" is within range, the handler uses the pass control structure to allow the trigger to reach the Engine to set the custom property.

Use this capability to check the value of any custom property before allowing it to be set. For example, if a custom property is supposed to be boolean (true or false), a setProp handler can trap the trigger if the value is anything but true or false:

```
setProp myBoolean newValue
  if newValue is "true" or newValue is "false" then pass myBoolean  --allow engine to set custom prop
exit myBoolean
```

Using The Message Path With A SetProp Trigger

Because setProp triggers use the message path, a single object can receive the setProp triggers for all the objects it owns. For example, setProp triggers for all objects on a card are sent to the card if the control's script has no handler for that custom property. You can take advantage of the message path to implement the same setProp behavior for objects that all have the same custom property.

Caution: If a setProp handler sets its custom property, for an object that has that setProp handler in its message path, a runaway recursion will result. To avoid this problem, set the lockMessages property to true before setting the custom property.

For example, suppose all the cards in a stack have a custom property named "cLastChanged". Instead of putting a setProp handler for this property in each card's script, put a single setProp handler in the stack script:

```
setProp cLastChanged newDate
  convert newDate to seconds
  lock messages  --prevent recursion
  set the cLastChanged of the target to newDate  --the target is a card
  unlock messages
end lastChanged
```

Note: To refer to the object whose property is being set, use the target function. The target refers to the object that first received the setProp trigger - the object whose custom property is being set - even if the handler being executed is in the script of another object.

Setting Properties Within A SetProp Handler

In the "cLastChanged" example above, the handler sets the custom property directly, instead of passing the setProp trigger. You must use this method if the handler makes a change to the property's value, because the pass control structure simply passes on the original value of the property.

Important: If you use the set command within a setProp handler to set the same custom property for the current object, no setProp trigger is sent to the target object. (This is to avoid runaway recursion, where the setProp handler triggers itself.) Setting a different custom property sends a setProp trigger. So does setting the handler's custom property for an object other than the one whose script contains the setProp handler.

Using this method will not only check the value of a property, and allow it to be set only if it's in range, but also change the value so that it is in the correct range and has the correct format. The following example is similar to the "cPercentUsed" handler above, but instead of beeping if newAmount is out of range, it forces the new value into the range 0–100:

```
setProp cPercentUsed newAmount
  if newAmount is a number then set the cPercentUsed of the target to max(zero,min(100,newAmount))
end cPercentUsed
```

Nonexistent Properties

If the custom property specified by a setProp handler doesn't exist, the setProp handler is still executed when a handler sets the property. If the handler passes the setProp trigger, the custom property is created.

Custom Property Sets And SetProp Handlers
A setProp handler for a custom property set behaves differently from a setProp handler for a custom property that's in the default set.

When a custom property is set in a custom property set, the setProp trigger is named for the set, not the property. The property name is passed in a parameter using a special notation. This means that, for custom properties in a set, you write a single setProp handler for the set, rather than one for each individual custom property.

The following example handles setProp triggers for all the custom properties in a custom property set called FrenchStrings, which contains custom properties named cStandardErrorPrompt, cFilePrompt, and perhaps other custom properties:

```
setProp FrenchStrings[myPropertyName] newValue
  switch myPropertyName  --myPropertyName parameter contains the name of the custom property being set
    case "cStandardErrorPrompt"
      set the FrenchStrings["cStandardErrorPrompt"] of the target to return & newValue & return
      exit FrenchStrings
      break
    case "cFilePrompt"
      set the FrenchStrings["cfilePrompt"] of the target to return& newValue & return
      exit FrenchStrings
      break
    default
      pass FrenchStrings  --other custom props in same custom prop set pass to engine to be set
  end switch
end FrenchStrings
```

The name of this setProp handler is the same name as the custom property set that it controls - "FrenchStrings". Because there is only one handler for all the custom properties in this set, the handler uses the switch control structure to perform a different action for each custom property it deals with.

Suppose you change the "cStandardErrorPrompt" custom property:
set the customPropertySet of this stack to "FrenchStrings"
set the cStandardErrorPrompt of this stack to field 1

A setProp trigger is sent to the stack, which causes the above handler to execute. The custom property - "cStandardErrorPrompt" - is placed in the "myPropertyName" parameter, and the new value - the contents of field 1 - is placed in the "newValue" parameter. The handler executes the case for "cStandardErrorPrompt", putting a return character before and after the property before setting it.

If you set a custom property other than "cStandardErrorPrompt" or "cFilePrompt" in the "FrenchStrings" set, the default case is executed. In this case, the pass control structure lets the setProp trigger proceed along the message path to the Engine to set the custom property.

Address a custom property in a set either by first switching to that set, or using array notation to specify both set and custom property. For example:
set the customPropertySet of me to "MySet"
set the cMyProperty of me to "true"

They are equivalent to: set the MySet["cMyProperty"] of me to "true" --set and prop in array notation

Regardless of how the custom property is set, if it's a member of a custom property set, the setProp trigger has the name of the set - not the custom property itself - and you must use a setProp handler in the form described above to trap the setProp trigger.

Responding to A Request For The Value Of A Custom Property - getProp.
When a custom property is used in an expression, a getProp call is sent to the object whose property's value is being requested.

A getProp call acts very much like a custom function call. It's sent to a particular object. If that object's script contains a getProp handler for the custom property, the handler is executed, and the value is returned. Otherwise, the call travels along the message path until it finds a handler for the property. If the getProp call reaches the end of the message path without being trapped, the Engine returns the value.

You can include as many getProp handlers in a script as needed.

The Structure Of A GetProp Handler
Unlike a message handler, a getProp handler begins with the word getProp instead of the word on. This is followed by the handler's name (which is the same name of the custom property). A getProp handler, like all handlers, ends with the word "end" followed by the handler's name. See getProp in the Dictionary.

The following example is a getProp handler for a custom property named "cPercentUsed":

```
getProp cPercentUsed
  global LastAccessTime
  put the seconds into LastAccessTime
  pass cPercentUsed
end cPercentUsed
```

If the "cPercentUsed" custom property is used in an expression, the getProp handler is executed:
put the cPercentUsed of card 1 into myVariable

When this statement is executed, a getProp call to the card is sent to retrieve the value of the "cPercentUsed" custom property. This executes the getProp handler for the property. The example handler stores the current date and time in a global variable before the property is evaluated.

Returning A Value From A GetProp Handler
When the getProp trigger reaches the engine - the last stop in the message path - the Engine gets the custom property from the object and returns its value. To let a trigger pass further along the message path, use the pass control structure. The pass control structure stops the current handler and sends the call on to the next object in the message path, just as though the object didn't have a handler for the custom property.

To return a value other than the value that's stored in the custom property - for example, if you want to reformat the value first - use the return control structure instead of passing the getProp call. The following example is a getProp handler for a custom property named "cLastChanged", which holds a date in seconds:

```
getProp cLastChanged
  get the cLastChanged of the target
  convert it to long date
  return it  --return value not stored in custom prop
end lastChanged
```

The return control structure, when used in a getProp handler, returns a property value to the handler that requested it. In the above example, the converted date - not the stored value - is returned. Return the actual stored value of a custom property or any value at all from a getProp handler.

Important: If using a custom property's value within the property's getProp handler, no getProp call is sent to the target object. This is to avoid runaway recursion, where the getProp handler calls itself.

A getProp handler can either use the return control structure to return a value, or use the pass control structure to let the Engine

get the custom property value from the object.

If the getProp call is trapped before it reaches the engine and no value is returned in the getProp handler, the custom property reports a value of empty. In other words, a getProp handler must include either a return control structure or a pass control structure, or its custom property will always be reported as empty.

Using The Message Path With A GetProp Call
Because getProp calls use the message path, a single object can receive the getProp calls for all the objects it owns. For example, getProp calls for all objects on a card are sent to the card if the control's script has no handler for that property. You can take advantage of the message path to implement the same getProp behavior for objects that all have the same custom property.

Caution: If a getProp handler is not attached to the object that has the custom property and it uses the value of the custom property, a runaway recursion will result. To avoid this problem, set the lockMessages property to true before getting the custom property's value.

Nonexistent Properties
If the custom property specified by a getProp handler doesn't exist, the getProp handler is still executed if the property is used in an expression. Nonexistent properties report empty. Getting the value of a custom property that doesn't exist does not cause a script error.

Custom Property Sets And GetProp Handlers
A getProp handler for a custom property set behaves differently from a getProp handler for a custom property that's in the default set.

When using the value of a custom property in a custom property set, the getProp call is named for the set, not the property. The custom property name is passed in a parameter using array notation. For custom properties in a set, write a single getProp handler for the set, rather than one for each individual custom property.

The following example handles getProp calls for all custom properties in a custom property set called ExpertSettings, which contains custom properties named cFileMenuContents, cEditMenuContents, and perhaps other custom properties:

```
getProp ExpertSettings[PropertyName]  --PropertyName parameter contains the name of the custom property
  switch PropertyName
    case "cFileMenuContents"
      if the ExpertSettings[cFileMenuContents] of the target is empty then return "(No Items"
      else pass ExpertSettings
      break
    case "CEditMenuContents"
      if the ExpertSettings[cEditMenuContents] of the target is empty then return the NoviceSettings[cEditMenuContents] of the
target
      else pass ExpertSettings
      break
    default
      pass expertSettings
  end switch
end ExpertSettings
```

The name of this getProp handler is the same as the name of the custom property set that it controls - "ExpertSettings". Because there is only one handler for all the custom properties in this set, the handler uses the switch control structure to perform a different action for each property that it deals with.

Suppose you get the "cFileMenuContents" custom property:
set the customPropertySet of button 1 to "ExpertSettings"
put the cFileMenuContents of button 1 into me

A getProp call to the button is sent, which causes the above handler to execute. The property queried - "cFileMenuContents" - is placed in the "PropertyName" parameter. The handler executes the case for "cFileMenuContents". If the property is empty, it returns "(No Items". Otherwise, the pass control structure lets the getProp call proceed along the message path, and when it reaches the engine, the current value is returned.

7.10) Virtual Properties
A virtual property is a custom property that exists only in a setProp and/or getProp handler, and is never actually set. Virtual properties are never attached to the object. Instead, they act to trigger setProp or getProp handlers that do the actual work.

Using the set command with a virtual property, its setProp handler is executed, but the setProp trigger is not passed to the engine, so the virtual property is not attached to an object. Using a virtual property in an expression, its getProp handler returns a value without referring to the object. In both cases, using a virtual property simply executes a handler.

Use virtual properties to:
o Give an object a set of behaviors.
o Compute a value for an object.
o Implement a new property that acts like a built-in property.

When To Use Virtual Properties
Because they're not stored with the object, virtual properties are transient: they're re-computed every time requested. When a custom property depends on other properties that may be set independently, it's appropriate to use a virtual property.

For example, the following handler computes the current position of a scrollbar as a percentage (instead of an absolute number):

```
getProp asPercentage  --of a scrollbar
  put the endValue of the target - the startValue of the target into tValueExtent
  return the thumbPosition of me * 100 / tValueExtent
end asPercentage
```

The "asPercentage" virtual property depends on the scrollbar's thumbPosition, which can be changed at any time (either by the user or by a handler). A custom property for the object would have to be re-computed every time the scrollbar is updated in order to stay current. By using a virtual property, the value of the property is never out of date, because the getProp handler recomputes it every time the "asPercentage" of the scrollbar is called.

Virtual properties are also useful solutions when a property's value is large. Because the virtual property isn't stored with the object, it doesn't take up disk space, and only takes up memory when it's computed.

Another reason to use a virtual property is to avoid redundancy. The following handler sets the width of an object, not in pixels, but as a percentage of the object's owner's width:

```
setProp percentWidth newPercentage
  set the width of the target to the width of the owner of the target * newPercentage / 100
end percentWidth
```

Suppose this handler is placed in the script of a card button in a 320-pixel-wide stack. If the button's "percentWidth" is set to 25, the button's width is set to 80, 25% of the card's 320-pixel width. It doesn't make much sense to store an object's percentWidth, because it's based on the object's width and its owner's width.

Consider using virtual properties to define an attribute of an object, but it doesn't make sense to store the attribute with the object - because it would be redundant, because possible changes to the object mean it would have to be re-computed anyway, or because the property is too large to be easily stored.

Handlers For A Virtual Property
A handler for a virtual property is structured like a handler for a custom property. The only structural difference is, since the

handler has already done everything necessary, there's no need to actually attach the virtual property to the object or get its value from the object. When setting a virtual property or using its value, the setProp trigger or getProp call does not reach the engine, but is trapped by a handler first.

Virtual Property SetProp Handlers
A setProp handler for an ordinary custom property includes the pass control structure, allowing the setProp trigger to reach the engine and set the custom property (or else it includes a set command that sets the property directly).

A handler for a virtual property does not include the pass control structure, because a virtual property should not be set. Since the property is set automatically when the trigger reaches the end of the message path, a virtual property's handler does not pass the trigger.

If you examine an object's custom properties after setting a virtual property, you'll find that the custom property hasn't actually been created. This happens because the setProp handler traps the call to set the property; unless you pass the setProp trigger, the property isn't passed to the Engine, and the property isn't set.

Virtual Property GetProp Handlers
Similarly, a getProp handler for an ordinary custom property either gets the property's value directly, or passes the getProp call so the engine can return the property's value. But in the case of a virtual property, the object doesn't include the property, so the getProp handler must return the value.

Creating New Object Properties
Use virtual properties to create a new property that applies to all objects, or to all objects of a particular type. Such a property acts like a built-in property, because it can be used for any object. And because a virtual property doesn't rely on a custom property being stored in the object, there's no need to create the property for each new object created. The virtual property is computed only when used in an expression.

The following example describes how to implement a virtual property called "percentWidth" that behaves like a built-in property. Suppose the "percentWidth" handler above is placed in a stack script instead of in a button's script:

```
setProp percentWidth newPercentage
  set the width of the target to the width of the owner of the target * newPercentage / 100
end percentWidth
```

Because setProp triggers use the message path, if setting the "percentWidth" of any object in the stack, the stack receives the setProp trigger (unless it's trapped by another object first). This means that if the handler is in the stack's script, you can set the "percentWidth" property of any object in the stack.

If the handler is placed in a backscript, you can set the "percentWidth" of any object, anywhere in the application.

Tip: To refer to the object whose property is being set, use the target function. The target refers to the object that first received the setProp trigger - the object whose custom property is being set - even if the handler being executed is in the script of another object.

Getting the "percentWidth" property
The matching getProp handler, which retrieves the "percentWidth" of an object, looks like this:

```
getProp percentWidth
  return 100 * (the width of the target / the width of the owner of the target)
end percentWidth
```

If the handler above is placed in a card button's script, the following statement reports the button's width as a percentage:
```
put the percentWidth of button "My Button" into field 12
```

For example, if the stack is 320 pixels wide and the button is 50 pixels wide, the button's width is 15% of the card width, and the

statement puts "15" into the field.

Like the setProp handler for this property, the getProp handler should be placed far along the message path. Putting it in a stack script makes the property available to all objects in the stack; putting it in a backscript makes the property available to all objects in the application.

Limiting The "percentWidth" Property
Most built-in properties don't apply to all object types, and you might also want to create a virtual property that only applies to certain types of objects. For example, it's not very useful to get the width of a substack as a percentage of its mainstack, or the width of a card as a percentage of the stack's width.

Limit the property to certain object types by checking the target object's name:

```
setProp percentWidth newPercentage
  if word 1 of the name of the target is among the words of "stack card" then exit setProp
  set the width of the target to the width of the owner of the target * newPercentage / 100
end percentWidth
```

The first word of an object's name is the object type, so the above revised handler ignores setting the "percentWidth" if the object is a stack or card.

7.11) Managing Windows, Palettes, and Dialogs
There is complete control over all aspects of window management, including moving, re-layering, and changing window mode.

Moving A Window
The location property or rectangle property of a stack is used to move the stack window. The location property specifies the center of the stack's window, relative to the top left corner of the main screen. Unlike the location of objects, the location of a stack is specified in absolute coordinates.

The following statement moves a stack to the center of the main screen:
```
set the location of stack "Wave" to the screenLoc
```

The rectangle property of a stack specifies the position of all four edges, and can be used to resize the stack window as well as move it:
```
set the rectangle of this stack to "100,100,600,200"
```

Tip: To open a window at a particular place without flickering, set the stack's location or rectangle property to the desired value either before going to it, or in the stack's preOpenStack handler.

Use associated properties to move a stack window too. Changing a stack's bottom or top property moves the window up or down on the screen. Changing the left or right property moves the window from side to side.

Changing A Window's Layer
Bring a window to the front by using the go command:
```
go stack "Alpha"
```

If the stack is already open, the go command brings it to the front, without changing its mode.

To find the layer order of open stack windows, use the openStacks function. This function lists all open stack windows in order from front to back.

The Palette Layer
Palette windows float above editable windows and modeless dialog boxes. A palette will always above a standard window, even if you bring the standard window to the front. This helps ensure that palettes, which usually contain tools used in any window,

cannot disappear behind document windows. It's also a good reason to design small palette windows, because other windows cannot be moved if a palette blocks part of the window.

The System Palette Layer
System windows - stacks whose systemWindow property is true - float above all other windows, in every running application. Even if the user brings another application to the front, your application's system windows remain in front of all windows. System windows are always in front of other windows.

The Active Window
In most applications, commands are applied to the active window. Given the flexibility to use several different window types, not all of which are editable, the current stack is not always the active window. The current stack is the target of menu choices such as View > Go Next and is the stack specified by the expression this stack.

For example, executing the find command may have unexpected results if stacks of different modes are open, because under these conditions, the search may target a stack that is not the front-most window.

Finding The Current Stack
The current stack - the stack that responds to commands - is designated by the defaultStack property. To determine which stack is the current stack, use the following rules:
o If any stacks are opened in an editable window, the current stack is the front-most unlocked stack. (the cantModify property = false)
o If there are no unlocked stacks open, the current stack is the front-most locked stack in an editable window.
o If there are no stacks open in an editable window, the current stack is the front-most stack in a modeless dialog box.
o If there are no editable or modeless windows open, the current stack is the front-most palette.

Another way of expressing this set of rules is to say that the current stack is the front-most stack with the lowest mode property. You can find out which stack has the lowest mode using the topStack function.

The topStack function and the defaultStack property:
The topStack function goes through the open stacks first by mode, then by layer. For example, if any editable windows are open, the topmost editable window is the topStack. If there are no editable windows, the topStack is the topmost modeless dialog box, and so on.

The defaultStack property specifies which stack is the current stack. By default, the defaultStack is set to the topStack, although you can change the
defaultStack to any open stack.

Changing The Current Stack
To operate on a stack other than the current stack, set the defaultStack property to the stack you want to target before executing commands. Usually, the defaultStack is the topStack, but you can change it to override the usual rules about which window is active. You cannot set the defaultStack to a stack that is not already open.

Note: If Linux systems are set up to use pointer focus rather than click-to-type or explicit focus, you may experience unexpected results when using this program, since the current stack will change as you move the mouse pointer. It is recommended configuring your Linux system to use explicit focus when using this program or any applications created with it.

Creating A Backdrop
A solid or patterned backdrop behind your application's windows prevent other applications' windows from being seen, but doesn't close them. A backdrop is appropriate for applications like a game or kiosk, where the user doesn't need to see other applications, and to keep distractions to a minimum.

To create a backdrop, set the backdrop property to either a valid color reference, or the ID of an image to use as a tiled pattern:
set the backdrop to "#99FF66" --color
set the backdrop to 1943 --unquoted image ID

In the development environment, display a backdrop by choosing View > Backdrop. Specify a backdrop color by choosing Edit > Preferences > Appearance, Backdrop.

Open, Closed, and Hidden Windows
Each open stack is displayed in a stack window. A stack can be open without being visible, and can be loaded into memory without being open.

Hidden Stacks
A stack window can be either shown or hidden, depending on the stack's visible property. This means a window can be open without being visible on the screen.

Tip: To list all open stacks, whether they're visible or hidden, use the openStacks function.
```
put the openStacks  --to message box, includes open IDE stacks
```

Loaded Stacks
A stack can also be loaded into memory without actually being open. A stack whose window is closed (not just hidden) is not listed by the openStacks function. However, it takes up memory, and its objects are accessible to other stacks. For example, if a closed stack that's loaded into memory contains a certain image, the image can be used as a button icon in another stack.

A stack can be loaded into memory without being open under any of the following conditions:
o A handler in a stack referred to a property of the closed stack. This automatically loads the referenced stack into memory.
o The stack is in the same stack file as another stack that is open. For example, a subStack.
o The stack was opened and then closed, and its destroyStack property was set to false. When false, the stack is closed but remains in memory.

Tip: To list all stacks in memory, whether they're open or closed, the "()" form of the revLoadedStacks function must be used.
```
put revLoadedStacks()  --to message box, only your stacks.
```

The States Of A Stack
A stack can be in any of four states:
o Open and visible: The stack is loaded into memory, its window is open and visible.
o Open and hidden: The stack is loaded into memory, its window is open and hidden, and it is listed in Window menu and Project Browser.
o Closed: The stack is not loaded into memory and has no effect on other stacks.
o Closed but loaded into memory: The stack is loaded into memory, its window is closed, and it is not listed in Window menu or the openStacks function. However, its objects are still available to other stacks, and it is listed in Project Browser. A stack that is closed but loaded into memory has a mode property of zero.

Tip: To remove such a stack from memory, choose Tools > Project Browser, find the stack's name, and Control-click it (Mac OS X) or Right-click it (Linux or Windows) in the Project Browser window and choose "Close and Remove from Memory" from the contextual menu.

Window Types And The Mode Property
Find a stack window's type by checking the stack's mode property. This read-only property reports a number from 0-14 that depends on the window type. For example, the mode of an editable window is 1, and the mode of a palette is 4. See mode in the Dictionary.

Use the mode property to check what sort of window a stack is being displayed in:
```
if the mode of this stack is "5" then close this stack  --modal dialog box
else beep  --another type of window
```

Window Appearance
Details of a window's appearance, such as the height of its title bar and the background color or pattern in the window itself, are mainly determined by the stack's mode. There are a few additional elements of window appearance that can be controlled with specific properties: the metal, backGround color, backGround pattern, and decorations properties.

The Metal Property
On Mac OS X systems, use a stack's metal property to give the stack window a textured metal appearance. This metal appearance applies to the stack's title bar and its background.

Tip: The metal appearance, in general, should be used only for the main window of an application, and only for windows that represent a physical media device such as a CD player. See Apple's Aqua user-interface guidelines for more information.

Window Background Color Or Pattern
The background color or pattern of a window's content area - the part that isn't the title bar - is determined by the window type and operating system, by default. For example, on Mac OS X systems, a striped background appears in palettes and modeless dialog boxes.

For a window to have a specific color or pattern, set the stack's backgroundColor or backgroundPattern property. This color or pattern overrides the usual color or background pattern:
set the backgroundColor of stack "Alpha" to "aliceblue"
set the backgroundPattern of stack "Beta" to 2452 --image ID

The Decorations Property
Most of the properties that pertain to a window's appearance can also be set in the stack's decorations property. The decorations of a stack consists of a comma-separated list of decorations:
set the decorations of stack "Gamma" to "title,minimize"

The statement above shows the stack's title bar, sets the stack's minimizeBox property to true, and sets the other stack decorations properties to false. Conversely, setting a stack's minimizeBox property to true, its decorations property is changed to include "minimize" as one of its items. In this way, the decorations property of a stack interacts with its closeBox, maximizeBox, minimizeBox, zoomBox, metal, shadow, menu, and systemWindow properties.
set the decorations of stack "Gamma" to "default" --appropriate decorations for the window mode
---
The Decorations Property And Menu Bars In A Window
On Linux and Windows systems, the menu bar appears at the top of the window. On these systems, whether a window displays its menu bar is determined by whether the stack's decorations property includes "menu":
set the decorations of this stack to "title,menu"

On Mac OS X systems, the menu bar appears at the top of the screen, outside any window. On these systems, the "menu" decoration has no effect.

Title Bar
The user drags the title bar of a window to move the window around the screen. In general, if the title bar is not displayed, the user cannot move the window. Use the decorations property to hide and show a window's title bar. When the user drags the window, a moveStack message is sent to the current card.

The decorations property affects only whether the window can be moved by dragging it. Even if a stack's decorations property does not include the title bar decoration, you can still set a stack's location, rectangle, and related properties to move or resize the window.

