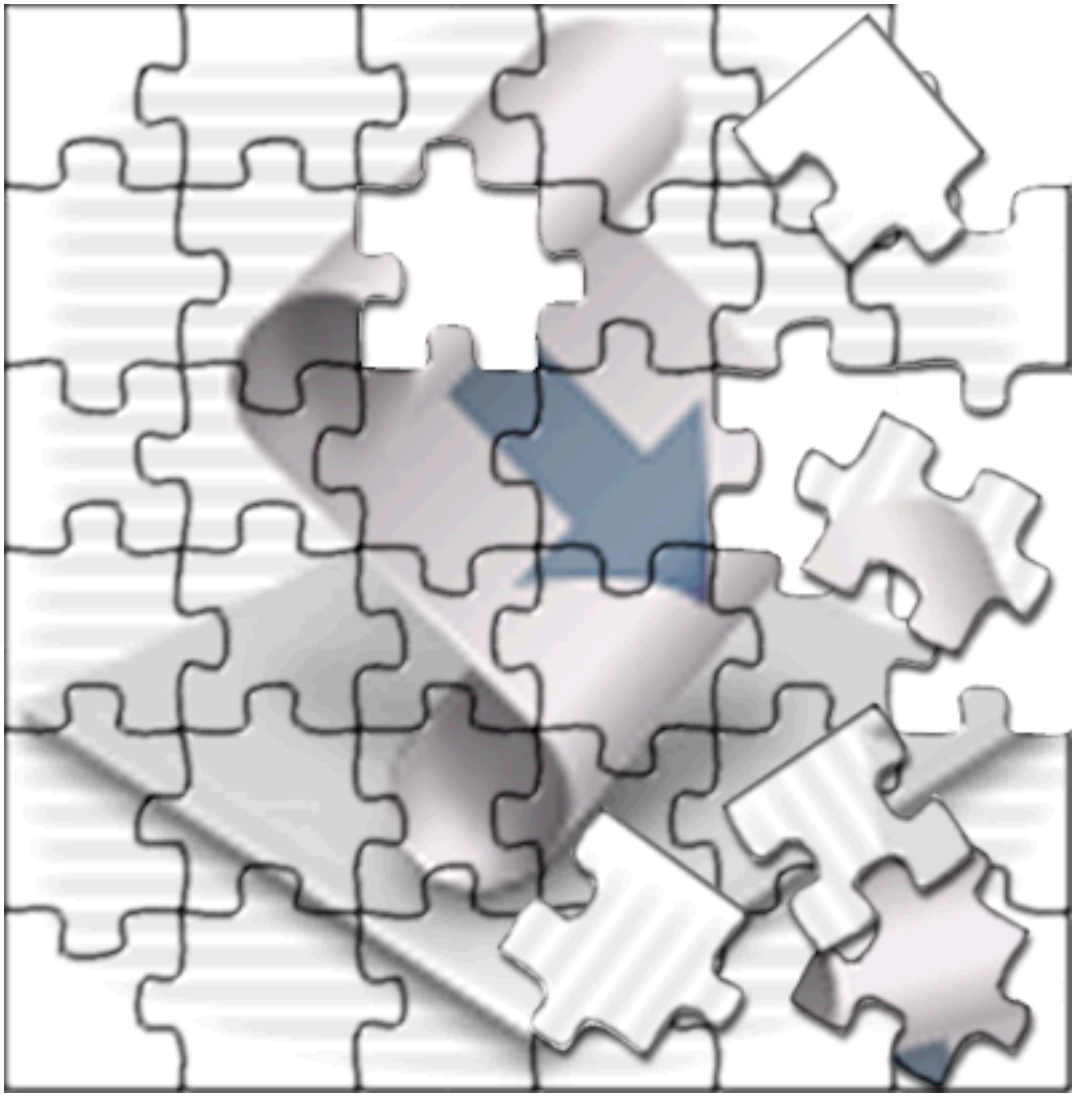
APPLESCRIPT FOR ABSOLUTE STARTERS



By Bert Altenburg
Cover graphics: Peter Fischer

INTRODUCTION

AppleScript is a revolutionary Apple technology that makes communication between computer programs possible. For example, with AppleScript you can
- retrieve e-mails from Mail and store them in a database;
- tell a picture editing program to change the resolution of a series of pictures, resize them, and send the resulting pictures to another computer or post them on the Web;
- and much, much more.

An AppleScript, or script for short, is a series of written instructions in a scripting language named AppleScript. This language resembles the English language, making AppleScripts both easy to read, write and understand.

Despite its power, AppleScript is heavily used in a couple of fields only. The publishing industry depends on it for workflow automation (PhotoShop, QuarkExpress, InDesign). Filemaker Pro developers use it for creating Mac-based kiosks, which you can find in malls and museums (k-Builder). Apart from the programs mentioned, many more major and minor Macintosh programs like GraphicConverter, BBEdit, and Word are AppleScriptable. That means you can use AppleScript to boss these programs around. Scripting applications is not the focus of this book, however. There are other books on the market that show you how to do that. If these books provide an introduction to AppleScript, it is usually cursory and they quickly dash to the really juicy stuff, which generally requires a modest or good knowledge of the basics of AppleScript. The aim of this book is to provide you just that.