Stack Window Title
The title that appears in the title bar of a stack window is determined by the stack's label property. Changing a stack's label in a script immediately updates the window's title. If the label is empty, the title bar displays the stack's name property.

If the stack is in an editable window whose cantModify is false, an asterisk appears after the window title to indicate this, and if the stack has more than one card, the card number also appears in the window title. These indicators do not appear if the stack has a label.

Because the window title is determined by the stack's label property instead of its name property, you have a great deal of

flexibility in changing the window title. Scripts refer to a stack by its name, not its label, so the window title can be changed without changing any scripts that refer to the stack.

The Close Box
The close box allows the user to close the window by clicking it. To hide or show the close box, set the stack's closeBox property:
set the closeBox of stack "Bravo" to "false"

When the user clicks the close box, a closeStackRequest message is sent, followed by a closeStack message, to the current card.

The closeBox property affects only whether the window can be closed by clicking. Even if a stack's closeBox property is false, the close command in a handler or the message box will still close the window.

The Minimize Box Or Collapse Box
The terminology and behavior of this part of the title bar varies depending on platform. The minimize box shrinks the window to a desktop icon.

To hide or show the minimize box or collapse box, set the stack's minimizeBox property:
set the minimizeBox of this stack to "true"

On Mac OS X and Linux systems, set a stack's icon property to specify the icon that appears when the stack is minimized. When the user clicks the minimize box or collapse box, an iconifyStack message is sent to the current card.

On Mac OS X systems, the collapse box shrinks the window so only its title bar is shown. The minimize box (Mac OS X and Windows systems) or iconify box (Linux systems) shrinks the window to a desktop icon.

The Maximize Box Or Zoom Box
The terminology and behavior of this part of the title bar varies depending on platform. On Mac OS X systems, the zoom box switches the window between its current size and maximum size. The maximize box (Linux and Windows systems) expands the window to its maximum size.

To hide or show the zoom box or maximize box, set the stack's zoomBox property:
set the zoomBox of stack "Hello" to false

When the user clicks the zoom box or maximize box, a resizeStack message is sent to the current card.

Making A Stack Resizable
A stack's resizable property determines whether the user can change its size by dragging a corner or edge (depending on operating system) of the stack window.

Tip: To move and resize objects automatically to fit when a stack is resized, use the Geometry tab in the control's property inspector.

Some stack modes cannot be resized, regardless of the setting of the stack's resizable property. Modal dialog boxes, sheets, and drawers cannot be resized by the user, and do not display a resize box.

The resizable property affects only whether the window can be resized by dragging a corner or edge. Even if a stack's resizable property is set to false, you can still set a stack's location, rectangle, and related properties to move or resize the window.

When the user resizes a stack, a resizeStack message is sent to the current card.

Irregularly-Shaped And Translucent Windows
You can set a stack's windowShape property to the transparent, or alpha channel of an image that has been imported together with its alpha channel. This creates a window with "holes" or a window with variable translucency. Apply a shape to any type of stack, regardless of the mode it is opened, allowing such a window to exhibit modal behavior as a dialog, float as a palette, etc.

Use either a GIF or PNG image for irregularly shaped windows. For translucency use PNG images. Translucency is currently only supported on Windows and Mac OS X systems.

7.12) Programming Menus & Menu Bars
Menus are not a separate object type. Create a menu from either a button or a stack, then use special commands to display the menu or to include it in a menu bar.

This topic discusses menu bars, menus that are not in the menu bar (such as contextual menus, popup menus, and option menus), how to make menus with special features such as check-marks and submenus, and how to use a stack window as a menu for total control over menu appearance.

To create menu bars that work cross-platform, choose Tools > Menu Builder. The details about menu bars in this topic are needed only to edit menu bars by script, for example to include specific features not supported by the Menu Builder.

Menu Types
Several menu types are supported: pulldown menus, option menus (usually called popup menus on Mac OS X), popup menus (usually called contextual menus on Mac OS X), and combo boxes.

Each of these menu types is implemented by creating a button. If the button's style property is set to "menu", clicking it causes a menu to appear. The button's menuMode property determines what kind of menu is displayed.

Even menu bars are created by making a pulldown-menu button for each menu, then grouping the buttons to create a single menu bar. The menu bar can be moved to the top of the stack window (on Linux and Windows systems). To display the menu bar in the standard location at the top of the screen on Mac OS X systems, set the stack's menubar property to the group's name. The name of each button is displayed in the menu bar as a menu, and pulling down a menu displays the contents of the button as a list of menu items.

Button Menus
Create a button menu by dragging out one of the menu objects from the tools palette, set the style of the button to "menu", and set the menuMode of the button to the appropriate menu type in the button's property inspector or in a script handler.

To create the individual menu items that will appear in the menu, set the button's text property to the menu's contents, one menu item per line in the button's property inspector or in a script handler.

The button displays the specified menu type, and when clicked, the text entered is displayed as individual menu items in the menu.

Tip: To dynamically change the menu's contents at the time it's displayed, put a mouseDown handler in the button's script that puts the text of the desired menu items into the button. When the menu appears, it displays the new menu items.

For menus that retain a state (such as option menus and combo boxes), the button's label property holds the text of the currently chosen menu item.

Handling The menuPick Message
When the user chooses an item from the menu, a menuPick message is sent to the button. The message parameter is the name of the menu item chosen. To perform an action when the user chooses a menu item, place a menuPick handler like this one into the button's script:

```
on menuPick theMenuItem
  switch theMenuItem
    case "Name Of First Item"
      --do stuff for first item
      break
    case "Name Of Second Item"
```

```
    --do stuff for second item
    break
  case "Name Of Third Item"
    --do stuff for second item
    break
 end switch
end menuPick
```

Changing The Currently-Chosen Menu Item
For menus that retain a state (such as option menus and combo boxes), change the currently-chosen menu item by changing the button's label property to the text of the newly chosen item.

If changing the currently-chosen menu item in option menus, also set the button's menuHistory property to the line number of the newly chosen item. This ensures that the new choice will be the one under the mouse pointer the next time the user clicks the menu.
```
set the menuHistory of btn 1 to "1"  --default first menuItem for: option, combo, tab btns
```

Creating Cascading Pull-Down Menus
To create a cascading pull-down menu (also called a submenu, pull-right menu, or hierarchical menu), add a tab character to the start of menu items to place in the submenu. For example, the following text, when placed in a menu button, creates two menu items, then a submenu containing two more items, and finally a last menu item:

```
First Item
Second Item
Third Item Is A Submenu
        First Item In Submenu
        Second Item In Submenu
Last Menu Item Not In Submenu
```

The depth of a submenu item is determined by the number of tab characters before the menu item's name. The submenu item becomes part of the closest line above the submenu item that has one fewer leading tab character. This means that the first line of a menu cannot start with a tab character, and any line in the button's text can have at most one more tab character than the preceding line.

Important: Only create cascading pull-down menus. You cannot create a cascading combo box at all, and cascading option menus do not work properly on all platforms.

Cascading Menus And The MenuPick Message
When the user chooses a menu item in a cascading menu, the parameter of the menuPick message contains the menu item name and the name of the submenu it's part of, separated by a vertical bar (|). For example, if the user chooses the "Second Item In Submenu" from the pull-down menu above, the parameter sent with the menuPick message is:
```
Third Item Is A Submenu|Second Item In Submenu
```

7.13.4 Ticks, Dashes, & Checks In Menus
There are several special characters you can put at the start of a line in the button's contents to change the behavior of the menu item.
Pull-down menu can be edited in tools > MenuBuilder, the menu button's properties Inspector, or by code.

Pull-down menu special characters:
```
-  --divider line
!c  --check-mark
!n  --uncheck
!r  --diamond
!u  --no diamond
&  --next char underline for alt-key shortcut
```

/ --next char for ctrl-key or cmd-key shortcut
(--disable menuItem
tab --subMenuItem (place special chars before tab)
\(--"(" in menuItem
\) --")" in menuItem

Pull-down menu:
Manage enable/disable in mouseDown handler in group script of menubar.
Manage check/diamond in menuPick handler in script of menu button.

If any of the above special characters is included in a submenu item, the special character must be placed at the start of the line - before the tab characters that make it a submenu item.

Note: You cannot create divider lines in combo boxes or option menus on Windows systems.

There are three other special characters that can appear anywhere in a line:
o Putting the & character anywhere in a line underlines the next character and makes it the keyboard mnemonic for that menu item on Windows systems. The & character does not appear in the menu, and is not sent with the parameter to the menuPick message.

o Putting the / character anywhere in a line makes the next character the keyboard equivalent for the menu item. The / character and character following it does not appear in the menu, and is not sent with the parameter to the menuPick message. To put an & or / character in the text of a menu, double the characters: && or //.

o Putting the (character anywhere in a line disables the menu item. To put a (character in a menu item without disabling it, precede it with a backslash character: \(.

Note: You cannot disable lines in combo boxes or option menus on Windows systems.

All of the above special characters are filtered out of the parameter sent with the menuPick message when the user chooses a menu item. The parameter is the same as the characters that are actually displayed in the menu.

Note: The font and color of a button menu is determined by the button's font and color properties. However, on Mac OS X systems, the font and color of the option menus and popup menus is controlled by the operating system's settings if the lookAndFeel is set to "Appearance Manager", rather than by the button's font and color properties.

Enabling And Disabling Menu Items
To enable or disable a menu item in a handler, add or remove the (special character, but it is generally easier to use the enable menu and disable menu commands:
enable menuItem 3 of button "My Menu"
disable menuItem 4 of me

These commands add or remove the (special character at the start of the designated line of the button's contents.

Menu Bars On Linux And Windows Systems
A menu bar is made up of a group of menu buttons, with the menuMode property of each button set to "pulldown".

Tip: The Menu Builder can automatically create a menu bar. To use the Menu Builder, choose Tools > Menu Builder.

To create a menu bar by hand without using the Menu Builder:
1. Create a button for each menu, set the style of each button to "menu", and set the menuMode of each button to "pulldown". Set these properties in a handler, or choose Object > New Control > Pulldown Menu to create each button.

2. Put the menu items into each button's contents. In each button's script, create a menuPick handler to perform whatever actions needed when a menu item is chosen.

3. Select the buttons and form them into a group, then move the group to the appropriate position at the top of the window. For Windows systems, set the textFont of the group to empty to use the system font.
put the effective textFont of the mouseControl --currently Segoe UI for windows

Important: The buttons in a menu bar should not overlap. Overlapping buttons may cause unexpected behavior when the user tries to use a menu.

Menu Bars On Mac OS X Systems
To create a Mac OS X menu bar, follow the same steps as for a Linux and Windows menu bar above. This places a group of buttons, each of whose menuMode property is set to "pulldown", at the top of your stack window.

Next, set the menubar property of the stack to the name of the group. This does two things: it displays the menus in the menu bar at the top of the screen, and it shortens the stack window and scrolls it up so the group of menu buttons is not visible in the window. Since the menus are in the menu bar, you don't need to see them in the stack window too.

Important: If the stack has more than one card, make sure the group is placed on all the cards. To place a group on a card, choose Object > Place Group, or use the place command. This ensures the menu bar will be accessible on all cards of the stack, and prevents the stack from changing size moving from card to card.

The Default Menu Bar
If other stacks in the app don't have their own menu bars, set the defaultMenubar global property to the name of the menu group, as well as setting the stack's menubar property. The defaultMenubar is used for any stack that doesn't have a menu bar of its own.

Tip: For a custom menu bar to work correctly inside the development environment, you must set the defaultMenubar to the name of your menu group. This overrides the IDE menu bar. Get the IDE menu bar back by choosing the pointer tool. The Menu Builder sets these properties automatically.

Button Menu References
If the button is a button menu that's being displayed in the menu bar, use the word "menu" to refer to it:
get menuItem 2 of menu "Edit" --same as: get line 2 of button "Edit"

Because menus are also buttons, use a button reference to get the same information. But you may need to specify the group and stack the button is in, to avoid ambiguity. For example, if there is a standard button named "Edit" on the current card, the expression button "Edit" refers to that button, not to the one in the menu bar.

An unambiguous button reference to a menu might look like this:
get line 2 of button "Edit" of group "Menu" of stack "Main" --second menuItem
---
The Layer Of Menu Buttons
For a menu bar to work properly on Mac OS X systems, the menus must be in layer order within the group. The button for the File menu must be numbered 1, the button for the Edit menu must be 2, and so on when creating a menu bar by hand. The Menu Builder does this automatically.

Changing Menus Dynamically
To dynamically change a menu's contents with a mouseDown handler at the time the menu is displayed, place the mouseDown handler in the group's script. A menu button in the menu bar doesn't receive mouseDown messages, but its group does.

The EditMenus Property
When you set the menubar property of a stack to the name of a group, the stack is resized and scrolled up so the part of the window that holds the menus is not visible. To reverse this action to select and edit the menu buttons, set the editMenus property to true. This resizes the stack window so the button menus are again visible to use IDE tools to make changes to them.
set the editMenus of group "Menu" of stack "Main" to "true" --edit button menus

To scroll the stack window again so that the menus are hidden, set the editMenus property back to false.

Special Menu Items
A few menu items on Mac OS X are handled directly by the operating system. To accommodate these special menu items while creating a fully cross-platform menu bar, the File menu, the Edit menu, and the Help menu are treated differently.

By following these guidelines, you can make sure menus will appear properly on all operating systems without having to write special code or create platform-specific menu bars.

The File Menu And The "Quit" Menu Item
On Mac OS X systems, the "Quit" menu item is normally placed in the Application menu (which is maintained by the operating system) rather than in the File menu, as is standard on other platforms. To accommodate this user-interface standard, the Program expects the last menu item of the File menu button to be "Quit". The menu item above it must be a divider line (a dash "-").

The Edit Menu And The "Preferences" MenuItem
On Mac OS X systems, the "Preferences" menu item is also normally placed in the Application menu. To accommodate this user-interface standard, the Program expects the last menu Item of the Edit menu button to be "Preferences". The menu item above it must be a divider line (a dash "-").

The Help Menu And The "About This Application" Menu Item
When the Mac OS X menu bar is set up, it automatically makes the last button the Help menu (regardless of the button's name). The standard Help menu items, such as the "Search" bar are included automatically; you don't need to include them in the Help menu button, and they can't be eliminated from the Help menu.

To accommodate this user-interface standard, the Program expects the last menu item of the Help menu button to be "About". The menu item above it must be a divider line (a dash "-"), and above that must be at least one menu item.

Tip: If the application's user interface is presented in a language other than English, set the label of the File, Edit, and Help menu buttons to the correct translation. This ensures the Engine can find the menu while the menu is displayed in the correct language.

Choosing The Special Menu Items
When the user chooses any of these special menu items, a menuPick message is sent to the button that the menu item is contained in. This ensures that the button scripts will work on all platforms, even if a menu item is displayed in a different menu to comply with user-interface guidelines.

Stack Menus
Button menus can be used for most kinds of standard menus. However, to create a menu with a feature that is not supported by button menus - for example, a popup menu that provides pictures rather than text as the choices - create a menu from a stack.

Creating A Stack Menu
To create a stack menu, create a stack with an object for each menu item. Since the stack menu is a stack and each menu item is an object, the menu items receive mouse messages such as mouseEnter, mouseLeave, and mouseUp.

When the user chooses an item from the stack menu, a mouseUp message is sent to that object. To respond to a menu item choice, instead of handling the menuPick message, place a mouseUp handler in the script of the object.

To create a stack menu that looks like a standard menu:
1. Create a button in the stack for each menu item. The button's autoArm and armBorder properties should be set to true. Or you can choose Object > New Control > Menu Item to create a button with its properties set to the appropriate values.

2. Be sure to set the rectangle of the stack to the appropriate size for the menu. When the menu is opened, the stack will be displayed exactly as it looks in an editable window.

3. Finally, either set the menuName property of a button to a reference to the stack, or place a mouseDown handler containing a pulldown, popup, or option command in the script of each object. When the button or object is clicked, the stack menu appears.

Displaying A Stack Menu
Stack menus can be associated with a button, just like button menus. But when the button is clicked, instead of displaying a menu with the button's contents, a stack is displayed with the behavior of a menu.

To display a stack menu without associating it with a button, use the pulldown, popup, or option command. Normally, you use these commands in a mouseDown handler, so the menu appears under the mouse pointer:

```
on mouseDown  --in card script
  popup stack "My Menu Panel"
end mouseDown
```

Displaying Context Sensitive Menus
There are also several commands to display a context menu. Usually, these commands are used in a mouseDown handler - normally either in a card or stack script
o Popup command: open a stack as a popup menu.
o Pulldown command: open a stack as a pulldown menu.
o Option command: open a stack as an option menu.

Tip: If you set a button's menuName property to the name of a stack, the stack menu is displayed automatically when the user clicks the button. The popup, pulldown, and option commands are only needed to display a stack menu without using a button.

7.13) Searching And Navigating Cards Using The Find Command
The find command can search the fields of the current stack, then navigate to and highlight the results of the search automatically. While it is possible to build such a command using comparison features, the find command provides a complete, pre-built solution. See find in the Dictionary.
find [form] textToFind [in field]

o Form can be one of: normal, character(s) or char(s), word(s), string, whole. Normal is default.
o TextToFind is any expression that evaluates to a string.
o Field is any expression that evaluates to a field reference. If a field is not specified, all fields are searched in the current stack (except fields whose dontSearch property is set to true).

```
find "heart"
find string "beat must go on" in field "Quotes"
```

When the find command finds a match, it highlights the match on the screen - and if necessary navigates to the card that contains the match and scrolls the field so the text is in view. The find command can also be used to return the location of the text that was found.

To reset the find command so that it starts searching at the beginning again:
```
find empty
```

7.14) Using Drag And Drop
There is complete control over drag and drop - both within stack windows and between other applications.
---
o Drag/Drop messages: dragDrop, dragStart, dragEnd, dragEnter, dragLeave, dragMove.
o Drag/Drop functions: dragSource, dragDestination.
o Drag/Drop properties: dragData, allowableDragActions, dragAction, dragDelta, dragImage, dragImageOffset, fullDragData.

Initiating A Drag And Drop

To begin a drag and drop operation, the user clicks and holds the mouse pointer over a selection of text from within an unlocked field. A dragStart message is sent. To allow drags from a locked field or from another object type, in the object's dragStart handler, set the dragData property specifically to the data to drag. When there is a value in the dragData, a drag and drop is initiated when the mouse is clicked and dragged.

Drag And Drop Text

```
on dragStart  --drag text from source field
  if the lockText of me then set the dragData["text"] to line 1 of me  --locked source field
  else set the dragData["text"] to the selectedChunk  --unlocked source field
  pass dragStart  --enable drag
end dragStart
```

Set the dragData, an array, to contain any of the following quoted types of data:
o text - the plain text being dragged.
o HTML - the styled text being dragged, in the same format as the htmlText.
o RTF - the styled text being dragged, in the same format as the RTFText.
o Unicode - the text being dragged, in the same format as the unicodeText
o image - the data of an image in PNG format.
o files - the name and location of the file or files being dragged, one per line.

The example above will enable the user to drag text from a source field to another unlocked field via drag and drop. Dragging from an unlocked source field cuts the text from the source field unless a dragAction is set in the destination. Dragging from a locked source field copies the text from the source field unless a dragAction is set in the destination. Text dragged to a locked destination field is not accepted without dragEnter and dragDrop handlers in the locked destination field.

The mechanics of drag and drop between and within unlocked fields is automatic, no coding is required. See dragData and dragStart in the Dictionary.

Drag And Drop An Object

To drag an object, use the grab command in a mouseDown handler. In this example, a button is dragged to an image. If it doesn't intersect the image's rectangle, the button moves back to its starting location:

```
local sStartLoc  --script local
on mouseDown  --grab a button or image to drag
 put the loc of me into sStartLoc  --start location
 grab me
end mouseDown
on mouseUp
 if intersect(me, image "Tile") then
  set the loc of me to the loc of image "Tile"  --cover over
  answer "I made it!" with "OK" titled "Drag Successful"
 else move me to sStartLoc  --reset
end mouseUp
```

In another example, only objects with a specified name can be dragged:

```
on mouseDown  --in cd script, drag object "Test1"
  if target() contains "Test" then grab target()
  else pass mouseDown
end mouseDown
```

Tracking During A Drag And Drop Operation

Use the dragEnter message in a destination to show an outline around an object or change the cursor when the mouse moves into it during a drag operation.

```
on dragEnter  --show a green outline around the drop target
 set the borderColor of the target to "green"
```

```
end dragEnter
```

If the user attempts to drop data onto a locked field, the drop is blocked. To enable the drop, unlock the field:

```
on dragEnter  --in locked destination field
  if the dragData["text"] <> empty and the lockText of me = "true" then set the lockText of me to "false"
  pass dragEnter
end dragEnter
```

Or to enable the drop on a locked field, use both dragEnter and dragDrop message handlers:

```
on dragEnter  --in locked destination field
  set the dragAction to "copy"  --none, move, copy, link
  pass dragEnter
end dragEnter
on dragDrop  --in locked destination field
  if the text of me = empty then put the dragData["text"] into me
  else put cr & the dragData["text"] after me  --append as line
  pass dragDrop
end dragDrop
```

Use the dragMove message to update the screen whenever the cursor moves during a drag and drop operation.

```
on dragMove  --in field script, change cursor for drop onto link
 if the textStyle of the mouseChunk contains "link" then set the cursor to the ID of image "Drop Here"
 else set the cursor to the ID of image "DontDrop"
end dragMove
```

Use the dragLeave message to remove any outline around an object or change the cursor when the mouse moves out of an object during a drag operation.

```
on dragLeave  --remove outline around the drop-no-longer target
 set the borderColor of the target to empty
end dragLeave
```

For more details, see dragEnter, dragMove and dragLeave in the Dictionary.

Responding To A Drag And Drop

Use the dragDrop message to perform an action when the user drops data onto an unlocked field or another object type.

```
on dragDrop -- check whether a file is being dropped
 if the dragData["files"] is empty then beep 2
 pass dragDrop
end dragDrop
```

Use the dragDestination function to retrieve the long id of the destination object. Use the dragSource function to retrieve the long id of the source object that was the source of the drag.

A dragEnd message is sent to the source field when a drag and drop has been completed. If the source field is unlocked, dragged text is cut from the source field. To prevent this and only copy dragged text, trap the dragEnd message. Place a dragEnd message handler in the unlocked source field but don't pass the dragEnd message to the Engine.

```
on dragEnd  --in unlocked source field, drag copy of text
 --trap dragEnd message
end dragEnd
```

Use the dropChunk function to retrieve the location of text that was dropped in a field. For example, to select the text dropped:

```
on dragDrop  --select text dropped
 select the dropChunk
 pass dragDrop
end dragDrop
```

For more details, see dragDrop, dragEnter, dragDestination, dragEnd, dragSource , and dropChunk in the Dictionary.

Prevent Dragging And Dropping To A Field
To prevent dragging text from a field, trap the dragStart message:
```
on dragStart  --in any field
  --do nothing, trap the message
end dragStart
```

To prevent dropping text into a field during a drag and drop, trap the dragDrop message:
```
on dragDrop -- in destination field
  --do nothing, trap the message
end dragDrop
```

See the entries for dragStart, dragEnter, and dragDrop in the Dictionary.
==========

CHAPTER 8) WORKING WITH MEDIA
Import Image, Import Referenced Image, Import Screen Capture, Create Image, Paint Tools, Export Image, Animated GIF, Vector Graphic Tools, Create Graphic By Script, Video, Player Object, Sound, Audio Clip, Visual Effects, Custom Skins, Blend Modes, Full Screen, Backdrop.

One of the most popular uses for the Program is to create full blown multimedia applications. Even if not creating a traditional multimedia app, many apps require a compelling user interface.

This topic details media support. Image features and capabilities are described: import and export images, manipulate images, working with masks, the clipboard and screen capture, and animated GIFs. Vector graphic features and capabilities are described including manipulate by script. Video and audio features are covered, and how to create custom themed buttons and make visual transition effects.

A wide variety of image formats are supported, including the popular PNG and JPEG formats. PNG images are space efficient and have full support for alpha channels. JPEG images are good for displaying photos. Five image formats can be exported.

o PNG: Export. Supports mask, gamma adjustment, alpha channels, interlacing.
o JPEG: Export. Supports progressive JPEGs; lossy compression. Can set the level of compression.
o GIF: Export. GIF87a and GIF89a; supports animation and interlaced GIFs; maximum 256 colors.
o BMP: Export. Uncompressed.
o PBM: Export. 1-bit (black and white), paint.
o PGM: NoExport. Grayscale.
o PPM: NoExport.
o XBM: NoExport.
o XPM: NoExport. 1-bit (black and white).
o XWD: NoExport.
o PICT: NoExport. Uncompressed.

- Modify images using the paint tools, or manipulate the binary data by script or using an external.
- Create images from any of the native objects, including buttons, fields and graphics, which can be exported in five formats.
- Copy images from the clipboard or export images to the clipboard.
- Capture a portion of the screen, or the entire screen.

8.1) Importing Images
To import an image, choose File > Import As Control > Image File, select an image file to import. This will import the image file into a new image object on the current card. This is equivalent to executing the import paint command in the Message Box.

Tip: To reuse an image throughout an application, for example as part of a custom skin, create a substack and import the images into that, or place all imported images on a card. Then reference them throughout your stack file for the skin, button icons, or images.

8.2) Importing Referenced Images
To reference an image file on disk choose File > New Referenced Control > Image File. This creates an image object on the current card and sets its fileName property to the path of the image file selected. Referenced images do not expand the size of the stack and are only loaded from disk into memory when the user navigates to the current card or otherwise displays them on screen. Referenced images are ideal to update in an image editor and immediately see the changes without having to re-import.

Consider creating a folder next to the stack file to contain referenced image files. Use the Inspector to modify the file path of previously referenced images dragged to the folder. This allows you to move the stack and folder of images together onto a different system.

Use the standalone builder to copy referenced images into a directory (updating each image's fileName property) or to copy referenced images into the stack.

Important: You cannot use the paint tools or manipulate the binary data of referenced images. They must be imported first, modified, then exported.

8.3) Import Using Screen Capture
To import an image by capturing a portion of the screen, choose File > Import As Control > Snapshot. Select the region of the screen to import. To take a screen capture by script, use the import snapshot command. To specify a section of the screen to import without displaying the cross-hairs, use the "from rect" form to create an image object on the current card from the rectangular area specified:
```
import snapshot from "100,100,200,200"  --rect of snapshot
```

8.4) Creating Images
To create an image, drag an image object from the tools palette to the stack. Paint on the image using the paint tools, or set the fileName reference to an image, or manipulate the binary data of the image.

8.5) Using The Paint Tools
To access the paint tools, press the fold out triangle at the bottom right of the tools palette.

Paint Tools
o Select: Drag to select a rectangular area of an image. Shift constraints to a square; ctrl/cmd duplicates selection.
o Bucket: Fills shapes with color. Will fill any pixel connected to the pixel clicked with the brush color; ctrl/cmd-click to fill with transparency.
o SprayCan: Draw an airbrushed color using the brush shape. Ctrl/cmd-click to spray with transparency.
o Eraser: Remove color from an area leaving it transparent. Uses the brush shape.
o Polygon: Draw polygon shape (click polygon to select sides). Shift constrains lines angles to multiples of 22.5°; ctrl/cmd creates transparency.
o Line: Draw a straight line. Shift constrains lines angles to multiples of 22.5°; ctrl/cmd creates transparency.
o Freehand: Draw a freehand curve. If filled option is chosen the beginning and end of the curve are joined automatically; alt/opt prevents drawing line border; ctrl/cmd creates transparency.
o Pencil: Draw a single-pixel-width freehand line. Ctrl/cmd creates transparency.
o Brush: Draw brush strokes using the brush shape. Ctrl/cmd creates transparency; ctrl/cmd-click to magnify.
o Fill Color: Select a color to fill shapes or use with the brush tools.
o Line Color: Select a color to draw lines or use with the pencil tool.
o Brush Shape: Choose a brush shape for use with the brush, eraser, and airbrush tools.

- To magnify an image, right click it with the pointer tool and choose Magnify from the menu.
- After editing an image, it will be re-compressed into the format specified by the paintCompression global property.

Scripting With The Paint Tools
Painting by script can be useful to show the user each paint action. To control the paint tools by script, create an image and choose the paint tool to use. Set the appropriate brush, pattern, or line size. Then use the drag command to paint. The following example creates an image, chooses the brush tool, selects a small circular brush shape, selects a red color, then draws a line:

```
on mouseUp
  set the resizeQuality of the templateImage to "best"  --normal, good, best
  set the rect of the templateImage to "100,100,400,400"  --size of image
  create image
  choose brush tool
  set the brush to "8"
  set the brushColor to "red"  --or an RGB triplet
  set the dragSpeed to "20"  --very slow
  drag from "120,120" to "300,300"  --start point to stop point
end mouseUp
```

see resizeQuality, templateImage, choose, brush, brushColor, dragSpeed, and drag in the Dictionary.

- Reduce the size of an image using the crop command.
- Rotate an image using the rotate command.
- Adjust the quality of the scaling algorithm using the resizeQuality property before setting an image's rectangle.

Manipulating Binary Image Data
To manipulate the binary data of an image, use the image's imageData property. This property returns the color and transparency value of each pixel in the image in a consistent format regardless of the format the image is saved in. The imageData is stored as binary, with each pixel represented by 4 bytes. To convert it to and from RGB values use the byteToNum and numToByte functions.

For example, the numeric value of the red, green, and blue channels respectively for the tenth pixel are given by the expressions:
byteToNum(char (4 * 9 + 2) of the imageData of image <image>)
byteToNum(char (4 * 9 + 3) of the imageData of image <image>)
byteToNum(char (4 * 9 + 4) of the imageData of image <image>)

- To manipulate the binary data of an image using an external, use the imagePixMapID property.
- When you set the imageData of an image, the image will be re-compressed into the format specified by the paintCompression global property.

Rendering An Image From Objects
Use the import snapshot command to create an image from objects or a region of a stack. Instead of specifying a rectangle in global coordinates, specify a stack or object.

Tip: Unlike screen capture, the stack or object imported as an image does not need to be displayed on screen.

To import a snapshot of a region of a stack:
import snapshot from "100,100,200,200" of stack "Layout"

To import a snapshot of an object:
import snapshot from button 5 of stack "Objects"

The import snapshot command creates a new image in the current defaultStack. The image is encoded using the current paintCompression format.

To save this snapshot directly to a file instead of creating an image, use the export snapshot command:
export snapshot from the selectedObject to file "MySnap.jpg" as JPEG

8.6) Exporting Images
To export an image in the current format that it's stored in, put it into a binary file using the URL command. The following example prompts the user to select a file then export the image into it:

```
ask file "Please select a file:"
if it <> empty then put image "MyPicture" into url ("binfile:" & it)  --url is not a function
```

To export an image in a different format, use the export command.
export image "MyPicture" to file "Picture.png" as PNG

```
on mouseUp  --save graphic as jpg
  ask file "Where to save the graphic?" with type "Graphic Files|jpg,png,gif|"
  if it = empty then exit to top  --cancel
  export snapshot from group "MyChart" to it as PNG
end mouseUp
```

You can also export an image to a variable. See export in the Dictionary.

Copying And Pasting Images
To copy an image internally without using the system clipboard, put it into a variable or into another image.
put image 1 into image 2

To re-compress the image in a different format, use the export command to export it to a variable, then put that variable into an image.

To copy an image to the clipboard, use the copy command.
copy image 1

To paste an image from the clipboard, use the paste command.

To transfer the binary data of an image to and from the clipboard, get and set the clipBoardData["image"] property. See clipBoardData in the Dictionary.

8.7) Working With Animated GIFs
Import an animated GIF image in the same way you import other images. Set the repeatCount property to specify how often to play the animation. Setting the repeatCount to "0" pauses the animation, and setting it to "-1" causes it to repeat forever.

To change the current frame, set the currentFrame property.

An animated GIF as a button icon will play simultaneously in each button it is used for.

8.8) Working With Vector Graphics
In addition to bitmap graphics, vector graphics are also supported. Create vector graphics using the graphics tools, or by script. Manipulate them interactively in the IDE or by script. Relayer them, export a description of them, or convert them to bitmap format.

The Vector Graphic Tools
To see the graphic tools, unfold the triangle at the bottom right of the tools palette. The graphics tools are located above the paint tools on the tools palette. The graphic tools operate in the same way as the paint tools, except each time you draw a shape

a new graphic object is created. Unlike paint graphics, you can resize and adjust graphic objects after creating them.

8.9) Creating Graphics By Script
To create graphics by script, set the properties of the templateGraphic, then use the create graphic command.

Manipulating Graphics By Script
Because each graphic is an individual object, manipulate its appearance by setting properties rather than using the drag command (as with the paint tools). You can control all properties of the graphic object by script including the rectangle, line, and fill properties. Change a graphic from one type to another (e.g. a rectangle to an oval) by setting its style property.

The polygon style of graphic has the points property to set the individual points of a line.

Simple motion can be applied using the move command. See move in the Dictionary. For example, to move a graphic:
move graphic 1 relative "100,0" without waiting --100px from left to right in a straight line

To program a more complex animation effect, calculate the changes to the points or rectangles and set these values using timer based messaging. The following example scales a graphic named "Rectangle" down by 100 pixels over the course of 1 second.

```
local sCount  --script local
on mouseUp
  put "0" into sCount
  scaleGraphic  --custom command
end mouseUp

on scaleGraphic  --custom command handler
  add 1 to sCount
  if sCount > "100" then exit scaleGraphic  --escape loop
  get the rect of graphic "Rectangle"
  add 1 to item 1 of it
  add 1 to item 2 of it
  subtract 1 from item 3 of it
  subtract 1 from item 4 of it
  set the rect of graphic "Rectangle" to it
  send "scaleGraphic" to me in 10 milliseconds  --send in time loop
end scaleGraphic
```

8.10) Working With Video
Playback of video is supported with the player object.

On Windows systems, the player object uses DirectShow. Format support depends on which codecs are installed. A list of the file formats and compression types available as standard on Windows is available in the MSDN documentation. The player object on desktop supports: wav, aiff, au, mp3, wma, avi, mpg, wmv. The K-Lite Basic codec expands desktop support to (includes LAV filters): aac, m4a, mp4, mov, flac, ogg. https://www.codecguide.com/download_kl.htm. Choose basic, click small mirror1/2 download link.

On Mac OS X systems, the player object uses the AV Foundation framework which already supports most a/v formats.

On Linux Systems, there is no sound support. The player object can play back video using VLC player or mPlayer. There are some functionality limitations: the alwaysBuffer, startTime, endTime, showController and playSelection properties have no effect, and play step forward/play step back do not work reliably. Install VLC player (recommended) https://www.videolan.org/vlc/#download or mPlayer http://www.mplayerhq.hu/design4/dload.html. VLC has versions to support most Linux distros with easy install. mPlayer download and install is more difficult using Synaptic, Suse, or commandline. CmdLine: 'sudo apt-get install mPlayer'.

There is built-in support for the animated GIF format. Animated GIF files can be played back without 3rd party software. Other formats supported by plugins in web browsers can be played back using revBrowser (e.g. Flash).

8.11) The Player Object
Use the player object to play and interact with video and audio. To create a player object, drag one from the tools palette onto a stack. To select a movie file to play, open the Inspector for the player object and select a file to use as the source. Doing this sets the player's fileName property.

To stream a movie from an Internet server, set the fileName property to the URL address of the stream.

To play a player, use the start and stop commands.
start player 1
stop player 1

Player Properties
o showController: Show or hide the player controller. set the showController of player 1 to "false"
o currentTime: Set the current frame. set the currentTime of player 1 to "1000"
o startTime: The start time of the selection. set the startTime of player 1 to "500"
o endTime: The end time of the selection. set the endTime of player 1 to "1000"
o showSelection: Show the selection in the controller. set the showSelection of player 1 to "true"
o playSelection: Play only the selection. set the playSelection of player 1 to "true"
o playRate: The speed to play the movie. set the playRate of player 1 to "2" --set to "-1" to play backwards
o looping: Cause playback to loop. set the looping of player 1 to "true"
o playLoudness: Set the volume. set the playLoudness of player 1 to "50"
o tracks: List of tracks within the movie. put the tracks of player 1 into tTracksList
o enabledTracks: Enable or disable tracks. set the enabledTracks of player 1 to "3"
o callbacks. A list of messages sent when movie reaches specified time points.
set the callbacks of player 1 to "1000,nextScene"

o duration & timeScale: The duration of the movie and the number of intervals per second of a movie.
put (the duration of me/the timeScale of me) into tRunTime

o alwaysBuffer: Force the player to be buffered, allows objects to be drawn on top and current frame printed (mac/linux only).
set the alwaysBuffer of player 1 to "true"

o currentTimeChanged: Message sent when the current frame changes.
```
on currentTimeChanged pInterval
  put pInterval into field "Time Code"
end currentTimeChanged
```

For more information on these terms, see the Dictionary.

8.12) Working With Sounds
In addition to playing back sound in a wide variety of formats using the player object, there is in-built support for playback of WAV, AIFF and AU format audio clips.

The player object is recommended for playing audio as it supports a wider range of formats and compression types. Use the audio clip when basic audio playback is needed on systems that don't have 3rd party libraries supporting the player object installed.

There are two main ways to access an audio file: reference the sound by its file path, or import the sound directly into a stack. The reference method is best with many large sound files. The import method is best with small sound files and prevents the file

from getting lost or separated from the stack, but imports can't use a player object. Imports don't use a file path and must be wav, aiff, au.

Importing An Audio Clip
To import an audioClip choose File > Import As Control > Audio File. This will import the selected audio file into the current stack as an audioClip. The audio file must be one of .wav, .aiff, or .au format.

Playing An Audio Clip
To play an audio clip:
play audioClip 1

To stop playing an audio clip:
play stop

To set the volume:
set the playLoudness of audioclip 1 to "50" --volume to 50%

8.13) Working With Visual Transition Effects
Visual transition effects are supported when changing cards or hiding and showing objects. Use the visual effect command to display a visual effect.

To go to the next card with a dissolve visual effect:
visual effect dissolve slow --don't quote effect
go next card

To make changes to objects on the screen with a visual effect (e.g. hide, show, or move them):
lock screen for visual effect
hide image 1
show image 3
unlock screen with visual effect wipe right

For a list of visual effects, see visual effect in the Dictionary.

8.14 Creating Custom Skins
In addition to support for system native objects, you can create an entirely custom look, or skin, for an application. All of the built-in elements can be replaced with themed graphics to create rich multimedia. For example, buttons support "icons" of unlimited size to replace the native appearance of a button. Stack windows can be irregularly shaped and even contain holes and alpha mask (variable) transparency. All objects support a variety of transfer modes or inks. Stacks can take over the entire screen or be displayed with a backdrop.

Custom Themed Buttons
To create a custom button, first create a regular button. Open the Inspector for the button and set its style to Transparent and turn off the Show Name property. Next, switch to the Icons tab in the Inspector. Turn off the Hilite Border property. Click the Icon square to select an imported image to use as an icon for the button. To set the mouseDown state, set the hilite icon. To set the roll over state set the hover icon.

Tip: To use the same set of images as icons throughout a stack file, create a substack or card and import all theme images into it. Set the ID property of each image to a high value (between 10,000 and 100,000) to make them unique. You can now reference that image for a button icon anywhere in the application. To change themes, just replace an image with another image and give it the same ID.

Tip: Any object can behave like a button. For example, to create a graphic that responds when a user clicks on it, create the graphic and add a mouseUp handler to its script, the same as a button.

Irregular Windows
To create an irregularly shaped window, import or create an image that has a transparency mask. Then use the Stack Inspector to choose that image as the stack's Shape. To change the shape by script, set the stack's windowShape property. Many modern window managers support alpha blended windows (variable degrees of transparency). To create a window with an alpha channel, import a PNG image that contains an alpha channel and set the windowShape of the stack to this image.

8.15) Blend Modes - transfer modes or inks.
Blend modes determine how an object's colors combine with the colors of the pixels underneath the object to determine how the object's color is displayed.

To set the blend mode of an object, use the Blending tab in its Inspector or set the object's ink property. All objects support blend modes, with the exception of stacks. See ink in the Dictionary.
```
set the ink of image "Picture" to "blendBurn"
```

To set the degree of transparency of an object, set the object's blendLevel property. All objects (including stacks) support blendLevel:
```
set the blendLevel of button 1 to "50"  --50% transparent
```

8.16) Full Screen Mode
A stack can be displayed as full screen by setting its fullScreen property to true:
```
set the fullScreen of this stack to "true"  --false to exit full screen mode
```

To hide or show the menu bar on Mac OS X, use the hide menubar or show menubar commands:
```
hide menuBar
show menuBar
```

Similarly, use the hide taskbar and show taskbar on Windows systems to hide and show the taskbar.

8.17) Displaying A Backdrop
To display a backdrop set the backDrop global property to a solid color or the ID of an image:
```
set the backDrop to "black"
```

To remove the backDrop:
```
set the backDrop to none
```

==========

ADVANCED CONCEPTS

CHAPTER 9) PRINTING AND REPORTS

Intro, Printer Settings, Printer Dialogs, Paper Options, Job Options, Print Card, Print Field, Print Text, Complex Layout, Multiple Pages, Scrolling Field, Print Range, Print Browser.

9.1) Introduction To Printing
Printing is a vital aspect of many applications. A comprehensive set of printing capabilities is provided. Whether printing out a stack, labels, or producing complex reports, it has the features needed.

A number of printing methods is supported. Use the print card command and have the Engine manage the layout of cards on the paper. Alternatively use the print into rectangle commands to take full control over the layout. The former method is most suited to implementing simple prints, while the latter is for complex layout printing or printing reports. Finally, use the built-in field printing commands to print the contents of any text field.

Also included is a full set of features to access and set printer device options, including options such as margins, page range, and number of copies. On Mac OS X systems print directly to a PDF file - even without showing the user a print dialog. On Windows systems print directly to an XPS file (or PDF file with 3rd party software), and on Linux systems a Postscript file. This feature is invaluable to produce a high resolution PDF, XPS, or Postscript file from a stack.

9.2) Controlling Printer Device Settings
There is full code control over the user's printer device and any available printer settings.

Choosing A Printer
Use the availablePrinters function to list the printers available on the user's system. Printers can include fax modems and networked devices. If the availablePrinters returns empty, no printer is assigned. For example, to place a list of the available printers into a list field:
put the availablePrinters into field "ListOfPrinters" --line list

Set the printerName property to the printer to use to print. Use any printer listed in the availablePrinters function. This property is useful when producing an in-house utility that needs to print to a specific printer on the corporate network, or for automatically restoring the user's previous printer choice stored in a preferences file.
set the printerName to the cSavedPrinter of stack "My Preferences" --custom property of stack

The printerFeatures property provides a list of the features supported by the currently selected printer. Features will vary widely from device to device, but typical features may include things such as "collate", "color" and "duplex". Use this property to enable and disable output options in any custom printer settings dialog.

Choosing Output Mode (e.g. Print to File)
The printerOutput global property chooses the output mode for subsequent printing commands. This property is set by the system to the default printer on startup and can be changed with a system print dialog in which the user chooses a different printer. If this property is set to a device it will output to the physical printer. Alternatively, you can set it to a full file path to print to a file. On Mac OS X this will create a PDF file, on Windows an XPS file, and on Linux a Postscript file. On Mac OS X, there is a preview option. For example, to save the current card to a file:

```
ask file "Save as:
set the printerOutput to ("file:" & it )  --output to PDF, XPS, or Postscript
print this card
```

9.3) Working With Printer Dialogs
In most applications that need to print, provide a way to bring up the standard OS Print and Page Setup dialogs. Typically these dialogs are available from Print and Page Setup menuItems in the File menu of your application. When the user makes changes in these dialogs, the changes are made accessible to you in global properties. It is not necessary to bring up the standard OS dialogs to alter printer settings. The appropriate printer settings can be altered directly by code instead.

Note: On Linux systems a recent version of GTK must be installed to display the OS printer dialog. If not installed, the Engine will display its own printer dialog which has been built-in as a stack and script library. This dialog mimics the standard system printer dialog and sets the printing global properties directly.

Tip: Force the Engine to use its own internal print settings dialog by setting the systemPrintSelector global property to false. Advanced users may customize this internal printer dialog by running toplevel "print dialog" or toplevel "page setup" in the Message Box. Save a copy of the stack as it will be overwritten each time you upgrade this program. The print and page setup dialogs must be included in a standalone application to use them. Ensure that the check box Print Dialog is turned on in the Standalone Application Settings dialog for the application. You do not need to include these dialogs if using the OS native print dialogs.

To bring up the standard OS printer dialog, use the answer printer command. If the user does not press the cancel button then any changes to the printer settings will be reflected in the global printing properties.

answer printer

To bring up the standard OS page setup dialog, use the answer page setup command.
answer page setup

Saving Printer Settings
To save or set a complete set of options relating to the current printer, which includes every setting in the OS Page Setup and Print dialogs, use the printerSettings global property.

The printerSettings property is a binary string that completely describes the current settings. The property contains the name of the printer and the settings currently in use.

Caution: Do not attempt to modify the printerSettings, but rather get and set it in its entirety. To access individual printer properties, use the global printing properties described below.

When setting the printerSettings property to a saved value, the Engine will choose the printer named in the property and set all of its settings to those contained within the property. If the printer cannot be found, the error "unknown printer" will be returned in the result. If the printer is found but the settings are not valid then the Engine will choose the printer and reset it to default values.

Note: Save a separate copy of the printerSettings property for each printer or OS you intend to use. The printerSettings property cannot be transferred between platforms. For example, a printerSettings property generated on a Windows computer cannot be used on Mac OS X - even for the same printer. To alter settings across different platforms and printer types use the global printing properties described below.

Use the printerSettings property for convenience when using the same printer and printer settings, or are setting esoteric properties not listed in the global printing properties described below.

To save the current printer settings into a custom property stored on the current stack:
set the cSavedPrinterSettings of this stack to the printerSettings --custom property of stack
save this stack

Then to restore these settings at a later time:
set the printerSettings to the cSavedPrinterSettings of this stack
---

9.4) Paper Related Options
This section discusses how to get and set paper related options - the rectangle area of the paper, the paper size, the orientation, and the scale used to print onto paper. These paper options apply to all types of card, field, and layout printing.

PrintRectangle
Use the printRectangle property to get the printable rectangle area within the paper (returned in device co-ordinates). This property takes into account any settings applied by the user in the Page Setup and Printer dialogs including the print orientation (landscape or portrait). The rectangle is represented left,top,right,bottom and is always relative to the top left of the page - thus the top left will always be 0,0. The printRectangle will always be within the printPaperRectangle - the rectangular area of the sheet of paper.

Note: The printRectangle property is read only and cannot be set directly. To alter it you must set other options relating to the paper, for example the
printPaperOrientation.

Important: Do not confuse the printMargins and other card layout printing properties with paper properties such as the printRectangle. The printMargins only applies to printing cards using the automatic card layout capabilities. Thus the printMargins has no effect on printRectangle.

PrintPaperOrientation
Use the printPaperOrientation property to get and set the orientation of the print out. This property may be set to one of the following values:
o portrait: rotated 0 degrees.
o landscape: rotated 90 degrees clockwise.
o reverse portrait: rotated 180 degrees clockwise.
o reverse landscape: rotated 270 degrees clockwise.

set the printPaperOrientation to "landscape"

PrintPaperScale
Use the printPaperScale property to apply a scale factor to a print out after all other layout and scaling options. For example, if a printing layout features printing a series of cards at 50% scale, then sets the printPaperScale, this factor will be applied to the entire layout after the card layout scaling has been calculated.

To print between 1% and 100%, set the printPaperScale to a number between 0 and 1. To print at 200%, set the printPaperScale to 2.
set the printPaperScale to 0.5 --50%

9.5) Job Related Options
This section discusses how to get and set job related options - the number of copies, duplex printing, collation, color, title, and printable area.

Important: The available job options all depend on what the currently selected printer supports (use the printerFeatures property to retrieve a list of features supported by the current printer).

PrintCopies
Use the printCopies property to get and set the number of copies to print. The printCopies should be set to a value of 1 or more.
set the printCopies to 5 --print 5 copies

PrintDuplex
Use the printDuplex property to tell the printer to print double sided. This property may be set to any of the following values:
o none: no double-sided printing.
o short edge: double-sided printing with tumble (flip the non-facing page).
o long edge: double-sided printing without tumble.
set the printDuplex to "short edge"

PrintCollate
Use the printCollate property to specify whether to interleave multiple copies of a print job. If a print job has three pages, P1, P2, and P3:
set the printCollate to "true"
set the printCopies to "2" --output = P1,P2,P3,P1,P2,P3

set the printCollate to "false"
set the printCopies to "2" --output = P1,P1,P2,P2,P3,P3

PrintColors
Use the printColors property to specify whether to print in color or not. If "color" is not among the lines of the printerFeatures then this property will have no effect and all print jobs will be printed in monochrome. PrintColors may be set to either true or false. For example, to check if color printing is supported on the current printer and use it if it is:
if "color" is among the lines of the printerFeatures then set the printColors to "true"

PrintTitle
Use the printTitle property to specify the name of the next print job in the system printer queue. Setting this property to match

the name of the user's document will ensure that the user is able to recognize it in the system printer queue utility. If the printTitle is empty at the start of a printing loop, the title of the defaultStack will be used.
set the printTitle to "My Report 1"

PrintRectangle
Use the printRectangle property to determine the printable region of the physical page as returned by the printer. This rectangle will always be contained within the printPaperRectangle. Thus use the printRectangle and not the printPaperRectangle when calculating a print layout. The printPaperRectangle is useful if generating a print preview and want to show the entire area of the paper including any margin areas that cannot be printed on. This property is read only and cannot be set directly.

Printer Font Metrics (Windows)
Windows systems sometimes use different versions of the same font for displaying text on screen and printing. This can result in layouts and line breaks differing between the screen display and the printed output. To prevent this from happening, you can tell the Engine to use the printer fonts for display on screen. To do this, set a stack's formatForPrinting property to true.

Do:
o Set the formatForPrintingstack property to true before loading a stack in memory. If the stack is already loaded, set this property to true then save and reload it. (Save then use Close and Remove from Memory in the File menu).
o Create a stack off screen (with formatForPrinting set to true) with your print layout template and copy text into it prior to printing.
o Set the formatForPrinting to true before doing any print layout related calculations on the stack.
o Set the formatForPrinting to true on any print preview stack being displayed to the user.

Don't:
o Don't allow the user to directly edit text in fields whose formatForPrinting is set to true. Attempting to do this may cause display anomalies. Set this property to false and reload the stack first.
o Don't use stacks with formatForPrintingset to true for display on screen, as this will show text that has been optimized for print display (instead of screen display), which is harder to read on screen.
o Don't use this property on other platforms - Windows is the only platform that uses different font versions for screen and print.
o Don't use the windowBoundingRect property to constrain display of a stack who's formatForPrinting has been set to true - this property will be ignored when the stack is opened or maximized.

9.6) Printing A Card
After setting the printer, paper, and job options, use one of the print commands to start printing. The print card command prints a card. See print in the Dictionary.
print this card --print the current card
print card 12 --print card 12, even if not current card

To print a scale between 1 and 100%, set the printScale property to a number between 0 and 1. To print at 200%, set the printScale to 2.

Important: The printScale applies to each card printed. It is not related to the printPaperScale which is applied to the entire print job after all other scaling calculations have been applied.

When printing a card, use the printMargins property to specify the margins around the border of the card on the page. When calculating placement on the printed page, all calculations assume there are 72 dots per inch - regardless of platform or printer device. The Engine will automatically adjust the print out for resolution of the actual device. This makes it easier to calculate the printed layout:
set the printMargins to "36,36,36,36" --L,T,R,B 1"=72px, .5"=36px, margins around border of card

Important: printMargins only applies when using print card directly. It does not have any effect on printing cards into a layout. The printCardBorders property specifies whether or not the bevel border around the edge of a card should appear in the print out.

Card Layout Options
When using the basic print card form of the print command, there are two layout options that can customize the positioning of cards on the printed page. For further flexibility, see the section on printing a layout, below.

Use the printRowsFirst property to specify whether cards should be printed in rows across and down, or in columns down then across. The following example is useful for printing labels.

1. Create a stack and size it to be small - the size of a mailing label.
2. Create a single field, and in the field Inspector turn off the Shared Text property.
3. Group the field and in the group property Inspector turn on Behave As Background.
4. Enter text in the field for the first mailing label - name, address, city, state, zip.
5. Create eight more cards, and in each field enter text for a different mailing label.

Now implement the printing commands. If this was a real application the printing commands could be put in a File > Print menuItem. For this example, execute the following commands in the multi-line Message Box (open the Message Box then press the second icon to get the multi-line pane), and run with ctrl/cmd-returnKey.

o Use the printRowsFirst property to specify printing columns instead of rows.
o Use the printGutters property to specify the margin between each card, default = "36,36" or .5 inch horizontally and vertically.
o Use the printCardBorders property to specify printing a border around each card.

```
answer printer  --choose printer options
set the printRowsFirst to "false"  --columns
set the printGutters to "18,18"  --.25 inch gutter
set the printCardBorders to "true"  --border
print 9 cards
```

9.7) Printing Fields & Text
RevPrintField
Use the revPrintField command to print a field. This command takes a single parameter, a reference to the long id or long name of a field. This command only allows printing a single field. To include a header and footer or text constructed by code, see the revPrintText command below.

```
revPrintField the long id of field "TextDocument"
```

Tip: revPrintField is implemented as a script library located in the IDE. The script library creates an invisible stack, sets the rectangle of that stack to the current paper size, sets the formatForPrinting to true, creates a field, then copies the contents of the field you specify into this invisible stack. It then prints the field one page at a time, scrolling the text after each page. Advanced users can locate this library script by going to the Back Scripts tab in the Message Box, turning on the checkbox for Show UI Back Scripts, then editing the script of button "revPrintBack". The revPrintField handler is near the top of the script.

RevPrintText
Use the revPrintText command to print plain or styled text together with an optional header and footer and font.
revPrintText textToPrint [,headerText [,footerText [,fieldTemplate]]]

o textToPrint is anything which evaluates to a string. To print styled text, use HTML instead of plain text. (To convert a field containing styled text to a HTML use the htmlText property.)
o headerText and footerText contains the text to use as a header and footer. You may include an expression that is computed for each page. For more details on using expressions, see revPrintText in the Dictionary.
o fieldTemplate specifies a long id or long name field reference whose font will be used for printing.

```
revPrintText tMyReport,tab&"June Report",,the long id of field "Font"  --centered header, no footer, font
revPrintText (the htmlText of fld "Info"),"Information",the time && the date  --header, footer, inherited font
```

RevShowPrintDialog
Use the revShowPrintDialog command to control whether the system printer and page setups dialogs should be shown by revPrintField or revPrintText.
revShowPrintDialog true, false --show system printer dialog, not page setup dialog
revPrintField the long id of field "text document"

Printing A Report
For example, to print a report as styled or plain text, to pdf or paper, use the revPrintText command. In stack "Report Weekly" place scrolling field "Report" and button "Print Report". Enter report text in the field. Edit text styles as desired. In the Message Box:
set the tabStops of field "Report" to "120" --for tabbed data
set the fixedLineHeight of field "Report" to "false" --for styled text

In the button "Print Report" place the following script:

```
on mouseUp
  local tChoice, tReport
  answer "Print a report in styled text or plain text?" with "Cancel" or "Styled" or "Plain" titled "Print Report"
  if it = "Cancel" then exit to top
  put it into tChoice
  revShowPrintDialog true, true  --show system printer dialog, page setup dialog. Print to pdf or paper.
  if the result = "Cancel" then exit to top
  if tChoice = "Styled" then
    put the long date &cr& the htmlText of fld "Report" into tReport
    revPrintText tReport,,tab& the time &","&& the date, the long id of fld "Report"  --no header, centered footer, font
  else if tChoice = "Plain" then
    put the long date &cr& fld "Report" into tReport
    revPrintText tReport,,tab& the time &","&& the date,the long id of field "Report"  --no header, centered footer, font
  end if
end mouseUp
```

See revPrintField, revPrintText, and revShowPrintDialog in the Dictionary.

9.8) Printing A Complex Layout
To print a more complex layout than allowed with the basic print card command or text printing commands, use the print card into rect variant to create any sort of layout.
print card from topLeft to bottomRight into pageRect

o topLeft is the top left coordinate of the current card to start printing at.
o bottomRight is the bottom right coordinate of the current card to stop printing at.
o pageRect is the rectangular area on the paper to print into.

Important: printMargins only applies when using the print card command directly. It does not have any effect on printing cards into a layout. Use the printRectangle property to get the printable area when working with layout printing.

For example, to print a text field and have the output scale to take up the entire width of the paper and the top half the height:
put "0,0" &","& item 3 of the printRectangle &","& round(item 4 of the printRectangle / 2) into tPageRect
print this card from the topleft of field "MyText" to the bottomRight of field "MyText" into tPageRect
---
Print Multiple Stack Components
Construct a complex layout taking components from multiple stacks by printing a sequence of rectangles onto the same page. For example, from one stack that contains a standard header and footer, another stack that contains a logo, and a third stack that contains text.

Use the open printing command to start a print job, print each component into the appropriate rectangle on the paper, then use the close printing command to send the print job to the printer. For example, combine the components of two stacks with printable regions onto a single sheet of paper:

```
answer printer  --show system print settings dialog
open printing  --start a print job
set the defaultStack to "MyHeader"  --already opened stack with a header
print this card from the topLeft of field "Header" to the bottomRight of field "Header" into the rect of field "Header"
put the bottom of field "Header" into tHeaderBottom  --save bottom coordinates
set the defaultStack to "ReportEditor"  --already opened stack with text
--print the table field below the header:
print this card from the topleft of field "ReportTable" to the bottomRight of field "ReportTable" \
into 0, tHeaderBottom, the right of field "ReportTable", (the bottom of field "Report Table" + tHeaderBottom)
close printing  --send print job to printer
```

9.9) Printing Multiple Pages

Multiple Pages When Card Printing

To print multiple pages when card printing, specify which cards to print as part of the print command.

print {range}

Examples:

```
print this card  --print current card
print all cards  --print all cards of current stack
print 10 cards  --print next 10 cards starting with current card
print card 3 to 7  --print card 3 to 7 of current stack
print marked cards  --print all cards where the mark property is true
```

Multiple Pages When Layout Printing

To print multiple pages when layout printing, use the open printing command to open a print job. Print the layout for the first page. Next use the print break command to insert a page break into the print job. Then print the layout for the second page, and so on. Finally, use the close printing command to send the print job to the printer.

9.10) Working With Scrolling Fields When Layout Printing

To print a single scrolling text field, use the revPrintText command. To incorporate the contents of a scrolling field within a layout, use the pageHeights property to scroll the field each time you print a page, then print break to move to the next page.

The pageHeights property returns a list of values to indicate how far a scrolling field needs to be scrolled to avoid clipping a line of text on each page printed out. (Use this feature in conjunction with the formatForPrinting property.)

```
put the pageHeights of field "Body Text" into tHeightsList  --line list of pageHeights
set the vScroll of field "Body Text" to "0"  --scroll to top of field
set the vScrollbar of fld "Body Text" to "false"  --hide
set the showBorder of fld "Body Text" to "false"  --hide
set the borderWidth of fld "Body Text" to "0"  --hide
open printing  --start print job
repeat for each line i in tHeightsList
  set the vScroll of field "Body Text" to the vScroll of field "Body Text" + i  --scroll down in chunks
  print this card from the topLeft of field "Body Text" to the bottomRight of field "Body Text"  --option: into tHeaderRect
  print break  --new page
end repeat
close printing  --send print job to printer
set the vScroll of field "Body Text" to "0"  --scroll to top of field
set the vScrollbar of fld "Body Text" to "true"  --show
```

```
set the showBorder of fld "Body Text" to "true"  --show
set the borderWidth of fld "Body Text" to "2"  --default
```

Tip: You can incorporate scrolling fields into a template print layout stack to make it easier to manage printing a complex layout. Create a field in a template print stack, turn off its vScrollbar and set its borderWidth to 0. At the start of each print job, copy the text font and size using the textFont and textSize properties, then copy the contents of the text field to print using the htmlText property.

9.11) Working With Print Ranges
Use the printRanges property to get a list of pages the user has selected in the printer settings dialog. Use this property when printing an extremely complex layout to avoid printing pages the user has not selected. To use this property, open a system printer dialog and store the printRanges in a variable. Next set the printRanges to "all". Then send only the pages that were selected (stored in the variable) to the printer.

If you ignore the printRanges property the Engine will handle this setting automatically. Send every page to the printer as normal and the Engine will ignore the pages the user has not selected in the print dialog. Handle this option manually only if printing an extremely complex layout and want to save processing time building the layout for every unselected page.

Use the printPageNumber to get the number of the page currently being printed during a printing loop.

9.12) Printing A Browser Object
To print the contents of a browser object, use the revBrowserPrint command. See revBrowserPrint in the Dictionary.
==========

CHAPTER 10) DEPLOY AN APPLICATION
It's easy to deploy your application to anyone. Using the standalone building capability, create a native application for each desktop platform: Windows, MacOS X, and Linux. Users can run these applications like any other application they download and install. Standalones can have their own identity as true applications, include icons and document associations.

10.1) Building A Standalone Application
After finishing a stack, distribute it others without the Program by building it into a standalone. The full feature set of a stack is built into a standalone, with the exception of coding scripts of objects.

The standalone builder can build standalones for any desktop platform, from any desktop platform. For example, build a Windows standalone on a Mac OS X machine. However, check the standalone looks and behaves correctly on each platform deployed to. It is harder to debug an standalone, so test the stack thoroughly before building a standalone.

Step By Step
Deploying a stack to a standalone is straightforward:
1. Open the stack in the IDE.
2. Select File > Standalone Application Settings.
3. Select the settings for the standalone.
4. Make sure that checkboxes for each platform to deploy to are checked.
5. Close the standalone settings dialog.
6. Select File > Save As Standalone Application.
The standalone will be packaged up and placed in the selected output folder.

Standalone Applications Settings
Choose File > Standalone Application Settings to create settings for a standalone. The settings entered are applied to the current front most editable stack and are saved with the stack. This means you only need to enter the settings once for each platform deployed to. The same settings will apply with a future build.

General Settings
o Mode Selector: Choose between the different standalone application settings screens.
o Standalone Name: Set the name of the finished standalone. Don’t include file extensions (.exe on Windows or .app on Mac OS X) as the standalone builder can create standalones for multiple platforms and will add the appropriate extension automatically.
o Inclusions Selector: Choose the components to include in the standalone. Choose to search for required inclusions automatically (default), or manually select the components to include.

Search For Inclusions
This is the default option. When selected, the Engine will search the stack file (mainstack and substacks) to attempt to determine what components the stack uses. It will then include those items.

Select Inclusions
Select this option to specify the components to include manually. Use this option if the stack dynamically loads components that cannot be searched for at this point automatically, or if you know exactly what components the stack uses and want to speed up the standalone building process by skipping the automatic search step. If necessary inclusions are missing, the standalone may fail. If custom error reporting or the Program's standalone error reporting dialog are not added, such a failure may be silent - an operation in the standalone will stop working without notifying the user.

Profiles Settings: Choose between the Property Profile settings options. Only alter settings in this area if you have used Property Profiles.
o Remove all profiles: Remove all profiles and builds the standalone using the currently active profile on each object. The default selection. Select this option if you don't need to change profile in the standalone and want to save disk space by removing extraneous profile information.

Stacks Settings
o Add Stack Files: Use this section to add additional stack files to the standalone. Any stacks added to this section will be added to the stackFilesproperty of the main stack of the standalone. This means that any scripts within the standalone will be able to locate and reference these stacks by name.

o Advanced Options: Use this section to control exactly how multiple stack files are managed in the standalone.

o Move Substacks Into Individual Files: If you select this option, each of the substacks in the stack files selected will be moved into their own individual stack file, located in the data folder or within the application bundle of the standalone.

o Rename Stackfiles Generically: Rename each substack file using a number on disk (instead of using the name of the substack). Select this option if you do not want the names of stacks to be visible to the end user in the filing system.

o Create Folder For Stackfiles: Create a folder and place the stack files into that folder, instead of storing them at the same level as the standalone executable. All references in the stackFiles property will refer to this folder using a relative path so the stacks can still be located by the standalone.

o Individual Stack Options: Select a stack file on the left then an individual stack from within the file to set options on that stack.

o Set DestroyStack To True: Set this option have the selected stack file to be removed from memory when it is closed. This option is useful when loading large stacks into memory and want them removed when closed.

o Encrypt With Password: Secure the scripts within the selected stack file with a password. This provides a basic level of encryption that prevents someone from casually reading the scripts in the stack by opening the file in a binary file viewer.

Important: A stack file directly attached to a standalone application cannot have changes saved to it. This stack is bound directly to the executable file that runs. The OS locks an executable file while it is running. To save changes in a standalone, split the stack up into multiple files. A common technique is to create a "splash" stack that may contain a welcome screen and then loads the stacks that make up the rest of the application. These stacks are referenced as stackFiles on this pane in the standalone settings dialog. It is thus possible to automatically update these component stacks, or to save changes to them. You may also create preference files in the appropriate location on the end user's system. See specialFolderPath in the dictionary.

Copy Files Settings
o Non-Stack Files In The Application: List non-stack files to be included in the standalone. Use this feature to include help documents, read me text files, and other resources to include with the standalone each time you build.

o Copy Referenced Files: Loop over all image and player objects in stacks and copies any files referenced in the fileName property of these objects into the standalone. Then automatically sets the fileName property to reference these files in the standalone using referenced file paths.

o Destination Folder: Create a subfolder within the standalone to copy the image and movie files to.

Additional Resources Settings
The following built-in resources are available by default:
Ask Dialog, Answer Dialog, Browser, Browser (CEF), Brushes, Cursors, Database, Magnify, PDF Printer, Print Dialog, SSL & Encryption, XML.

The following script libraries are available by default:
Data Grid, Geometry, Internet, Printing, Zip, Speech, Table, XMLRPC.

The following database drivers are available by default:
ODBC, MySQL, SQLite, PostgreSQL.

MacOS X Deployment Settings
o Build For Mac OS X 64-bit: Build a standalone that includes a 64-bit slice.
o Application Icon: Choose an application icon to represent the standalone in the Finder, in icns format.
o Document Icon: Choose a document icon to represent the standalone's documents in the Finder, in icns format.
o Icons For Ask/Answer Dialogs: Choose an icon to display whenever the ask or answer commands are used to display a dialog. On Mac OS X, the convention is that these dialogs should display the standalone icon. The icon should be stored in the stack as an image, or selected from the Program's built-in icons. If using a built-in icon, select the relevant inclusion on the General tab if selecting inclusions manually.

PLIST
o Enter Information And Have The Engine Write The PLIST: Have the Engine fill out the PLIST for the standalone automatically. The PLIST is a settings file stored in XML format stored as part of every Mac OS X application. It contains information about the standalone, including its name, version number, copyright notice, and document associations. Having the Engine create this file is the recommended option.

o Choose A File To Import Into The Application Bundle: Choose to import a PLIST file instead of having the Engine create one. Select this option if you have created your own highly customized PLIST to use for the standalone in each build.
o Short Version/Long Version: The version information to include with the standalone.
o Get Info String: The visible text displayed in the standalone's Get Info window by the Finder.
o Copyright Notice: The copyright notice for the standalone.
o Bundle Identifier: A unique identifier for the standalone used by Mac OS X to identify the standalone.

Windows Deployment Settings
o Build For Windows x86: Build a 32-bit standalone for the Microsoft Windows OS.
o Build For Windows x86_64: Build a 64-bit standalone for the Microsoft Windows OS.
o Application Icon: Choose an application icon to represent the standalone in Windows, in .ico format.
o Document Icon: Choose a document icon to represent the standalone's documents in Windows, in .ico format.
o Version Information: The version info stored as part of the standalone and displayed in the Windows property inspector and dialogs.
o UAC Execution Level: Select the user account control level that applies to the standalone. For more info, consult MSDN.

Linux Deployment Settings
o Build For Linux: Build a standalone for 32-bit Linux.

o Build For Linux x64. Build a standalone for 64-bit Linux.
o Include: Select built-in dialogs to include. These dialogs are useful if the standalone may run on a system that does not include these dialogs as part of the OS. These dialogs are not needed if running a recent version of GTK.

Android Deployment Settings
o Build For Android armv7: Include armv7 binaries in the Android standalone.
o Build For Android arm64: Include arm64 binaries in the Android standalone.
o Build For Android x86: Include x86 binaries in the Android standalone.
o Build For Android x86_64: Include x86_64 binaries in the Android standalone.
o Label: The application name.
o Identifier: The application identifier for the standalone. This should begin with a reverse domain name of a domain you own.
o Version Name: The version number of the standalone.
o Version Code: The build number of the standalone.
o Icon: The standalone icon.
o Splash: The standalone splash screen (applicable to the Educational Personal license only)
o Signing; Choose to sign for development, with your own key or not to sign.
o Key; Choose a Java keystore file to sign with.
o Install Location: Allow installation onto external storage.
o Custom URL Scheme: Add a custom url scheme to the standalone to launch when a url is opened that uses the scheme.
o Push Sender ID: The project number from Google's Cloud Messaging API. See lesson Push Notifications With Android.
o Status Bar Icon: The icon shown in the status bar for a notification.
o Hardware Accelerated: Enable hardware acceleration of the standalone.
o Initial Orientation: The orientation on the device when the application initially launches.
o Status Bar: Set the visibility of the status bar.
o In App Purchasing: Specify the store used for the in-app purchasing API.
o Minimum Android Version: Choose the minimum version of Android the standalone should run on.
o Choose: Whether required, prohibited, or not applicable to function. Camera, Camera Autofocus, Camera Flash, Front Camera, Accelerometer, Telephony, Telephony CDMA, Telephony GSM, Fake Touch, Touchscreen, Multitouch, Multitouch Distinct, Multitouch Jazzhand.
o Request Permission To: Write External Storage, Internet Access, Camera Access, Read Contacts, Write To Contacts, Read NFC Tag, Fine Location, Course Location, Vibration, Idle Timer, Ad Support.

Testing A Simulated Android Deployment
Step By Step
Testing a simulated android deployment is straightforward:
1. Open the stack in the IDE.
2. Select File > Standalone Application Settings.
3. Select the settings for the Android standalone.
4. Close the standalone settings dialog.
5. Choose Development > Test Target > YourTargetName.
This opens a cascading menu to select a connected Android device or simulator to use when simulating deployment for testing.
6. Click the Test button or choose Development > Test.
This simulates Android deployment of the stack to the selected test target.

Bug Report Settings
o Include Error Reporting Dialog: Include an error reporting stack in the standalone. Select this option if testing the standalone and want details of any errors in it, or if not including your own error reporting routines.

o HTMLText For Dialog: The text to display to the user in the dialog that comes up when an error is encountered. This text should be in compatible HTML format. Create and format the text in a field then copy the field's HTMLText property.

o Dialog Icon: The icon to display in the error dialog. This should be stored as an image in the stack.
o Allow User To Enter Comments: Display a box for the user to give more information. This info will be included in the report.
o Allow User To Save Report To File: Allow the user to save the report to a file. Select this option for users to send an error report to you.

o Allow User To Email Report: Allow the user to email the report. Select this option for users to email an error report to you. This option loads the system default email client and populates the email with the contents of the error report and the user's comments. The To: field is sent to the email address specified in the email address field.
==========

CHAPTER 11) ERROR HANDLING AND DEBUGGING

Error Dialog, Suppressing Errors, Output Info, Debugger, Variable Watcher, Message Watcher, Custom Error Handling, Standalone Errors.

In an ideal world everyone would write perfect code and there would never be any need to debug. However in reality, virtually every project is going to require some degree of debugging. The more complex the project the more likely this is to be true. Fortunately, there's a full plethora of debugging tools and techniques available that make it easier to track down errors. The live run-edit cycle allows you to see the effect of corrections as soon as they're made. And, unlike working in a lower-level language, a mistake will trigger a human-friendly error message pointing to where the error occurred, instead of unexpectedly quitting.

As well as the set of built-in error reporting and debugging tools, there's flexibility over error handling for the end user of the application.

If encountering a problem with your code, there are a number of methods available to help track it down. This section details some of the main techniques to use.

11.1) The Error Dialogs

Often the first thing that will alert you to a problem is an error dialog. There are two possible types of error dialog.

Execution Error Dialog

The Execution Error dialog is displayed when a script is running and the Engine encounters an error, if Development > Script Debug Mode is turned off. Execution will halt. If you know what the error is from looking at it, use the Script button to go directly to the script and edit it. The line that generated the error will be highlighted in the script window. If Script Debug Mode is on, the script editor will automatically open up on the execution error and display the line that generated the error.

Compiling Error Dialog

The Compiling Error dialog is displayed when a script cannot be compiled because of a syntax error. This dialog is typically displayed when attempting to compile a change to a script by pressing the Apply button in the Script Editor. Correct the error and press the Apply button to compile the script again.

Caution: If a compile error is generated then the entire script to which it applies will not be compiled, not just the line or handler that contains the error. If other scripts are attempting to call commands or functions in this script they will not be able to run until the problem is corrected and the script compiled again. Because the Engine compiles all the scripts within a stack when it loads them, a Script Error dialog can be generated when opening a stack for the first time if the stack was saved with a script that could not compile.

Do not confuse the Execution Error and Compiling Error dialogs. The Execution Error dialog occurs when a script is running and cannot continue due to an error. The error dialog will start with the words "executing at [time]". The Execution Error dialog appears when attempting to compile a script that contains a syntax error. The error dialog will start with the words "compiling at [time]".

Tip: Turning on the Variable Checking option in the Script Editor, the Engine will require you to declare all variables and enclose all literal strings in quotes, or it will report a script compile error. This can be useful to catch mistakes before running a script. If this option is turned on and you attempt to compile an existing script that has not been written in this way, a large number of errors will need to be corrected before you can compile it.

11.2) Suppressing Errors And Messages

If a stack gets into an unstable state where it is generating a large number of errors, you can temporarily turn off sending messages to the stack or displaying error messages. This can enable you to edit the stack to make the appropriate changes.

To suppress messages, choose Development > Suppress Messages. Normal system event messages will stop being sent to the stack (for example, clicking a button will no longer trigger a mouseUp handler, changing cards will no longer trigger an openCard handler). Custom messages can still be sent directly to objects within the stack.

To suppress errors, choose Development > Suppress Errors. This will prevent any errors from triggering an error dialog.

Caution: Turn messages and errors back on when finished editing. Otherwise the stack will not work, or any error that comes up during stack operation will cause execution to halt but will not display an error message.

Tip: Execute a single command in the Message Box with a similar effect as Suppress Messages by including lock messages; before it. For example, in the Message Box:
lock messages; go next --go without messages
lock messages; quit --exit without messages

11.3) Output Information As A Script Executes
For info on the state of a particular variable or condition during execution, use the Variable Watcher. However sometimes it's easier to use the put command to output into to the Message Box, a field, or a text file, or use the write command to output to the Console, or use the answer command to output to a dialog.

Output To The Message Box
The Message Box is a convenient way to output info while a script is running. Display the info within the IDE without having to create a field or stack to display it. Use a put command without specifying a destination automatically outputs to the Message Box:
put tMyVar --display contents of tMyVar in Message Box
if tMyVar = "true" then put tData --conditional display contents of tData

Whenever displaying something in the Message Box, a special global variable called message (often abbreviated to msg) is updated. This allows you to check the contents of the Message Box easily, as well as add or append information rather than replacing it.
put cr& tInformation &cr after msg --display tInformation in new line after existing data in Message Box.

Output To A Field
Create a stack that contains fields used for debugging:
put tMyVar &cr after field "Debugging Info" of stack "My Debugger"

Tip: You may want to create a stack with tools that make it easier to debug. Create this stack then save it into your plugins folder so it is available from the Development menu in the IDE.

Output To A Text File
To log info more permanently, use a text file. You may want to store the file path in a global variable to change it easily.
put tMyVar &cr after url ("file:debug.txt")

Output To Standard Out (stdOut / Console On Mac OS X)
The stdOut is a useful place to log messages. Unlike the Message Box, it is easy to log a sequence of events which can be scrolled back to review later. It also has the advantage of being external to the Program so using it does not interfere with application window layering or focus in any way. The stdOut is only available on Linux or Mac OS X systems. On Mac OS X, it can be accessed by opening the Console application, located in the Utilities folder within Applications.

The syntax to write something to stdOut or the Console is:
write tMessage &cr to stdout

Tip: If writing a lot of data out to the console, append the time to each one to make it easier to debug. If you need more granularity than seconds, use the milliseconds instead of the long time.
write tMessage && the long time &cr to stdOut.

Tip: If inserting debugging output statements into your code, consider making them conditional on a global variable. This allows you to turn on debugging by setting the variable true without making changes to code. Even better it prevents forgotten debugging code in the application inadvertently filling the console with messages on an end user's system.
if gDebugging then write tMessage &cr to stdOut --global gDebugging = true/false
---
Output To A Dialog
To display a dialog with the contents of a statement, use the answer command. This method is suitable to output something quickly, but is unsuitable for larger debugging tasks as it pauses execution to display a dialog each time.
if tMyVar is "true" then answer tData --conditional display contents of tData in dialog
---
Interrupting Execution
To interrupt a script while it is running, press ctrl/cmd-period. Interrupting a script will only work if the global allowInterrupts property is set to true. Use for a run-away script, often in a repeat forever loop with no conditional escape. Code a fail-safe into untested complex repeat loops:
if the shiftKey = "down" then exit to top --fail-safe escape

11.4) The Debugger
Use the debugger to track down a problem with a script. The debugger provides a simple interface to step through a script line by line as it executes, watching the results of each statement as it happens. If the Variable Watcher is loaded from within the debugger, it will show the contents of all variables used in a particular handler. These values are updated with each step through. Edit these values while a script is running to fix a problem or test a section of code with a different set of data as it runs.

To activate the debugger, first choose Development > Script Debug Mode is turned on. Then open the script to debug and click in the gray bar to the left of the line to begin debugging. Alternatively, write the command breakPoint into the script. Using the breakPoint command allows a conditional break:
if tMyVar is "true" then breakPoint

Next run the script as normal. When the Engine reaches a breakpoint it will pause execution and load the debugger.

Important: To see the contents of variables while a script is running, wait for the debugger to open, then choose Debug > Variable Watcher.

Press the Step Into button to execute the current line and move to the next line. Step Over performs the same action, except that if the current line calls a function it will run that entire function without opening it in the debugger.

Once you have found the cause of the problem, press the Run button to exit the debugger and run the script at full speed. Alternatively press the Abort button to halt execution on the current line and edit the script.

Tip: To improve the performance when debugging a complex script, set a breakpoint further down in the script during the debugging process and press the Run button instead of pressing Step Over. The debugger is temporarily deactivated until the new breakpoint is reached, which means the script runs at full speed. When using Step Over, the debugger is still active even though it does not display a routine as it runs, which is slower.

11.5) The Variable Watcher
o Context Menu: Choose the execution context to display. This will show all the available contexts currently executing. For example, if a button has called a function and currently paused within that function, the variables displayed will be for that function. Go back and look at the variables in the handler that called the function by choosing the handler from this menu.

o Conditional Breakpoints: Attach a conditional breakpoint to a variable. For example, to create a breakpoint that can be

triggered when tMyVar = 2, click on the breakpoint area to the left of the line with tMyVar and type: tMyVar = 2 into the dialog. Press the Run button in the debugger. Execution will halt if tMyVar becomes equal to 2.

o List Of Variables: This lists all the variables in the current execution context. The contents of each variable is listed in the area on the right. If the contents is truncated, click on a variable to have the full contents displayed in the edit area at the bottom. The display area is updated with each step while debugging.

o Edit Area: Display and the contents of the currently selected variable. To change the content, enter a new value.
o Edit Script: Go to the Script Editor or debugger window for the current execution context.

11.6) The Message Watcher
The Message Watcher lets you see what messages are sent during a particular operation. Use the Message Watcher to create a log of the sequence of messages generated during a particular operation. It also logs how long each message takes to run. This makes it useful to locate bottlenecks in the code. Choose Development > Message Watcher.

Message Watcher Options
o Message List Area: List messages as they happen. The format is the name of the message, the time the message was sent and the number of milliseconds since the last message. Click to select a message, double click to open it in the Code Editor.
o Object Field: Show the object that the message was sent to. Click on a line in the Message List to update this field.
o Message Type: Show the type for the selected message - command, function, getProp, or setProp.
o Active: Activate the Message Watcher. Deactivate after capturing the sequence of events desired and stop logging additional messages.
o Clear: Clear the message list field.
o Suppress: Set which messages to suppress and which to display. Use this option to narrow down the messages logged.

Suppress Messages Options
o Action Handled: Don't log any message that causes a handler to run when it is sent.
o Action Not Handled: Don't log any message that does not cause a handler to run when it is sent. This is the default option and prevents the log from filling up with messages that do not cause any scripts to run.
o IDE Messages: Don't log IDE messages. The IDE generates lots of messages. This is the default option.
o Handler Type: Don't log the selected type of handler. For example, to prevent displaying all function calls, check the function box.
o Message List: A list of messages not to log. By default mouseMove is listed as mouseMove messages would flood the display.
o Add: Add a message name to prevent it from being logged.
o Delete: Delete a message name to cause it to be logged in the future.

11.7) Custom Error Handling
When creating an application to be distributed, consider including a method for catching and handling errors. There are two methods.

Try/Catch Control Structure
The try/catch control structure can be inserted around any code routine that you know may encounter problems when in use. For example, use the try/catch method around a routine that reads in a file from disk to handle the case where a corrupted file has been selected.

Custom ErrorDialog Handler
The second method is to write a custom errorDialog handler. Unlike a try/catch control structure, an errorDialog handler is global to the application (or card or stack within an application) and does not allow a script that encounters an error to continue. Use the errorDialog method to display a custom error dialog when an error occurs that could not be predicted and reported using try/catch.

Using Try/Catch
Enclose code that may be error prone within a try/catch control structure. The following example shows a routine that opens and

reads from a file enclosed in a try statement. If an execution error occurs in any of the statements after the try clause, the catch clause will be triggered with the details of the error:

```
try
  open file tFile
  read from file tFile until eof  --read to end of file
  close file tFile
catch tSomeError  --contains details of error
  answer "Sorry, an error occurred reading a file." &cr& tSomeError with "OK" titled "Error"
end try
if tSomeError <> empty then exit to top  --stop execution
```

Tip: The data from the error routine is returned in the internal format that's used to display execution errors. To look up the human friendly string associated with a particular error, look at the first item returned against the list of execution errors stored in the IDE. This will only work in the IDE:
put line (item 1 of tSomeError) of the cErrorsList of card 1 of stack "revErrorDisplay"

To include statements that will run regardless of whether there has been an error or not, include the statements as part of a finally clause.

To create readable error messages for cases where you anticipate there may be an error, use the throw keyword. For example, to display an error message when the result for opening a file returns something:
open file tFile
if the result is not empty then throw the result

In the example above, if the file cannot be opened and the result returns something, the value of the result will be passed to the catch statement in the tSomeError variable.

Writing A Custom ErrorDialog Handler
When an execution error occurs, an errorDialog message is sent. The IDE uses this message to display the execution error dialog. However, you can write and include a custom errorDialog handler. This is useful if planning to distribute an application. For example, create a stack that transmits the error information directly to your technical support department or displays a custom message on screen. A basic errorDialog handler:

```
on errorDialog pError
  answer "Sorry, there was an error." &cr& pError with "OK" titled "Error"
end errorDialog
```

This routine will also be activated if using the throw keyword to throw an error (outside of a try/catch control structure).

11.8) Standalone Errors
We recommend debugging stacks as fully as possible in the IDE using the Debugger and other tools. Occasionally you may need to track
down a problem that only occurs in a standalone application. If this happens, use the techniques above for writing information out to the stdOut or a text file. Include an error display dialog using the Bug Reports tab within the Standalone Settings screen, checking either the Allow User To Save Report To File or Allow User To Email Report buttons to view the errors generated.
==========

CHAPTER 12) WORKING WITH DATABASES

Intro, Basics, Database Cursor, SQLite Example, Choosing A Database, Drivers.

With the Database library, a stack can communicate with external SQL databases. Get data from single-user and multi-user databases, update data in them, get information about the database structure, and display data from the database in a stack.

This topic discusses how to install necessary software to communicate with databases, and how to use the Database library to communicate with a database. You should know how to write short scripts and should understand the basic concepts of SQL databases (rows and columns, database cursors, and SQL queries). This topic includes an example on how to set up and create a SQL database.

A few terms used, such as "field" and "cursor", are part of the standard terminology for working with databases, but have a different meaning in stack development. When referring to database-specific terms, the documentation usually uses phrases like "database field" or "database cursor" as a reminder.

12.1) Introduction To Databases
A database is an external resource that holds information, structured in a special form for quick access and retrieval. Databases can be:
o Any size from small to extremely large.
o Located on the same system as the stack or on a remote server.
o Accessible by one user at a time or by many users at once.

SQL Databases
A SQL database is a database accessed and controlled using SQL, a standard database-access language which is widely supported. Use SQL queries (statements in the SQL language) to specify the part of the database to work with, to get data, or to make changes to the database.

SQL access is built-in and fully-featured. Send any SQL statement to an SQL database. Open multiple databases (or multiple connections to the same
database), maintain multiple record sets (database cursors) per connection, and send and receive binary data as well as text. Do all this using the commands and functions in the Database library. See database in the Dictionary.

12.2) The Basics Of Database Structure
A database is built of records, which in turn are built out of database fields. A database field is the smallest part of a database that can be separately addressed. Each database field contains a particular kind of information. This might be a name, a file path, a picture, or any other kind of information. Each record contains one value for each of its fields. A set of records is called a database table, and one or more tables comprise a database.

Here's an example: there is a database of customers for a business. The database fields of this database might include the customer name, a unique customer ID number, and a shipping address. Each record consists of the information for a single customer, so each record has a different customer name, shipping address, and so on.

Note: The database structure being described resembles a multiple-card stack that has the same fields on each card. A database field is like a field in a stack, and a record is like a card. A stack set up this way can act as a database. A stack lacks some of the features of an external database, such as the ability to perform SQL queries and the ability to be accessed by more than one user.

Think of the set of customer records as a grid (like a spreadsheet). Each row is a record, and each column is a database field, so each cell in the grid contains a different piece of information about a particular customer. Here's an example of a Customers database table:

ID	Customer	Address	Country
123	Jane Jones	234 E. Street	U.K.
336	Acme Corp	PO Box 2378	USA
823	CanCo Inc.	CanCo Blvd.	Japan

There are three rows in this database (each is the record for a particular customer) and four columns (each is one of the database fields). A row of the database means one single customer record, which has one value for each field. A column of the database means the set of all values for one of the fields, one for each record (for example, the set of all customer addresses).

The set of all customer records makes a database table. A database might include only this table, or it might include other related tables, such as a list of all sales. Related data can be connected from different tables of the same database. For example, if each record in a Sales table includes the ID number of the customer who bought the product, you can link all the sales records for a customer to that customer's record in the Customers table.

12.3) Database Cursors - record sets.
SQL works primarily with sets of records rather than individual rows. When sending a SQL query to a database, the query typically selects certain records to perform further operations on. The set of records resulting from a SQL query is called a database cursor, or record set. SQL queries let you describe the characteristics of the records you require, instead of processing each record one by one to find out whether it matches a criteria.

For example, consider the above Customer table. To work with only US customers, write a SQL query that selects only records where the country field is "USA". This subset of records then becomes a database cursor or record set. You can find out more about the fields and records contained in a record set, and move from record to record within this subset of the database. You can create more than one record set to work with multi-sets of records at a time.

Note: Different database implementations have different limitations on movement within a record set. For example, some databases won't allow backward movement within a record set, movement must be forward starting at the first record.

12.4) SQLite Example
This is an example of an SQLite database of a book library, with author and title. The auto-generated ID is required. The database uses the revOpenDatabase and revDataFromQuery functions, and revExecuteSQL command. In stack "SQLite1" put the following code in the stack script. It creates the database or opens an existing one, and loads it into a field:

```
local sDatabaseID  --script local, available to handlers below, persists until quit
on preOpenStack  --in stack script
  local tName, tDatabaseFile, tTableSQL, tSQL
  put the short name of this stack into tName
  put specialFolderPath("documents") & "/" & tName & ".sqlite" into tDatabaseFile
  if there is not a file tDatabaseFile then  --create db file
    put revOpenDatabase("sqlite",tDatabaseFile) into sDatabaseID
    put "CREATE TABLE books(BookID INTEGER PRIMARY KEY AUTOINCREMENT, Author varchar(255), Title varchar(255))" into
tTableSQL
    revExecuteSQL sDatabaseID,tTableSQL
    --add some initial entries:
    put "INSERT INTO books(Author,Title) VALUES ('Tolkien J','Lord Of The Rings')" into tSQL
    revExecuteSQL sDatabaseID,tSQL
    put "INSERT INTO books(Author,Title) VALUES ('Austen J','Pride And Prejudice')" into tSQL
    revExecuteSQL sDatabaseID,tSQL
    put "INSERT INTO books(Author,Title) VALUES ('Baum F','Wizard Of Oz')" into tSQL
    revExecuteSQL sDatabaseID,tSQL
  else put revOpenDatabase("sqlite",tDatabaseFile) into sDatabaseID  --db file exists
  put dbListAll() into fld "Display"
end preOpenStack

function dbListAll  --retrieve all from database
  local tSQL
  put "SELECT * FROM books" into tSQL  --select all
  return revDataFromQuery(tab,cr,sDatabaseID,tSQL)
end dbListAll

function dbSearch pSearchTerm  --retrieve where title or author matches search term
  local tSQL
```

```
  put "SELECT * FROM books WHERE author LIKE '%" & pSearchTerm & "%' OR title LIKE '%" & pSearchTerm & "%'" into tSQL
  return revDataFromQuery(tab,cr,sDatabaseID,tSQL)
end dbSearch

command AddRecord pAuthor, pTitle  --add a record
  local tSQL
  put "INSERT INTO books(Author,Title) VALUES (" &quote& pAuthor &quote& comma &quote& pTitle &quote& ")" into tSQL
  revExecuteSQL sDatabaseID,tSQL
  put dbListAll() into fld "Display"
end AddRecord

command DeleteRecord pID  --delete a record
  local tSQL
  put "DELETE FROM books WHERE BookID = '" & pID & "'" into tSQL
  revExecuteSQL sDatabaseID,tSQL
  put dbListAll() into fld "Display"
end DeleteRecord
```

---

o To see the database, place scrolling field "Display" on the stack. In the Message Box:

```
set the tabStops of fld "Display" to "90"  --for tabbed data
```

o To search the database, place text entry field "Search Term" on the stack. Put the following code in the script of field "Search Term":

```
on returnInField
  put dbSearch(line 1 of me) into msg  --message box
end returnInField
```

o To add and delete records, place button "+/- Record" on the stack. Put the following code in the script of btn "+/- Record":

```
on mouseUp
  local tAuthor, tTitle
  ask "Add Record: enter Author,Title." &cr& "Delete Record: enter record's ID integer." with "1" titled "Add/Delete Record"
  if it = empty then exit to top
  if the number of items in it = "2" then
    put item 1 of it into tAuthor
    put word 1 to -1 of item 2 of it into tTitle  --remove char 1 space
    AddRecord tAuthor, tTitle
  else if it is an integer and it >= "1" then DeleteRecord it
end mouseUp
```

Save the stack with ctrl-S. Close the stack and re-open to trigger the preOpenStack handler. A .sqlite database is created in the Documents folder. See revOpenDatabase, revExecuteSQL, revDataFromQuery, and all the revDatabase functions in the Dictionary.

12.5) Choosing A Database

The following databases are fully supported: Oracle, MySQL, SQLite, PostgreSQL, Valentina. There is also support for connecting to a database via ODBC. Use ODBC to use Access, FileMaker, MS SQL Server, and many other databases.

Database commands and functions use the same syntax regardless of the type of database. You don't need to learn a separate database language for each type. When first opening a database with the revOpenDatabase function, specify the type as one of the parameters so the Engine knows what type of database it's dealing with. The Database library handles the details of each type behind the scenes.

Reasons To Choose A Database Type

Which type of database to choose depends on a number of factors. If working with an existing database, or already have a

database manager installed, the decision is made for you. If already an expert at a particular database type, you'll probably prefer to go on using that one. Other factors may include price, performance, licensing model (commercial or open source), and platform support. If your users are on a particular platform, you'll need to choose a database that is supported on that platform.

Overview of ODBC
Open Database Connectivity (ODBC) is a system that allows developers to access any type of compatible database in a standard way. To communicate with a database, you usually have to add code that uses the database's own proprietary protocols. Without ODBC, in order to create a program that can communicate with FileMaker, Access, and Oracle databases, this program would have to include code for three different database protocols. With ODBC, communicate with any of these database types using the same code.

ODBC Managers
To work with databases through ODBC, two pieces of software are needed: an ODBC manager, plus a database driver for the specific database type being used. Windows and Mac OS X include ODBC software with the operating system. For Linux systems, download an ODBC manager and a set of drivers. (See Software for Database Access below.)

Performance For Direct Access Versus ODBC Access
Typically, accessing a database via ODBC takes more configuration and is slower than accessing the database directly. For this reason, access is built-in to MySQL and PostgreSQL databases directly without going through the ODBC protocol. This ability will be valuable for anyone doing complex or extensive professional database work.

The syntax of the functions in the Database library is identical for all database types, so there's no need to rewrite scripts to take advantage of the increased efficiency of direct access.

12.6) Software For Database Access - drivers.
To provide connectivity to databases, the Engine works with database drivers - software that translates application requests into the protocol required by a specific database.

Finding Database Drivers
Database drivers for certain database types are built-in. (The list of included database types depends on the platform.) The sections below list which database drivers are included with which platforms.

You have all the software needed to use the included database types. For other database types, obtain the appropriate database drivers before working with those databases.
o MySQL: database drivers included on Linux, Mac OS X, and Windows systems.
o PostgreSQL: database driver included on Linux, Mac OS X and Windows systems.
o SQLite: database drivers included on Linux, Mac OS X, and Windows systems.

ODBC Managers And Database Drivers
To use a database via ODBC, you must install the necessary ODBC software for your platform. (Some operating systems include an ODBC installation.) ODBC software includes one or more database drivers, plus an ODBC manager utility.

o ODBC On Windows Systems: Windows systems include the MDAC (Microsoft Data Access Components) package as part of the standard system installation. To configure ODBC on Windows systems, use the ODBC Data Sources control panel.

o ODBC On Mac OS X Systems: Mac OS X includes iODBC software as part of the standard system installation. To configure ODBC on Mac OS X systems, use the ODBC Administrator application in the Utilities folder.

o ODBC On Linux Systems: The Engine supports iODBC and UnixODBC on Linux systems. Download the iODBC software from the iODBC web site at http://www.iodbc.org/. You can download the unixODBC software from the unixODBC web site at http://www.unixodbc.org.

Important: The links to third-party web sites and information about third-party software are provided for convenience, but have no responsibility for the software packages and sites referenced.

Creating A DSN For ODBC Access
Once you have installed the necessary software, use the ODBC manager to create a DSN, which is a specification that identifies a particular database. Use the DSN to connect to the database via ODBC. The following example opens a connection to a database whose DSN is named "myDB":
get revOpenDatabase("ODBC","myDB",,"jones","pass")

One of the advantages of setting up a DSN is that if you wish to change the location of the database, you only need to edit the DSN settings, not the application code. Think of a DSN as a kind of shortcut or alias to your database.
==========

CHAPTER 13) WORKING WITH XML
Tree Structure, XML Library, Retrieving Info, Editing XML Library.

Extensible Markup Language, or XML, is a general-purpose language for exchanging structured data between different applications and across the Internet. It consists of text documents organized into a tree structure. It can generally be understood by both human and machine.

There is comprehensive support for XML through its built-in XML library. Additionally, standards exist to support exchange of XML over a network connection (or "web services"), most notably through the XML-RPC and SOAP protocols. There is a built-in XML-RPC library and examples to build SOAP applications available.

13.1) The XML Tree Structure
XML is simply a data tree. It must start with a root node, be well formed and nested. Tags may not overlap. For more information on XML see http://en.wikipedia.org/wiki/XML

Elements Of An XML Tree
o Root node: The start of the XML document, includes a declaration the file is XML, the version of XML in use, and the text encoding.
o Comment: Comments can be placed anywhere in the tree and must start with \<! and end with ->. Double dashes (--) are not allowed.
o Node/Tag: The items that make up the content within an XML document.
o Attributes: Properties attributable to a given node. A node may have zero or more properties.
o Empty Node: A method of specifying a node exists but is empty.
o Entity Reference: A method of specifying special characters. XML includes support for &, \<, >, ' and ". Additional entities can be defined using a Document Type Definition or DTD.

When To Use XML
XML has a number of advantages and disadvantages. It is predominantly useful when exchanging data between different applications or systems, but is not suitable for all application types.

The advantages of XML are: it is text based making it more easily readable by humans as well as just machines; it is self describing; it is based on international standards and in widespread use with a large number of editors available for it; the hierarchical structure makes it suitable for representing many types of document; and it is platform independent.

The disadvantages of XML are: it is sometimes less efficient than binary or other forms of text representations of data; for simple applications it is more complicated than necessary; and the hierarchical model may not be suitable for all data types.

You may decide that using XML is the best solution for your particular data storage or transmission requirements. Or you may be working on a project with others where using XML or a web service based on it is a requirement. However in many cases a binary format or database will be more appropriate. Give consideration to the method you intend to use as early as possible in the design of your application.

Methods For Handling XML

There is a build-in comprehensive XML library for working with XML documents. Using the XML library has the advantage of syntax documentation in the Dictionary and built-in functions for performing common operations on XML. However the library is implemented as an external command and does not benefit from native Engine syntax. For simple XML processing use the built in chunk expression support to do the parsing, matching or construction of XML. If working with complex XML, then the library includes a comprehensive suite of features. In addition to the XML library, there is a built-in script-based library for working with XML-RPC.

Tip: To see a list of commands for working with XML-RPC, filter the Dictionary with the term "XMLRPC".

13.2) The XML Library: Loading, Displaying And Unloading XML

Getting Started - Creating An XML Tree In Memory
To work with an XML document, start by creating an XML tree of that document in memory. Use the revCreateXMLTree function to create a tree in from a data source such as a variable, field, or download. Use the revCreateXMLTreeFromFile function to load an XML document from a file and create a tree.

revCreateXMLTree(XMLText, dontParseBadData, createTree, sendMessages)
revCreateXMLTreeFromFile(filePath, dontParseBadData, createTree, sendMessages)

In revCreateXMLTree, the XMLText is the string containing the XML. In revCreateXMLTreeFromFile, this parameter is replaced with the filePath to the XML document. Both functions return a single value - the ID of the tree that has been created.

Important: Both functions require you to specify all the parameters. You must store the ID returned by these functions in order to access the XML tree later in the script.

The dontParseBadData parameter specifies whether or not to attempt to parse poorly formed XML. If set to true, then bad data will be rejected and generate an error instead of constructing the tree in memory.

The createTree parameter specifies whether or not to create a tree in memory. Generally this is set to true, unless intending only to read in an XML file to determine whether or not it is properly structured.

The sendMessages parameter specifies whether or not messages should be sent when parsing the XML document. Messages can be useful to implement functionality such as a progress bar, progressively render or progressively process data from a large XML file as it is being parsed. If set to true, revXMLStartTree will be sent when the parsing starts, revStartXMLNode will be sent when a new node is encountered, and revEndXMLNode will be sent when a node has been completed, revStartXMLData will be sent at the start of a new block of data, and finally revXMLEndTree will be sent when processing is finished.

13.3) Retrieving Information From An XML Tree
After creating an XML tree in memory and stored the tree ID, use the functions in this section to retrieve information from within the tree. Any text fetched using the XML library will be in the encoding specified in the root node of the XML tree.

The examples in this section assume a tree has been loaded into memory using the revCreateXMLTree function which has returned an ID = 1.

Retrieving The Root Node
To retrieve the root node from the XML tree, use the revXMLRootNode(treeID) function.
```
put revXMLRootNode(1) into tRootNode  --tRootNode = stackFile
```

Retrieving The First Child Element In A Node
To retrieve the first child element use the revXMLFirstChild(treeID,parentNode) function.
o ParentNode contains the path to the node to retrieve the first child from. Nodes are referenced using a file-path like format with / used to denote the root and delimit nodes.
```
put revXMLFirstChild(1,tRootNode) into tFirstChild  --tRootNode = stackFile, tFirstChild = /stackFile/stack
```

Retrieving A List Of Children In A Node
To retrieve a list of children in a node use the revXMLChildNames(treeID,startNode,nameDelim,childName,includeChildCount) function.
o NameDelim is the delimiter that separates each name that is returned. To get a line list of names, specify return.
o ChildName is the name of the type of children to list.
o IncludeChildCount includes the number of each child in square brackets next to the name if set to true.

put revXMLChildNames(1,"/stackFile/stack",return,"card",true) into tNamesList

This results in tNamesList containing:
card[1]
card[2]

Retrieving The Contents Of The Children In A Node
To retrieve a list of children of a node including their contents, use the revXMLChildContents(treeID,startNode,tagDelim,nodeDelim,includeChildCount,depth) function.
o TagDelim is the delimiter that separates the contents of the node from its name.
o NodeDelim is the delimiter that separates each child node.
o Depth specifies the number of generations of children to include. If you use -1 as the depth then all children are return.

put revXMLChildContents(1,"/stackFile/stack",space,return,true,-1) into tContents

This results in tContents containing:
card[1]
field[1]
text[1] Hello World!
htmlText[1] Hello World
card[2]

Retrieving The Number Of Children In A Node
To retrieve the number of children of a node use the revXMLNumberOfChildren(treeID,startNode,childName,depth) function.
put revXMLNumberOfChildren(1,"/stackFile/stack","card",-1) into tContents

This results in tContents containing: 2

Retrieving The Parent Of A Node
To retrieve a node's parent use the revXMLParent(treeID,childNode) function.
put revXMLParent(1,"stackFile/stack") into tParent

Results in tParent containing: /stackFile

Retrieving An Attribute From A Node
To retrieve an attribute from a node use the revXMLAttribute(treeID,node,attributeName) function.
o AttributeName is the name of the attribute to retrieve the value for.
put revXMLAttribute(1,"/stackFile/stack","rect") into tRect

This results in tRect containing: 117,109,517,509

Retrieving All Attributes From A Node
To retrieve all attributes from a node use the revXMLAttributes(treeID,node,valueDelim,attributeDelim) function.
o ValueDelim is the delimiter that separates the attribute's name from its value.
o AttributeDelim is delimiter that separates the attribute's name & value pair from each other.
put revXMLAttributes(1,"/stackFile/stack/card/field",tab,return) into tFieldAttributes

This results in tFieldAttributes containing:
name Hello
rect 100,100,200,125

Retrieving the Contents Of Attributes
To retrieve the contents of a specified attribute from a node and its children, use the revXMLAttributeValues(treeID,startNode,childName,attributeName,delimiter,depth) function.
o ChildName is the name of the type of child to be searched, leave this blank to include all types of children.
o AttributeName is the name of the attribute to return the values for.
o Delimiter is the delimiter to be used to separate the values returned.
put revXMLAttributeValues(1,"/stackFile/", ,"rect",return,-1) into tRectsList

This results in tRectsList containing:
117,109,517,509
100,100,200,125

Retrieving The Contents Of A Node
To retrieve the contents of a specified node, use the revXMLNodeContents(treeID,node) function.
put revXMLNodeContents(1,"/stackFile/stack/card/field/htmlText") into tFieldContents

This results in tFieldContents containing:
Hello World

Note: The entity references for the < and > symbols have been translated into text in this result.

Retrieving Siblings
To retrieve the contents of the siblings of a node, use the revXMLNextSibling(treeID,siblingNode) function and the revXMLPreviousSibling(treeID,siblingNode) function.
o SiblingNode is the path to the node to retrieve the siblings from.

put revXMLPreviousSibling(1,"/stackFile/stack/card[2]") into tPrev
put revXMLNextSibling(1,"/stackFile/stack/card[1]") into tNext

This results in tPrev containing: /stackFile/stack/card[1]
And tNext containing: /stackFile/stack/card[2]

Searching For A Node
To search for a node based on an attribute, use revXMLMatchingNode.
revXMLMatchingNode(treeID, startNode, childName, attributeName, attributeValue, depth, [caseSensitive])

o ChildName is the name of the children to include in the search, if left blank all children are searched.
o AttributeName is the name of the attribute to search.
o AttributeValue is the search term to match.
o CaseSensitive optionally specifies whether the search should be case sensitive, default is false.

Using this function on the example XML file:
put revXMLMatchingNode(1,"/", ,"name","Hello",-1) into tMatch

This results in tMatch containing: /stackFile/stack/card[1]/field
---
Retrieving An Outline Of The Tree (or Portion Thereof)
To retrieve the contents of a specified node, use revXMLTree.
revXMLTree(treeID, startNode, nodeDelim, padding, includeChildCount, depth)

o NodeDelim is the delimiter that separates each node in the tree, use return to retrieve a list of nodes.

o Padding is the character to use to indent each level in the tree.

Using this function on the example XML file:
put revXMLTree(1,"/",return,space,true,-1) into tTree

This results in tTree containing:
stackFile[1]
stack[1]
card[1]
field[1]
text[1]
htmlText[1]
card[2]

Retrieving The Tree As XML (or Portion Thereof)
To retrieve the tree as XML use revXMLText. Use this function to save the XML to a file after modifying it.
revXMLText(treeID, startNode, [formatTree])

o FormatTree specifies whether or not to format the returned tree with return and space characters to make it easier to read by a human, default is false.

Using this function on the example XML file:
ask file "Save XML as:"
if it <> empty then put revXMLText(1,"/",true) into url ("file:" & it)

This results in the file the user specifies containing:
<stackFile>
<stack name="Example" rect="117,109,517,509">
<card>
<field name="Hello" rect="100,100,200,125">
<text>Hello World!</text>
<htmlText><p>Hello World</p></htmlText>
</field>
</card>
<card/>
</stack>
</stackFile>

Validating Against A DTD
To check whether the syntax of an XML file conforms to a DTD, use revXMLValidateDTD.
revXMLValidateDTD(treeID, DTDText)

o TreeID is the number returned by the revXMLCreateTree or revXMLCreateTreeFromFile function when creating the XML tree.
o DTDText is a Document Type Definition.

put revXMLValidateDTD(1,myDTD) into field "Errors"

Listing All XML Trees In Memory
To generate a list of all XML trees in memory, use revXMLTrees.
if myTree is not among the lines of revXMLTrees() then LoadTree myTree
---
Removing An XML Tree From Memory
To remove an XML tree from memory, use revDeleteXMLTree.
To remove all XML trees from memory, use revDeleteAllXMLTrees.
revDeleteXMLTree(1)

Both functions take a single parameter - the ID of the tree to be deleted. You should delete a tree when you have stopped using it.

Caution: Once an XML tree has been removed from memory, there is no way to get it back. Use the revXMLText function to retrieve the contents of the entire tree and save it first.
revXMLText(treeID [, startNode] [, formatted])

o StartNode is the path to the node to start. If a startNode is not specified, the revXMLText function starts at the root node and returns the entire XML tree.
o Formatted is whether or not to produce XML with indenting and line breaks, the default false is not to format the text resulting in XML being output as a single block of text with no line breaks.

put revXMLText(1, ,true) into url ("file:" & "New Customers.xml") --formatted text
---

13.4) Editing The XML Library
This section discusses how to edit XML trees. Before reading this section please read the section above on loading, displaying, and unloading XML.

Adding A New Child Node
To add a new node use the revAddXMLNode command.
revAddXMLNode treeID, parentNode, nodeName, nodeContents, [location]

o ParentNode is the name of the node to add the child to.
o NodeName is the name of the new node to create.
o NodeContents is the contents of the new node.
o Location - optionally specify "before" to place the new child at the start of the child nodes.

Use this command to add a button to the example XML file:
revAddXMLNode 1, "/stackFile/stack/card/", "button", ""

This results in the tree containing a new button:
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is a comment. -->
<stackFile>
<stack name="Example" rect="117,109,517,509">
<card>
<field name="Hello" rect="100,100,200,125">
<text>Hello World!</text>
<htmlText><p>Hello World</p></htmlText>
</field>
<button></button>
</card>
<card/>
</stack>
</stackFile>

To create another node at the same level as another node, use the revInsertXMLNode command instead.

Appending XML To A Tree
To add a new node use the revAppendXML command.
revAppendXML treeID, parentNode, newXML

o NewXML is XML to append to the tree.

Moving, Copying or Deleting A Node
To move a node use the revXMLMoveNode command.
revXMLMoveNode treeID, sourceNode, destinationNode [, location] [, relationship]

o SourceNode is the path to the node to move.
o DestinationNode is the path to the node to move to.
o Location specifies where the node should be moved to in the list of siblings, either "before" or "after".
o Relationship specifies whether to place the node alongside the destination as a sibling or below the destination as a child.

To copy a node use the revXMLCopyNode command.
To delete a node use revXMLDeleteNode command.

Putting Data Into A Node
To put data into a node use the revXMLPutIntoNode command.
revXMLPutIntoNode treeID, node, newContents

o NewContents is the text that the new node will contain.

Setting An Attribute
To set an attribute use the revXMLSetAttribute command.
revXMLSetAttribute treeID, node, attributeName, newValue

o AttributeName is the name of the attribute set the attribute on.
o NewValue is the value to set for the attribute.

Using this command to add a "showBorder" property to a field:
revXMLSetAttribute 1, "/stackFile/stack/card/button", "showBorder","true"

The field tag in the example tree now looks like this:
<field name="Hello" rect="100,100,200,125" showBorder="true">

Adding A DTD
To add a DTD to the tree, use the revXMLAddDTD command.
revXMLAddDTD treeID, DTDText

o DTDText is the text of the DTD to add.
==========

CHAPTER 14) TRANSFER INFO WITH FILES, THE INTERNET, AND SOCKETS
File Name & File Path, Special Folders, File Types, App Signature, File Ownership, File Extensions, Working With URLs, Upload & Download File, Browser, Web Form, FTP, HTTP, RevBrowser, Encryption & SSL, Write A Protocol With Sockets.

Reading and writing data to files or transferring data over the Internet are built-in features. Accessing data from a file typically takes just a single line of code. File path syntax uses the same format on each platform. A set of functions provides for copying, deleting, or renaming files, as well as accessing appropriate system and user folders.

Downloading and uploading data to the Internet typically takes just a single line of code. Support for the https, ftp, and post protocols is included. Syntax allows downloading in both the foreground and background. Additional libraries allow constructing multi-part form data, sending ftp commands, SSL, and encryption.

If built-in protocol support doesn't do what's needed, implement your own Internet protocols using its straightforward socket support. A very basic client server application can be written in a few lines of code.

14.1) File Name Specifications And File Paths
A file path is a way of describing the location of a file or folder so that it can be found by a handler. File paths are used throughout, to read to and write from text files, to reference an external file, and in many other situations. If an application refers to external files in any way, an understanding of file path is essential.

This topic discusses the syntax for creating and reading a file reference, and how to relate file paths to the location of an application so that they'll be accessible when the application is installed on another system with a different folder structure.

What Is A File Path?
A file path is a description of the exact location of a file or folder. The file path is created by starting at the top of the computer's file system, naming the disk or volume that the file is on, then naming every folder that encloses the file, in descending order, until the file is reached.

Locating A File
For example, the location of a file called "My File" is located inside a folder called "My Folder", which in turn is located inside a folder called "Top Folder", which is on a drive called "Hard Disk". All this information is needed to completely describe where the file is.

The Structure Of A File Path
/Hard Disk/Top Folder/My Folder/My File

To write a file path, start by naming the disk the file is on, then add each enclosing folder in order until arriving at the file.

Tip: To see the path to a file, enter the following in the message box:
answer file "Choose a file:"; put it --display file path in message box

Important: Each platform has its own way for programmers to specify file paths. The file path shown above is in the usual style for file paths on Linux systems. For cross-platform compatibility, the Engine uses this same forward slash / character in its file path regardless of the current platform. This way, you can generally specify files and work with paths in scripts without converting them when switching platforms.

File Paths On Windows Systems
On Windows systems, disks are named with a drive letter followed by a colon character (:). A typical file path on a Windows system looks like this: C:/folder/file.txt --windows
---
File Paths On Mac OS X Systems
On Mac OS X systems, the startup disk, rather than the desktop, is used as the top level of the folder structure. This means that the startup disk's name does not appear in file paths. Instead, the first part of the file path is the top-level folder that the file is in. If the disk "Hard Disk" is the startup disk, a typical path on Mac OS X systems might look like this:
/Top Folder/My Folder/My File --mac

Tip: To find the startup disk's name, check the first disk name returned by the volumes function.

For files on a disk that isn't the startup disk, the file path starts with "/Volumes" instead of "/". A typical file path to a file that's on a non-startup disk on a Mac OS X system looks like this:
/Volumes/Swap Disk/Folder/file.txt

Folder Paths
Construct the path of a folder the same way as the path to a file. A folder path always ends with a slash character (/). This final slash indicates that the path is to a folder rather than a file.

For example, this pathname describes a folder called "Project" inside a folder called "Forbin" on a disk named "Doomsday":
/Doomsday/Forbin/Project/ --folder path

If "Project" is a file, its pathname looks like this, without the final slash:

/Doomsday/Forbin/Project --file path (no last /)

File Paths For Mac OS X Bundles
A bundle is a special type of folder, used on Mac OS X, that is presented to the user as a single file but is maintained internally by the operating system as a folder. Many Mac OS X applications - including this program and the applications it creates - are stored and distributed as bundles that contain several files.

When the user double-clicks the bundle, the application starts up instead of a folder window opening to show the bundle's contents.

Take advantage of the bundle concept to include any needed support files with your application. If you place the files in the application's bundle, users ordinarily never see them, and the entire application - support files and all - behaves as a single icon.

Tip: To see the contents of a bundle, right-click (or control click) the bundle and choose "Show Package Contents" from the contextual menu.

Most of the time, the distinction between bundles and files doesn't matter. However it is recommended to refer to them as folders when coding. This will help to keep your code readable. To select a bundle in a file dialog use the answer file form. When moving, renaming, or deleting a bundle, refer to them as a folder.

Moving, Renaming, Or Deleting A Bundle
When using the rename command to rename a bundle, use the rename folder form of the command:
rename folder "/Volumes/Disk/Applications/MyApp/" to "/Volumes/Disk/Applications/OtherApp/" --bundle as folder

Similarly, when deleting with a bundle, use the delete folder command instead of delete file, and the revCopyFolder command instead of revCopyFile.

Referring To Files Inside A Bundle
When referring to a file inside a bundle, treat the bundle as if it were a folder. For example, if a file called "My Support.txt" is inside an application's bundle, the absolute path to the file might look like this:
/Volumes/Disk/Applications/MyApp/My Support.txt

The / Character In A File Or Folder Name
The slash (/) is not a legal character in Linux or Windows file or folder names, but it is legal for Mac OS X file or folder names to contain a slash. Since a slash in a file or folder name would cause ambiguity - is the slash part of a name, or does it separate one level of the hierarchy from the next? - the Engine substitutes a colon (:) for any slashes in folder or file names on Mac OS X systems.

For example, if a file on a Mac OS X system is named "Notes from 12/21/93", refer to it in a script as "Notes from 12:21:93". Since the colon is not a legal character in Mac OS X folder or file names, this removes the ambiguity.

Absolute And Relative File Paths
When describing how to get to a file, there are two options, absolute and relative.

o Absolute file path: Start from the top level, the name of the disk, and name each of the enclosing folders until you get to the file. This is called an absolute path, because it's independent of where you start from.
o Relative file path: Start from the current folder and describe how to get to the file from there. This is called a relative path, because it depends on where you start.

All the file paths shown so far in this topic are absolute paths.

Absolute File Paths
Absolute file paths do not depend on which folder a stack file is in or where the current folder is. An absolute path to a particular folder or file is always written the same way.

For example, suppose an application is in a folder called "Application Folder", and inside that is a folder called "Stories", and inside that is a file called "Westwind.txt".

The absolute file path of the application would be: /Hard Disk/Application Folder/My Application
The absolute path of the file would be: /Hard Disk/Application Folder/Stories/Westwind.txt

Note: On Mac OS X and Linux systems, absolute file paths always start with a slash character. On Windows systems, absolute file paths always start with a drive letter followed by a colon (:).

Relative File Paths
An alternative is to refer to the "Westwind" file by starting from the folder containing the application. Since the application is in "Application Folder", describe the location of the "Westwind" file relative to the application using it.

The relative file path of the application would be: My Application
The relative file path of the file would be: Stories/Westwind.txt

This relative pathname starts at "Application Folder" - the folder that holds the application - and describes how to get to the "Westwind" file from there: open the folder "Stories", then find "Westwind" inside.

A relative file path starts at a particular folder, rather than at the top of the file system like an absolute file path. The relative file path builds a file path from the starting folder to the file or folder whose location is being specified.

Finding The Current Folder
By default, the current folder is set to the folder containing the application, either the IDE or a standalone application. In the example above, the current folder is "Application Folder", because that's where the running application is located.

Tip: To change the current folder, set the defaultFolder property using an absolute file path. To get the current folder, get the defaultFolder property.
```
set the itemDel to "/"; put item -1 of the defaultFolder  --current folder of IDE in message box
```

Going Up To The Parent Folder
The relative path ".." indicates the current folder's parent folder. If the current folder is "Stories", the relative path .. is the same thing as the absolute path /Hard Disk/Application Folder/

Going Up Multiple Levels
To go up more than one level, use more than one "../". To go up two levels, use "../../"; to go up three levels, use "../../../", and so forth.

For example, if the current folder is "Stories", and its absolute path is: /Hard Disk/Application Folder/Stories/

To get to "My Application" in "Application Folder", go up one level to "Application Folder", then down one level to "My Application".
The relative path would be: ../My Application

To get to "Top Folder" on "Hard Disk", go up two levels - to "Application Folder", then to "Hard Disk" - and then down one level to "Top Folder".
The relative path would be: ../../Top Folder/

Starting At The Home Directory
On Mac OS X and Linux systems, the "~" character designates a user's home directory. A path that starts with "~/" is a relative path starting with the current user's home directory. A path that starts with "~", followed by the user ID of a user on that system, is a relative path starting with that user's home directory.

When To Use Relative And Absolute File Paths
Absolute file paths and relative file paths are interchangeable. Which one to use depends on a couple of factors.

Absolute file paths are easy to understand and they don't change depending on the current folder. This can be an advantage if changing the defaultFolder regularly. However absolute file paths always include the full name of the hard disk and folders leading up to the current working folder. Therefore, if you plan to distribute an application, work with relative paths, so that media shipped in subfolders with the application is still easy to locate.

Tip: By default, when linking to an image or resource using the Inspector, an absolute file path is inserted. If you plan to distribute an application, place any included media in a subfolder next to the stack and convert these file paths to relative file paths (by deleting the path elements up to the stack). No further changes will be needed when it comes time to distribute the application as a standalone.

It's OK to use absolute paths to specify files or folders the user selects after installation. For example, when asking the user to select a file (using the answer file command) and read data from the file, the disk name and folder structure of the user's computer will be used.

Special Folders

Modern operating systems have a set of special-purpose folders designated for a variety of purposes. For example, the contents of the desktop reside in a special folder. There's a special folder set aside for fonts and application preferences too.

These special folders don't always have the same name and location, so you can't rely on a stored file path to locate them. If an application is installed onto an OS localized into a different language, the names of the file path will be different, and some Windows special folders are named or placed differently depending on what version of Windows is running.

To find out the name and location of a special folder, regardless of any of these factors, use the specialFolderPath function. The function supports a number of forms for each operating system, describing the special folders for each one. Some of the forms are the same cross-platform. The following example will get the location of the Desktop folder on Windows, Mac OS X or Linux:

```
put specialFolderPath("desktop") &"/MyDBase.sqlite" into tPath  --file path to desktop of any user
put specialFolderPath("documents") & "/MyText.txt" into tPath  --file path to documents folder
if there is a file (specialFolderPath("desktop") &"/Snapshot1.png") then...
```

To get the path to the Start menu's folder on a Windows system:

```
put specialFolderPath("Start") into tFilePath
```

For a complete list, see specialFolderPath in the Dictionary.

14.2) File Types, Application Signatures & File Ownership

Double-clicking a document file automatically opens it with the application it's associated with. Each operating system has a different method for associating files with an application. In order to create files that belong to your standalone application, you must set up the association appropriately for each platform distributed on.

This topic describes how to correctly associate an application with the files it creates.

Windows File Extensions And Ownership

When a file is saved on a Windows system, a three-character extension is usually added to the file's name. The extension specifies the format of the file. To determine which application to launch when the user double-clicks a file, Windows checks the Windows registry to find out what application has registered itself as owning the file's extension. Each application can add keys to the registry to identify certain file extensions as belonging to it.

Applications That Don't Own Files

If an application does not create files to own, no modifications to the registry or file extensions are needed.

Applications That Own Their Own Files

If an application creates files with its own custom extension, during Windows installation, changes to the Windows registry must be made to identify the extension as belonging to the application.

Popular Windows installer programs will make these registry changes automatically, or use the setRegistry function.

Installing Custom Icons
Each Windows file can display its own icon. You can have separate icons for an application and for files it owns. Icon files must be stored in .ico format.

Custom Application Icons
To include a custom icon for a standalone application, use the "Application Icon" option on the Windows screen of the File > Standalone Application Settings window to specify the icon file. When you build the application, the icon will be included in the application.

Custom File Icons
To include a custom icon for documents, use the "Document Icon" option on the Windows screen of the File > Standalone Application Settings window to specify the icon file. When you build the application, the icon will be included in the application.

Important: For the correct icon to appear on files your application creates, the file's extension must be registered in the Windows registry.

File Extensions
Add an extension to the name of any Windows file. The extension may contain letters A-Z, digits 0-9, ' (single quote), !, @, #, $, %, ^, &, (,), -, _, {, }, `, or ~. The Windows registry associates applications with the extension for the files they own.

Mac OS X File Types And Creators
On Mac OS X each file has a file extension which determines which application owns it. However Mac OS X systems can also use the unique four-character creator signature and a four-character file type.

Mac OS X applications store file association information in a property list file, or plist. Each application's plist is stored as part of its application bundle.

Applications That Don't Own Files
To assign a unique creator signature when building a standalone application, enter the signature on the Mac OS X screen of the File > Standalone Application Settings window. The Engine automatically includes the creator signature in the application's plist.

Applications That Own Their Own Files
If an application creates files with your application's creator signature, include in the application's plist an entry for each file type used. Once the standalone application has been built, follow these steps to open the plist file:

1. Right click the application bundle, navigate to the contents folder and open the "Info.plist" file with a text editor or an app called "Property List Editor" from Apple's developer tools.

2. Locate the information for the document type. In Property List Editor, expand the "Root" node, then expand the "CFBundleDocumentTypes" node, then expand the "0" node. In a text editor, locate "CFBundleDocumentTypes". Below it, note the tags "<array>" and "<dict>". The information for the first document type is between "<dict>" and "</dict>".

3. Enter the file description, which is a short phrase describing what kind of file this is. In Property List Editor, change the value of "CFBundleTypeName" to the description you want to use. In a text editor, locate "CFBundleTypeName" in the document information. Below it is the file description, enclosed between "<string>" and "</string>": <string>OpenXTalk Stack</string> Change the description to the one you want to use.
Important: Do not change the tags (enclosed in "<" and ">"). Only change what's between them.

4. Enter the file extension. In Property List Editor, expand "CFBundleTypeExtensions" and enter the file extension in the "0" node. In a text editor, locate "CFBundleTypeExtensions" in the document information. Below it is the extension, enclosed in "<array>" and "<string>" tags. Change the extension to the one you want to use.

5. Enter the four-character file type. In Property List Editor, expand "CFBundleTypeOSTypes" and enter the file type in the "0" node. In a text editor, locate "CFBundleTypeOSTypes" in the document information. Below it is the file type, enclosed in "<array>" and "<string>" tags. Change the file type to the one you want to use. If the format for this type of file is standard (such as plain text), use a standard type (such as "TEXT"). If the format belongs to your application, use a custom file type of your choice.

Important: Apple reserves all file types with no uppercase letters. If you use a custom file type for an application, make sure it contains at least one uppercase letter.

To assign more file types to an application, copy the portion of the plist file inside the "CFBundleTypes" array between "<dict>" and "</dict>", including these tags. The "CFBundleTypes" node should now contain two "<dict>" nodes and all their contents. Repeat the steps above for each different file type your application can create.

Creating Files
When your application creates files, set the fileType property to the desired creator signature and file type for the new file. (For stack files created with the save command, use the stackFileType property instead.) When creating files, the application uses the current value of the fileType or stackFileType property to determine what creator and file type the new file should have.

It's important to understand that a file's creator signature determines which application is launched automatically when you double-click the file, but doesn't prevent other applications from being able to open that file. For example, if an application creates files of type "TEXT", any text editor can open the files. If your application creates stack files, and uses the file type "RSTK", then the Engine will be able to open the stack files, as well as your application.

File Extensions
Add an extension to the name of any Mac OS X file. When the user double-clicks a file with no creator signature, the operating system uses the extension to determine which application to use to open the file. An application bundle's name should end with the extension ".app".

Note: Apple's recommendations for determining file type and creator on Mac OS X systems are currently in flux. The recommended method for the present is to set a file type and creator signature, and also attach an extension to the end of each file's name when it is created. Valid extensions on Mac OS X systems are up to twelve characters in length, and may include the letters a-z, the digits 0-9, $, %, _, or ~. For up-to-date information on Apple's recommendations for Mac OS X, see Apple's developer documentation at <http://www.apple.com/developer/>.

Linux File Extensions
Linux systems do not have an overall required method for specifying a file's type, but most files on a Linux system are created with extensions in the file name, similar to the extensions used on Windows systems. These extensions may be of any length and may include any characters (other than /).

14.3) Working With URLs
A URL is a container for a file (or other resource), which may either be on the same system the application is running on, or on another system that's accessible via the Internet.

This topic discusses the various URL schemes, how to create and manipulate files using URLs, and how to transfer data between your system and a FTP or HTTP server.

To fully understand this topic, you should know how to create objects and write short scripts, understand how to use variables to hold data, and a basic understanding of the Internet.

An Overview Of URLs
A URL is a container for a file or other document, such as the output of a CGI on a web server. The data in a URL may be on the same system the application is running on, or may be on another system.

URLs are written like the URLs in a browser. Use the URL keyword to designate a URL, enclosing the URL's name in double quotes:

put field "Info" into url ("file:myfile.txt")
get url "http://www.example.org/stuff/nonsense.html"
put url "ftp://ftp.example.net/myfile" into field "Data"

URL Schemes
A URL scheme is a type of URL. Five URL schemes are supported with the URL keyword: http, https, ftp, file, binfile, and resfile.

The http, https, and ftp schemes designate documents or directories that are located on another system that's accessible via the Internet. The file, binfile, and resfile schemes designate local files.

The https Scheme
An https URL designates a document from a web server. When using an https URL in an expression, the Engine downloads the URL from the server and substitutes the downloaded data for the URL:
put url "https://www.example.org/home.html" into field "Page"

The ftp Scheme
A ftp URL designates a file or directory on a FTP server. When using a ftp URL in an expression, the Engine downloads the URL from the server and substitutes the downloaded data for the URL:
get url "ftp://user:passwd@ftp.example.net/picture.jpg"

When putting something into a ftp URL, the Engine uploads the data to the FTP server:
put image 10 into url "ftp://user:passwd@ftp.example.net/picture.jpg"

FTP servers require a user name and password, which are specified in the URL. Without a user name and password, the Engine adds the "anonymous" user name and a dummy password automatically, in accordance with the convention for public FTP servers. Uploading to a FTP server usually requires a registered user name and password though.

Directories On A FTP Server
A URL that ends with a slash (/) designates a directory (rather than a file). A ftp URL to a directory evaluates to a listing of the directory's contents.

The file Scheme
A file URL designates a file on your system:
put field "Stuff" into url ("file:/Disk/Folder/Testfile")

When using a file URL in an expression, the Engine gets the contents of the file designated and substitutes it for the URL. The following example puts the contents of a file into a variable:
put url ("file:Myfile.txt") into tMyVariable

When putting data into a file URL, the Engine puts the data into the file:
put tMyVariable into url ("file:/Volumes/Backup/Data.txt")

Note: As with local variables, if the file doesn't exist, putting data into it creates the file.

To create a URL from a file path the user provides, use the & operator:
answer file "Please choose a file to get:"
if it <> empty then get url ("file:" & it) --absolute file path to file
---
File Path Syntax And The File Scheme:
The file URL scheme uses the same file path syntax used elsewhere in statements. Use both absolute paths and relative paths in a file URL.

Conversion Of End-Of-Line Markers
Different operating systems use different characters to mark the end of a line. Mac OS X uses a return character (ASCII 13), Linux systems use a linefeed character (ASCII 10), and Windows systems use a return followed by a linefeed. To avoid problems, the

Engine always uses linefeeds internally when using a file URL as a container, and translates as needed. To avoid this translation, use the binfile scheme.

The binfile Scheme
A binfile URL designates a file on your system that contains binary data:
put url "binfile:beachball.gif" into image "Beachball"

When using a binfile URL in an expression, the Engine gets the contents of the file designated and substitutes it for the URL. The following example puts the contents of a binary file into a variable:
put url "binfile:picture.png" into tPictVar

When putting data into a binfile URL, the Engine puts the binary data into the file:
put tPictVar into url "binfile:/Volumes/Backup/Pict.png"
put image 1 into url "binfile:/Image.png"

As with local variables, if the file doesn't exist, putting data into it creates the file.

Note: The binfile scheme works like the file scheme, except the Engine does not attempt to convert end-of-line markers. This is because return and linefeed characters can be present in a binary file but not be intended to mark the end of the line. Changing these characters can corrupt a binary file, so the binfile scheme leaves them alone.

The resfile Scheme
On Mac OS Classic (and sometimes on Mac OS X systems), files can consist of either a data fork or a resource fork or both.

Important: While the Engine supports reading and writing resource fork files on Mac OS X and Mac OS Classic, this feature is only intended to help access and work with legacy files. It is not recommended to use resource forks when designing any new application.

The resource fork contains defined resources such as icons, menu definitions, dialog boxes, fonts, and so forth. A resfile URL designates the resource fork of a Mac OS Classic or Mac OS X file:
put myBinaryData into url "resfile:/Disk/Resources"

When using a resfile URL in an expression, the Engine gets the resource fork of the file designated and substitutes it for the URL.

When putting data into a resfile URL, the Engine puts the data into the file's resource fork.

Tip: A resfile URL specifies the entire resource fork, not just one resource. To work with individual resources, use the getResource, setResource, deleteResource, and copyResource functions.

The most common use for this URL scheme is to copy an entire resource fork from one file to another. To modify the data from a resfile URL, you need to understand the details of Apple's resource fork format.

Creating A Resource Fork
Unlike the file and binfile URL schemes, the resfile keyword cannot be used to create a file. If the file doesn't yet exist, you cannot use the resfile keyword to create it.

To create a new resource file, first use a file URL to create the file with an empty data fork, then write the needed data to its resource fork:
put empty into url "file:MyFile" --create an empty file
put myStoredResources into url "resfile:MyFile" --now with a resource fork
---
Manipulating URL Contents
Use a URL like any other container. Get the content of a URL, or use its content in any expression, or put any data into a URL.
o http, https, ftp, binfile, and resfile URLs can hold binary data.
o http, https, ftp, and file URLs can hold text.

The URL Keyword
To specify a URL container, use the URL keyword before the URL, which can use any of the five schemes described above:
if url "https://www.example.net/index.html" is not empty then...
get url "binfile:/Applications/Hover.app/data"
put 1+1 into url "file:Output.txt"

The URL keyword tells the Engine you are using the URL as a container.

Note: Some properties (such as the filename of a player or image) specify a URL as the property's value. Be careful not to include the URL keyword when specifying such properties, otherwise the property will be set to the content of the URL instead of the URL path.

Using The Content Of A URL
As with other containers, use the content of a URL by using a reference to the URL in an expression. The Engine substitutes the URL's content for the reference.

If the URL scheme refers to a local file (file, binfile, or resfile URLs), the Engine reads the content of the file and substitutes it for the URL reference in the expression:
answer url "file:../My File" --display the file's content
put url "binfile:Flowers.jpg" into tMyVariable
put url "resfile:Icons" into url "resfile:New Icons"

If the URL scheme refers to a document on another system (http, https, or ftp URLs), the Engine downloads the URL automatically, substituting the downloaded data for the URL reference:
answer url "https://www.example.net/files/Greeting.txt"

If the server sends back an error message - for example, if the file specified in an https URL doesn't exist - then the error message replaces the URL reference in the expression.

Important: When using an http, https, or ftp URL in an expression, the handler pauses until the Engine is finished downloading the URL. To allow a handler to continue while accessing these resources, use the load URL form of the command.

Putting Data Into A URL
As with other containers, you put data into a URL. The result of doing so depends on whether the URL scheme specifies a file on your system (file, binfile, or resfile) or on another system (http, https, or ftp).

If the URL scheme refers to a local file (file, binfile, or resfile URLs), the Engine puts the data into the specified file:
put field "My Text" into url "file:Storedtext.txt"
put image 1 into url "binfile:Picture.png"

If the URL scheme refers to a document on the Internet (http, https, or ftp URLs), the Engine uploads the data to the URL:
put tMyVar into url "ftp://me:pass@ftp.example.net/file.dat"

Important: Most web servers do not allow HTTP or HTTPS uploads. Use the ftp scheme with username and password.

Chunk Expressions And URLs
Like other containers, URLs can be used with chunk expressions to specify a portion of what's in a URL - a line, an item, a word, or a character. In this way, any chunk of a URL is like a container itself.

Use any chunk of a URL in an expression the same way used with a whole URL:
get line 2 of url "https://www.example.net/index.html"
put word 8 of url "file:/Disk/Folder/Myfile" into field 4
if char 1 of url "ftp://ftp.example.org/test.jpg" is "0" then...

Specify ranges, and even one chunk inside another:
put char 1 to 30 of url "binfile:/marks.dat" into tMyVar
answer line 1 to 3 of url "https://www.example.com/file"

Putting Data Into A Chunk
If the URL is local (a file, binfile, or resfile URL), put a value into a chunk of the URL:
put it into char 7 of url "binfile:/Picture.gif"
put return after word 25 of url "file:../datafile"
put field 3 into line 20 of url ("file:" & "Myfile.txt")

To make several changes to a file on a server, it's faster to first put the URL in a variable, change the chunk(s), then put the variable into the URL:
put url "ftp://me:secret@ftp.example.net/file.txt" into tMyVar
put field "New Info" after line 7 of tMyVar
put field "More" into word 22 of line 3 of tMyVar
put tMyVar into url "ftp://me:secret@ftp.example.net/file.txt"

This ensures the file only needs to be downloaded once and re-uploaded once, no matter how many changes are made.

URLs And Memory
URLs are only read into memory when using the URL in a statement. Other containers - like variables, fields, buttons, and images - are normally kept in memory, so accessing them doesn't increase memory usage.

This means that in order to read a URL or place a value in a chunk of a URL, the Engine reads the entire file into memory. Because of this, be cautious when using a URL to refer to any very large file.

Even referring to a single chunk in a URL, the Engine must place the entire URL in memory. An expression such as line 347882 of url "file:bigfile.txt" may be evaluated very slowly or even not work at all, if insufficient memory is available. If referring to a chunk of an http, https, or ftp URL, the Engine must download the entire file to find the chunk specified.

Tip: To read and write large quantities of data to a file, or search (seek) through the content of a large file without loading the entire file into memory, use the open file, read from file, seek, and close file commands instead of the URL commands.

Deleting URLs
Remove a URL with the delete URL command.

To delete a local file, use a file or binfile URL:
delete url "file:C:/My Programs/Test.exe"
delete url "binfile:../Mytext.txt"

It doesn't matter whether the file contains binary data or text; for deletion, these URL schemes are equivalent.

Tip: You can also use the delete file command to remove a file.

To delete the resource fork of a file, use a resfile URL:
delete url "resfile:/Volumes/Backup/Project.rev" --delete resource fork, leave file

Tip: To delete a single resource instead of the entire resource fork, use the deleteResource function.

To delete a file or directory from a FTP server, use a ftp URL:
delete url "ftp://root:secret@ftp.example.org/Deleteme.txt" --delete file
delete url "ftp://me:mine@ftp.example.net/trash/" --delete folder
---

The simplest way to transfer data to a FTP or HTTPS server is to use the put command to upload, or use the URL in an expression to download.

The Internet library includes additional commands to upload and download files to and from a FTP server. These additional commands offer more versatile options for monitoring and controlling the progress of the file transfer.

Uploading Using The Put Command
Use the put command with a ftp or https URL to upload data to a server:
put tMyVariable into url "ftp://user:pass@ftp.example.org/Newfile.txt"

Use the put command with a file or binfile URL as the source, to upload the file to a server:
put url "file:Newfile.txt" into url "ftp://user:pass@ftp.example.org/Newfile.txt"

When uploading data in this way, the operation is blocking: the handler pauses until the upload is finished. If there's an error, the error is placed in the result function:
put field "Data" into url tMyFTPDestination
if the result is not empty then beep 2

Important: Uploading or downloading a URL does not prevent other messages from being sent during the file transfer: the current handler is blocked, but other handlers are not. For example, the user might click a button that uploads or downloads another URL while the first URL is still being uploaded. In this case, the second file transfer is not performed and the result is set to "Error Previous request has not completed." To avoid this problem, set a flag while a URL is being uploaded, and check that flag when trying to upload or download URLs to make sure that there is not already a file transfer in progress.

Downloading Using A URL
Referring to a ftp or http URL in an expression downloads the document.
put url "ftp://ftp.example.net/Myfile.jpg" into image 1
get url "https://www.example.com/newstuff/newfile.html"

Use the put command with a file or binfile URL as the destination to download the document.
put url "ftp://ftp.example.net/Myfile.jpg" into url "binfile:/Disk/Folder/Myfile.jpg"

Non-Blocking File Transfers - the load command.
When transferring a file using URL containers, the file transfer stops the current handler until the transfer is done. This kind of operation is called a blocking operation, since it blocks the current handler as long as it's going on.

To transfer data using https without blocking, use the load command. To transfer large files using ftp, use the libURLftpUpload, libURLftpUploadFile, or libURLDownloadToFile commands.

Non-blocking file transfers have several advantages:
Since contacting a server may take some time due to network lag, the pause involved in a blocking operation may be long enough to be noticeable to the user. If a blocking operation involving a URL is going on, no other blocking operation can start until the previous one is finished.

If a non-blocking file transfer is going on, other non-blocking file transfers can start. This means that using library commands, the user can begin multiple file transfers without errors.

During a non-blocking file transfer, you can check and display the status of the transfer. The transfer's progress can be displayed and allow the user to cancel the file transfer.

Using The Load Command
The load command downloads the specified document in the background and places it in a cache. Once a document has been cached, it can be accessed nearly instantaneously when using its URL, because the Engine uses the cached copy in memory instead of downloading the URL again.

To use a file that has been downloaded by the load command, refer to it using the URL keyword as usual. When requesting the original URL, the Engine uses the cached file automatically.

For best performance, use the load command at a time when response speed isn't critical (such as when an application is starting up), and only use it for documents that must be displayed quickly, (such as images from the web shown when navigating to another card).

Checking Status When Using The Load Command - URLStatus function

While a file is being transferred using the load command, check the status of the transfer using the URLStatus function. This function returns the current status of a URL that's being downloaded or uploaded:

```
put the URLStatus of "ftp://ftp.example.com/Myfile.txt" into field "Current Status"
```

The URLStatus function returns one of the following values:

- queued: on hold until a previous request to the same site is completed.
- contacted: the site has been contacted but no data has been sent or received yet.
- requested: the URL has been requested.
- loading bytesTotal, bytesReceived: the URL data is being received.
- uploading bytesTotal, bytesReceived; the file is being uploaded to the URL.
- cached: the URL is in the cache and the download is complete.
- uploaded: the application has finished uploading the file to the URL.
- error: an error occurred and the URL was not transferred.
- timeout: the application timed out when attempting to transfer the URL.

To monitor the progress of a file transfer or display a progress bar, check the URLStatus function repeatedly during the transfer. The easiest way to do this is with timer based messaging.

Canceling A File Transfer & Emptying The Cache - unload command.

To cancel a transfer initiated with the load command and empty the cache, use the unload command.

```
unload url "https://example.org/newbeta"
```

Uploading And Downloading Large Files Using FTP

The Internet library provides a number of commands for transferring larger files via FTP without blocking.

- libURLftpUpload: upload data to a FTP server.
- libURLftpUploadFile: upload a file to a FTP server.
- libURLDownloadToFile: download a file from a FTP server to a local file.

The basic effect of these commands is the same as the effect of using URLs: data is transferred to or from the server. However, there are several differences in how the actual file transfer is handled. Because of these differences, the special library commands are more suitable for uploads and downloads, particularly if the file being transferred is large.

The following sets of statements each show one of the Internet library commands, with the equivalent use of a URL:

```
libURLftpUpload tMyVar,"ftp://me:pass@example.net/File.txt"
put tMyVar into url "ftp://me:pass@example.net/File.txt"  --equivalent

libURLftpUploadFile "test.data","ftp://ftp.example.org/test"
put url "binfile:test.data" into url "ftp://ftp.example.org/test"  --equivalent

libURLDownloadToFile "ftp://example.org/newbeta","/HD/File"
put url "ftp://example.org/newbeta" into url "binfile:/HD/File"  --equivalent
```

Using Callback Messages - the URLStatus function.

When starting a file transfer using the libURLftpUpload, libURLftpUploadFile, or libURLDownloadToFile commands, you can optionally specify a callback message, which is usually a custom message you write a handler for. This message is sent whenever the file transfer's URLStatus changes. Use the callback message to handle errors or to display the file transfer's status to the user.

The following example demonstrates how to display a status message to the user with handlers in a button's script.

```
on mouseUp
  libURLDownloadToFile "ftp://example.org/newbeta","/HD/LatestBeta","showStatus"
end mouseUp

on showStatus pURL
  put the URLStatus of pURL into field "Status"
end showStatus
```

When the user clicks the button, the mouseUp handler is executed. The libURLDownloadToFile command begins the file transfer, and its last parameter specifies that a showStatus message will be sent to the button whenever the command's URLStatus changes.

As the URLStatus changes periodically throughout the download process, the button's showStatus handler is executed repeatedly. Each time a showStatus message is sent, the handler places the new status in a field. The user can check this field at any time during the file transfer to see whether the download has started, how much of the file has been transferred, and whether there has been an error. This proceeds with the one click of the button.

If a file transfer was started using the libURLftpUpload, libURLftpUploadFile, or libURLDownloadToFile commands, cancel the transfer using the unload
command in a different button.

```
on mouseUp
  unload url "ftp://example.org/newbeta"
end mouseUp
```

Uploading, Downloading, And Memory

When using a URL as a container, the Engine places the entire URL in memory. For example, if downloading a file from a FTP server using the put command, the Engine downloads the whole contents of the file into memory before putting it into the destination container. If the file is too large to fit into available memory, a file transfer using this method will fail (and may cause other unexpected results).

The library commands libURLftpUpload, libURLftpUploadFile, and libURLDownloadToFile, however, do not require the entire file to be loaded into memory. Instead, they transfer the file one piece at a time. If a file is (or might be) too large to comfortably fit into available memory, always use the special library
commands to transfer it.

Using A Stack On A Server

Ordinarily, stack files are located on a local disk. You can also open and use a stack located on a FTP or HTTPs server. This capability allows you to update an application by downloading new stacks, make new functionality available via the Internet, and even keep most of your application on a server instead of storing it locally.

Going to a stack on a server:

As with local stack files, use the go command to open a stack that's stored on a server:

```
go stack url "https://www.example.org/myapp/Main.rev"
go stack url "ftp://user:pass@example.net/Secret.rev"
```

Note: For such a ftp statement to work, the stack file must have been uploaded as binary data, uncompressed, and not use encodings such as BinHex.

Tip: If you need to download a large stack, use the load command first to complete the download before using the go command to display the stack. This allows you to display a progress bar during the download. The Engine automatically downloads the stack file. The mainstack of the stack file then opens in a window, just as though you had used the go command to open a local stack file.

You can go directly to a specific card in the server's stack:
go card "My Card" of stack url "https://www.example.org/myapp/Main.rev"

To open a substack instead, use the substack's name:
go stack "My Substack" of url "https://www.example.org/myapp/Main.rev"

Using A Compressed Stack
You cannot directly open a stack that's compressed. However, since the stack URL is a container, you can use the URL as the parameter for the decompress function. The function takes the stack file data and decompresses it, producing the data of the original stack file. You can open the output of the function directly as a stack.

The following statement opens a compressed stack file on a server:
go decompress(stack url "https://www.example.net/comp.gz")

The statement automatically downloads the file "comp.gz", uncompresses it, and opens the mainstack of the file. The extension for a compressed file is ".gz".

Saving Stacks From A Server
When a stack is downloaded using the go command, it's loaded into memory, but not saved on a local disk. Such a stack behaves like a new (unsaved) stack until you use the save command to save it as a stack file.

Note: Saving a stack that has been downloaded with the go command does not re-upload it to its server. To upload a changed stack, you must save it to a local file, then use one of the methods described in this topic to upload the saved file to the server.

Other Internet Commands
The Internet library has a number of additional commands for working with web forms, ftp commands, custom settings, and troubleshooting. These commands are documented in the Dictionary.

14.5) Launching The User's Browser With A URL
To launch the default browser with a URL, use the launch URL command.
launch url "https://www.google.com"

Note: To render web pages within a stack, instead of launching an external browser, use the revBrowser.

14.6) Working With Web Forms
To post data to a web form, use the post command. To encode data to make it suitable for posting, use the libUrlFormData function. To create multi-part form data (as described in RFC 1867) use the libUrlMultipartFormData function. To add data to a multipart form one part at a time, use the libUrlMultipartFormAddPart function. This can be useful to specify the mime type or transfer encoding for each part.

14.7) Working With FTP
For details on basic uploading and downloading using FTP, see the section above.

The following commands provide additional capabilities when working with the ftp protocol:
o libURLSetFTPStopTime: set the timeout value for FTP transfers.
o libURLSetFTPMode: switch between active and passive mode for FTP transfers.
o libURLSetFTPListCommand: switch between sending LIST or NLST formats when listing the contents of a FTP directory.
o libURLftpCommand: send a ftp command to anftp server.
o libURLftpUpload: upload data.
o libURLftpUploadFile: upload a file, without loading the entire file into memory.
o libURLDownloadToFile: download data to a file, without loading the entire data into memory.

14.8) HTTP Methods And HTTPS URLs
The basic operations used by the HTTP protocol are called methods. For https URLs, the following HTTP methods are used under the following circumstances:

o GET: when an https URL in an expression is evaluated.
o PUT: when a value is put into an https URL.
o POST: when using the post command.
o DELETE: when using the delete URL command with an https URL.

Note: Many HTTPS servers do not implement the PUT and DELETE methods, which means you can't put values into an https URL or delete an https URL on such servers. It's common to use the FTP protocol instead to upload and delete files. Check with your server's administrator to find out what methods are supported.

The HTTPHeaders Property
When the Engine issues a GET or POST request, it constructs a minimal set of HTTP headers. For example, when issued on a Mac OS X system, the statement:
put url "https://www.example.org/Myfile" into tMyVariable

results in sending a GET request to the server:
GET /myfile HTTP/1.1
Host: 127.0.0.0
User-Agent: OpenXTalk (MacOS)

Add headers, or replace the Host or User-Agent header, by setting the HTTPHeaders property before using the URL:
set the HTTPHeaders to "User-Agent: MyApp" & return & "Connection: close"
put url "https://www.example.org/Myfile" into tMyVariable

Now the results of the GET request sent to the server looks like this:
GET /myfile HTTP/1.1
Host: 127.0.0.0
User-Agent: MyApp
Connection: close

The ftp URL scheme can be used to create a new file to a FTP server. As with the file and binfile schemes, putting something into the URL creates the file:
put dataToUpload into url "ftp://jane:pass@ftp.example.com/Newfile.dat"

Tip: Create a FTP directory by uploading a file to a new (nonexistent) directory. The directory is automatically created. Then delete the file, leaving a new empty directory on the server.
--create an empty file in the nonexistent directory:
put empty into url "ftp://jane:pass@example.com/newdir/dummy"
--delete unwanted empty file to leave new directory:
delete url "ftp://jane:pass@example.com/newdir/dummy"

Additional Transmission Settings
The following commands provide additional customization options for the Internet library:
o libUrlSetExpect100: set a limit to the size of data being posted before requesting a continue response from the server.
o libURLSetCustomHTTPHeaders: set the headers to be sent with each request to an HTTPS server.
o libURLFollowHttpRedirects: specify that GET requests should follow HTTP redirects and GET the page redirected to.
o libUrlSetAuthCallback: set a callback for handling authentication with https servers and proxies.

Troubleshooting
The following commands and functions can be useful when debugging an application that uses the Internet library.
o resetAll: close all open sockets and halts all pending Internet operations.

Caution: The resetAll command closes all open sockets, which includes any other sockets opened by the application and any sockets in use for other uploads and downloads. Because of this, avoid routine use of the resetAll command. Consider using it only during development, to clear up connection problems during debugging.

o libURLErrorData: returns any error that was caused during a download that was started with the load command.
o libURLVersion: returns the version of the Internet library.
o libURLSetLogField: specifies a field for logging information about uploads and downloads on screen.
o libURLLastRHHeaders: returns the headers sent by the remote host in the most recent HTTP transaction.
o libURLLastHTTPHeaders: returns the value of the httpHeadersproperty used for the previous HTTP request.

14.9) Rendering A Web Page Within A Stack - revBrowser
Use the revBrowser commands to render a web page within a stack. RevBrowser uses WebKit (Safari) on Mac OS X and Internet Explorer on Windows. Currently RevBrowser is not supported under Linux.

Create A Web Browser
To create a browser object in a stack, use the revBrowserOpen function. This function takes the windowID for the stack you want to open the browser in and a URL. Please note that the windowID is not the same as the stack's ID property.

```
put the windowId of this stack into tWinID
put revBrowserOpen(tWinID,"https://www.google.com") into sBrowserId
```

To set properties on the browser, use the revBrowserSet command. The following commands makes the border visible then sets the rectangle to be the same as an image named "BrowserImage":

```
revBrowserSet sBrowserId, "showborder","true"
revBrowserSet sBrowserId, "rect",rect of img "BrowserImage"
```

To close a browser when finished with it, use the revBrowserClose command. This command takes the windowID for the stack containing the browser:

```
revBrowserClose sBrowserId
```

RevBrowser supports a number of settings and messages. You can intercept a message whenever the user navigates to a link, prevent navigation, intercept clicks in the browser, requests to download files, or to open a new window.

For a complete list of revBrowser commands, see browser in the Dictionary.

14.10) Encryption And SSL
The Program includes an industrial strength encryption library to encrypt files or data transmissions. It also includes support for using Secure Sockets Layer (SSL) and the https protocol.

Encrypting And Decrypting Data
To encrypt data, use the encrypt command. The encrypt command supports a wide variety of industry standard methods of encryption. The list of installed methods can be retrieved by using the cipherNames function. To decrypt data, use the decrypt command. See encrypt, decrypt, and cipherNames in the Dictionary.

Tip: If using the encryption library on a Windows system, it is possible that another application will have installed DLLs that use the same name as the ones included to support encryption. To force your application to load these SSL DLLs, set the $PATH environment variable before loading the library.

```
put $PATH into tOldPath
put <path to SSL DLLs> into $PATH
get the cipherNames  --force loading of the SSL DLLs
put tOldPath into $PATH
```

Connecting Using HTTPS
Connect and download data from a URL using https in the same way that you access an http URL.

put url "https://www.example.com/store.php"

If there is an error, it will be placed into the result. To include a user name and password, use the following form:
https://user:password@www.example.com/

14.11) Writing A Protocol With Sockets
To implement a protocol, use the open socket command. To implement a secure protocol, use the open secure socket variant of the open socket command. Specify whether or not to include certification, a certificate, and a key. See open socket in the Dictionary.

To understand this chapter it is assumed you understand the basics of how the Internet works, including the concepts of sockets, IP addresses, and ports. More information on these concepts can be found in Wikipedia.

Tip: The standard protocols supported such as http and ftp have all been implemented as a scripted library for socket support. To examine this library, in the message box: edit script of button "revlibURL" of stack "revLibrary". Caution: if you accidentaly change anything, the Internet commands may cease to operate.

Opening A Connection
To open a connection use the open socket command. The following command opens a connection to the IP address specified in the tIPAddress variable and the port specified in the tPort variable. It sends the message "chatConnected" when a connection has been established:
open socket (tIPAddress & ":" & tPort) with message "chatConnected"

To open a secure socket, use the open secure socket variant of the command. To open a UDP datagram socket, use the open datagram socket variant of the command. For more information on these variants, see open socket in the Dictionary.

Looking Up A Host Name Or IP Address
Look up an IP address from a host name with the hostNameToAddress function. For example, to get the IP address for the openxtalk.org server:
put hostNameToAddress("www.openxtalk.org") into tIPAddress

To get the host name of the local machine, use the hostName function. To look up the name from an IP address, use the hostAddressToName function.

Reading And Writing Data
Once a connection opens, it will send a chatConnected message. To receive data, use the read from socket command. The following message reads data from the socket and sends a chatReceived message when reading is completed:
```
on chatConnected pSocket
  read from socket pSocket with message "chatReceived"
end chatConnected
```

Once reading from the socket is completed the chatReceived message can be used to process or display the data. It can then specify that it should continue to read from the socket until more data is received, sending another chatReceived message when done.
```
on chatReceived pSocket, pData
  put pData after field "ChatOutput"
  read from socket pSocket with message "chatReceived"
end chatReceived
```

To write data to the socket, use the write command:
write field "ChatText" to socket tSocket

Disconnecting
To disconnect, use the close socket command. Store a variable with details of any open sockets and close them when finished

using them or when the stack closes.
close socket (tIDAddress & ":" & tPort)

Listening For And Accepting Incoming Connections
To accept incoming connections on a given port, use the accept connections command. The following example listens for connections on port 1987 and sends a chatConnected message if a connection is established. Then start to read data from the socket in a chatConnected handler.
accept connections on port 1987 with message "chatConnected"

Handling Errors
If there is an error, the Engine will send a socketError message with the address of the socket and the error message. If a socket is closed a socketClosed message will be sent. If a socket times out waiting for data a socketTimeout message will be sent.

To get a list of sockets that are open, use the openSockets function. Set the default timeout interval by setting the socketTimeOutInterval property. For more details on all of these features, see the Dictionary.
==========

CHAPTER 15) EXTEND THE BUILT-IN CAPABILITIES
Read & Write To Command Shell, Launch An App, Close An App, AppleScript & VBScript, AppleEvents, Local Socket, PlugIn, Edit The IDE, Externals.

This chapter covers how to extend the built-in capabilities. It's useful if planning a front-end to any existing application or set of processes.

There are many ways to extend the Program. This topic explains how to run shell commands, start other applications, read and write to processes, execute AppleScript, VBScript, send and respond to AppleEvents, and communicate between multiple processes. It also explains where to get information to create external commands and functions (code written in lower level languages), and detail how to extend the IDE with a plugin or edit the IDE itself.

15.1) Read And Write To The Command Shell
Use the shell function to run a command shell and return the result. The following example displays a directory listing on Mac OS X:
answer shell("ls")

And this example stores a directory listing in a variable on Windows:
put shell("dir") into tDirectory

On Windows systems, to prevent a terminal window from being displayed, set the hideConsoleWindows global property to true.

Choose a different shell program by setting the shellPath global property. By default this is set to "/bin/sh" on Mac OS X and Linux, and "command.com" on Windows.

Tip: The shell function blocks the Engine until it is completed. To run a command shell in the background, write the shell script to a text file, then execute it with the launch command.

15.2) Launching Other Applications
Use the launch command to launch other applications, documents or URLs. To launch an application, supply the full path to the application. The following example opens a text document with TextEdit on Mac OS X:
launch "/Users/someuser/Desktop/MyTextDocument.rtf" with "/Applications/TextEdit.app"

Tip: To get the full path to an application, use the answer file command to select the application then copy it into your script. In the message box:
answer file "Select an application"; put it

To open a document with the application it is associated with use the launch document command.
launch document "C:/MyDocument.pdf"

To open a URL in the default web browser, use the launch URL command.
launch url "https://www.google.com/"

15.3) Opening And Closing Another Application
Use the open process command to open an application or process to read and write data from. Once opened, read from the process with the read from process command and write to it with the write to process command. To close a process you have opened, use the close process command. The openProcesses function returns a list of processes you have opened and the openProcessIDs function returns the process IDs of each one. For more details see the Dictionary.

Use the kill process command to send a signal to another application, to close it, or to force it to exit. See kill process in the Dictionary.

15.4) Using AppleScript And VBScript (Open Scripting Architecture or Windows Scripting Host)
To execute commands using AppleScript on Mac OS or VBScript on Windows, use the do as command. The do as command also allows you to use any other Open Scripting Architecture languages on Mac OS X or languages installed into the Windows Scripting Host on Windows.

To retrieve a list of the available installed languages, use the alternateLanguages function.
if "AppleScript" is among the lines of the alternateLanguages then do tScript as "AppleScript"

For example, to execute an AppleScript that brings the Finder on Mac OS X to the front, enter the following into a field:
tell application "Finder" activate end tell

Then run:
do field 1 as "AppleScript"

To retrieve a result from commands executed using do as, use the result function. Any error message will also be returned in the result. The following example displays the result of an addition performed using VBScript:
do "result = 1 + 1" as "VBScript"
answer the result

See do as in the Dictionary.

15.5) AppleEvents
To send an AppleEvent, use the send to program command. If the Engine receives an AppleEvent it will send an appleEvent message to the current card. Intercept this message to perform actions such as handling a request to quit the application or opening a document. The following example shows how to handle a request to quit a stack:

```
on appleEvent pClass, pID, pSender  --in current card script
  if pClass & pID is "aevtquit" then
    put checkSaveChanges() into tOkToQuit  --call a custom function prompting user to save changes
    if tOkToQuit is true then quit  --user pressed "save"
    else exit appleEvent  --user pressed "cancel"
  end if
end appleEvent
```

To retrieve additional information passed with the appleEvent use the request appleEvent data command. The following example shows how to handle a request to open a stack:

```
on appleEvent pClass, pID, pSender  --in current card script
  --appleEvent sent when stack is opened from the finder
  if pClass & pID is " aevtodoc " then
    --get the file path(s)
    request AppleEvent data
    put it into tFilesList
    repeat for each line i in tFilesList
      go stack i
    end repeat
  end if
end appleEvent
```

For more details see appleEvent in the Dictionary.

15.6) Using Local Sockets
To communicate between local applications a common technique is to open a local socket and communicate using that. A local socket can be used without code changes on all the platforms supported,. Choose a port number that is not used by a standard protocol - typically a high number.

This technique is used to create multiple programs that run independently but communicate with each other. It is a viable technique for running background tasks and provides a straightforward way to create an application that behaves as if threaded. Design your application such that additional instances can be launched to perform processing, data transfer, or other intensive activities. Modern OSes will allocate each application to an appropriate processor core.

By using local socket messaging to communicate with each application, you can keep your main application's user interface responsive and display status information. The following example shows you how to open a local socket to the local machine:
open socket to "127.0.0.1:10000" with message "gotConnection"

Tip: To simplify communication between multiple stacks, consider writing a simple library that sends and receives a handler name together with parameter data. To call a handler in the other stack, send the handler name and data to the library. The library will send the data over a socket. In the receiving program intercept the incoming data from the socket and use it to call the appropriate message with the parameter data received.

15.7) Creating PlugIns
Create a plugin to help perform tasks needed to do regularly in the IDE. Plugins are stacks. To create a plugin, drag your stack to Documents folder > My OpenXTalk folder > Plugins folder. Within the Program choose Development > Plugins > YourStackName. By default the new plugin will be loaded as a palette. Create a custom "Property Inspector" style utility or other object editing tool.

The Plugin Settings Screen
Choose Development > Plugins > Plugin Setting to apply settings to a plugin.

o Open plugin when:
By default a plugin will load when choosing it from development > plugins. To have a plugin load whenever the Program starts, select the "starts up" option. Use this if the plugin sets up the environment, for example by loading stacks in construction or adjusting window layout. To have a plugin load when the Program quits choose the "quits" option. Use this if the plugin performs clean up tasks before exit.

o Open as:
Choose the mode to open the plugin as. Choose the invisible option to load the plugin stack invisible. Use this option to create a plugin that installs a faceless library (for example by inserting a button within it into the front or backscripts), or to perform some other automated task that does not require a visible user interface.

Note: Loading from a plugin will not allow you to edit the plugin itself. To edit the plugin, first load it from the menu, then use the Project Browser to make it toplevel by right clicking on it in the list of stacks and choosing Toplevel from the popup menu.

o Send messages to plugin:
To have a plugin respond while working in the IDE, register it to receive messages. The IDE can send a variety of messages to the current card in the plugin as you change selections, switch tools, open and close stacks, etc. The messages that can be sent are listed below.
revCloseStack, revEditScript, revIDChanged, revMouseMove, revMoveControl, revNameChanged, revNewTool, revPreOpenCard, revPreOpenStack, revResizeControl, revResizeStack, revResumeStack, revSaveStackRequest, revSelectedObjectChanged, revSelectionChanged, revShutdown.

Tip: Internally the IDE implements these plugin messages by intercepting system messages sent by the Engine in the IDE frontScripts and backScripts, then sending out a corresponding message to any loaded plugin. Look up these messages in the Dictionary by removing the "rev" in front of the messages above.

For example, to have a plugin update whenever an object is selected with the pointer tool, select the revSelectedObjectChanged message. Then insert the following handler into the plugin's card script:

```
on revSelectedObjectChanged
  put the selObj into tObject
  --insert code to operate on tObject here
end revSelectedObjectChanged
```

15.8) Editing The IDE
The IDE (integrated development environment) components - the Menus, Tools Palette, Property Inspector, Script Editor, Debugger, etc. - are all implemented as stacks. The IDE has a series of library frontScripts and backScripts it uses to provide functionality both for the IDE and for your application. Some of these libraries are used only by the IDE (e.g. the debugger library), others (e.g. the Internet library, libURL) are copied into a standalone by the Standalone Builder.

Caution: Editing the IDE can easily cause the Program to become unusable. Only advanced users should attempt to edit the IDE. Back up an IDE stack prior to making any changes. Don't attempt to edit the IDE while working on any mission critical project.

To edit the IDE requires administrator privileges. On Windows right-click the Program shortcut on the desktop > run as administrator. For all platforms, choose View > UI Elements In Lists. This displays the IDE's own stacks within the Project Browser and other editing screens. Now load these stacks to edit them.

The IDE uses the stack "revLibrary" to provide much of its functionality. The scripts are stored as a series of buttons and loaded into the frontScripts and backScripts when the IDE is started. To edit these scripts, go to the Front Scripts or Back Scripts tab within the Message Box and check the "Show UI Scripts" checkbox. A mistake editing revFrontScript or revBackScript will make the Program non-responsive requiring a force-quit.

To make changes permanent, you must explicitly save the IDE stack by right-clicking it in the Project Browser and choose save. Unsaved changes revert back to the original after the Program is quit.

15.9) Externals - code written in lower level languages.
This program provides an external interface to extend it using a lower level language (usually C++). For example, with preexisting code that performs processing in a lower level language, write a library with this program and then call the library by writing a simple wrapper around it using this program's externals interface.

There is support for transmitting data to and from externals, as well as drawing into image objects within stack windows, manipulating the player object, and more.

Important: Much of what is provided by the externals API is now supported by the foreign function interface in Builder, where it is not necessary to write any glue code in C. While the externals interface is still supported, Builder is the recommended way to wrap native code for this program.

Note: Some aspects of the built-in functionality are supplied in the form of externals. These include the SSL library, the database library, the revBrowser library, zip library, video grabber, and XML libraries. These libraries can be included in a standalone application, or excluded if not needed saving disk space.

The Externals SDK
Occasionally the Builder may not be what's needed to create your extension. A legacy developer kit is provided for writing externals in C++ which includes documentation and examples.

The following newsletter articles will help you get started:
External Writing for the Uninitiated - Part 1
http://newsletters.livecode.com/november/issue13/newsletter5.php
External Writing for the Uninitiated - Part 2
http://newsletters.livecode.com/november/issue14/newsletter3.php
Writing Externals for Linux with 2.9 or later
http://newsletters.livecode.com/october/issue34/newsletter1.php
==========

APPENDIX A) KEYBOARD SHORTCUTS REFERENCE
o Mac Only
Preferences: cmd+,
Hide: cmd+H
Quit: cmd+Q

o File Menu
Open Stack: ctrl/cmd+O
Close: ctrl/cmd+W
Save: ctrl/cmd+S
Save As: shift+ctrl/cmd+S
Print Card: ctrl/cmd+P

o Edit Menu
Undo: ctrl/cmd+Z
Cut: ctrl/cmd+X
Copy: ctrl/cmd+C
Paste: ctrl/cmd+V
Paste Unformatted: alt+shift+ctrl+V , option+shift+cmd+V
Duplicate: ctrl/cmd+D
Select All: ctrl/cmd+A
Find: ctrl/cmd+F

o Tools Menu
Browse: ctrl/cmd+9
Pointer: ctrl/cmd+0
Tools: ctrl/cmd+T
App Browser: shift+ctrl/cmd+A
Project Browser: shift+ctrl/cmd+P
Message Box: ctrl/cmd+M

o Object Menu
Object Inspector: alt+shift+I , option+shift+I , or returnKey , or double-click selected object
Card Inspector: ctrl/cmd+J

Stack Inspector: ctrl/cmd+K
Object Script: ctrl/cmd+E
Card Script: alt+shift+C , option+shift+C
Stack Script: alt+shift+S , option+shift+S
Group Selected: ctrl/cmd+G
Edit Group: ctrl/cmd+R
New Card: ctrl/cmd+N
Move Back: ctrl/cmd+[
Move Forward: ctrl/cmd+]

o Text Menu
Plain: ctrl/cmd+;
Bold: ctrl/cmd+B
Italic: ctrl/cmd+I
Underline: ctrl/cmd+U

o View Menu
Toolbar Icons: shift+ctrl/cmd+I
Toolbar Text: shift+ctrl/cmd+T
Backdrop: shift+ctrl/cmd+B

o The Development Environment
Run (browse) tool: ctrl/cmd+9
Edit (pointer) tool: ctrl/cmd+0
Hide/show palettes: ctrl+tab , cmd+ctrl+tab
Display context menus: shift+ctrl+rightClick , shift+cmd+ctrl+click
Save all open stacks: alt+ctrl+S , option+cmd+S
Save A stack: ctrl/cmd+S
Save As: shift+ctrl/cmd+S
Apply default button in Save dialog: returnKey
Apply non-default button in Save dialog: ctrl/cmd+1st letter of button

o Navigation
Go first card: ctrl/cmd+1
Go previous card: ctrl/cmd+2
Go next card: ctrl/cmd+3
Go last card: ctrl/cmd+4
Go recent card: ctrl/cmd+5
Go top or bottom of field: ctrl/cmd+upArrow/downArrow

o Objects
Select all: ctrl/cmd+A
Duplicate: ctrl/cmd+D
New Card: ctrl/cmd+N
Nudge: ArrowKeys
Nudge by 10 pixels: shift+arrowKeys
Open selected object inspector: returnKey , or double-click object , or alt+shift+I , option+shift+I
Open card inspector: ctrl/cmd+J
Open stack inspector: ctrl/cmd+K
Remove styles from selected text: ctrl/cmd+;
Equalize width of selected: ctrl/cmd+=
Equalize heights of selected: shift+ctrl/cmd+=
Magnify image with paint tool: ctrl+rightClick , cmd-click
Apply transparency with paint tool: ctrl/cmd+click
Constrain paint tool selection to a square: shift

Constrain object aspect ratio: shift

o The Code Editor
Edit script of selected object: ctrl/cmd+E
Edit card script: shift+alt+C , shift+option+C
Edit stack script: shift+alt+S , shift+option+S
Edit script of object under mouse: alt+ctrl+click , option+cmd+click
Apply changes: returnKey or enterKey
Apply changes and close: returnKey twice or enterKey twice
Apply changes and save stack: ctrl/cmd+S
Comment out selected lines: ctrl/cmd+- (hyphen)
Remove comments: shift+ctrl/cmd+- (hyphen)
Switch to find mode: ctrl/cmd+F
Find next: ctrl/cmd+G
Find selected text: alt+ctrl+F , option+cmd+F
Format current handler: tab

o The Message Box
Open/close message box: ctrl/cmd+M
Clear message field: ctrl/cmd+U
Scroll through recent messages (single-line): upArrow/downArrow
Scroll through recent messages (multi-line): alt+upArrow/downArrow , option+upArrow/downArrow
Execute message: returnKey
Execute message (multi-line): ctrl/cmd+returnKey

o The Debugger
Run: F5
Stop: shift+F5
Step Over: F10
Step Into: F11
Step Out: shift+F11
Abort: ctrl/cmd+.
==========