It is intended to update and expand this book on a regular basis. So, you may want to check for new versions (see Chapter 15). A second book on scripting various programs is considered. This book is freeware, and you are encouraged to bring it to the attention of other Macintosh users. In this respect, please pay attention to Chapter 0 in this book on how you can promote the Mac.

Once you dive into the world of AppleScript, you'll notice that the term 'AppleScript' is used quite loosely for three different concepts.

- The AppleScript language: The English-like language which is used to give written instructions to your Mac;
- An AppleScript: A series of instructions, a.k.a. a script, written in the AppleScript language; and
- A part of the Mac operating system (Mac OS X), which actually reads an AppleScript and executes the instructions containing it.

In this book, if there is need to refer to one of these three concepts specifically, the following terms are used respectively:
- The AppleScript language;
- An AppleScript or a script (noun);
- The AppleScript component of Mac OS X.

Learning how to script with AppleScript is ideal as an introduction to programming. It leaves out most of the nitty-gritty work a programmer in a computer language such as Java has to do before she can even perform the easiest of tasks. AppleScript is easy enough that a 10 year

old can learn it, yet so powerful that professionals enjoy it too. That leaves plenty of room for growth for you. While not covered in this book, you can even use AppleScript to build computer programs that look and work just like the commercial programs you use on your Mac, with buttons, menus, scroll bars and all. This requires AppleScript Studio, provided for free by your favorite computer company.

What is the difference between scripting and programming? I'd like to think that if it is easy, it is scripting and if it is difficult, it is programming. However, javascripting is not easy in my book, so perhaps that definition is wonky.

How to use this book?
As you will see, some paragraphs are displayed in a green font. We suggest you read each chapter (at least) twice. The first time, skip the green text. The next time you read the chapters, include the green paragraphs. You will in effect rehearse what you have learned, but learn some interesting tidbits which would have been distracting the first time. By using the book in this way, you will level the inevitable learning curve to a gentle slope.

This book contains dozens of script examples. To make sure you link an explanation to the proper script, every script is labeled by a number placed between square brackets, like this: [4]. Most scripts consist of two or more lines. At times, a second number is used to refer to a particular line. For example, [4.3] refers to the third line of script [4].

You will not learn riding a horse by reading a book. Similarly, you will not learn AppleScript if you don't get your mitts on your Mac. This is an electronic book. You have no excuse for not switching to the Script Editor (see Chapter 2).

CHAPTER 0

BEFORE WE START

I wrote this book for you. As it is free, please allow me to say a couple of words on promoting the Mac in return. Every Macintosh user can help to promote their favorite computer platform with little effort. Here is how.

1) The more efficient with your Mac you are, the easier it is to get other people to consider a Mac. So, try stay up to date by visiting Mac-oriented websites and reading Mac magazines. Of course, learning AppleScript and putting it to use is great too. For companies, the use of AppleScript can save tons of money and time.

2) Show the world that not everybody is using a PC by making Macintosh more visible. Wearing a neat Mac T-shirt is one, but you can even make the Mac more visible from within your home. If you run CPU monitor (in the Utilities folder which you find in the Applications folder on your Mac), you will notice that your Mac uses its full processing power only occasionally. Scientists have initiated several distributed computing (DC) projects, such as Folding@home, that harness this unused processing power. You download a small, free program, called a DC client, and start processing work units. These DC clients run with the lowest level of priority. If you are using a program on your Mac and that program needs full processing power, the DC client immediately takes a back seat. So, you will not notice it is running. How does this help the Mac? Well, most DC projects keep rankings on their websites of work units processed. If you join a Mac team (you'll recognize their names in the rankings), you can help the Mac team of your choice to move up the rankings. So, users of other computer platforms will see how well Macs are doing. There are DC clients for many topics, such as math, curing diseases and more. To choose a DC project you like, check out

www.aspenleaf.com/distributed/distrib-projects.html

One problem with this suggestion: It may become addictive.

3) Make sure the Macintosh platform has the best software. No, you don't have to learn programming. Just make it a habit to give (polite) feedback to the developers of programs you use. Even if you tried a piece of software and didn't like it, tell the developer why. Report bugs by providing a description of the actions you performed when you experienced the bug. For a great free multimedia-based tutorial on how to do this, visit

www.macinstruct.com/tutorials/crash/index.html

4. Pay for the software you use. As long as the Macintosh software market is viable, developers will provide the software.

5. Please contact at least 3 Macintosh users who don't know about this book and tell them where to find it. Or advise them about the above 4 points.

OK, while you download a DC client in the background, let's get started!

CHAPTER 1

A SCRIPT IS A SERIES OF INSTRUCTIONS

AppleScript as part of the Macintosh Operating System can perform only a very limited number of tasks. For example, it can produce a beep. Let's take a look at the script [1] needed to make your Mac beep.

[1]
```
beep
```

This must be the world's shortest script, consisting of a single command or instruction. A line containing an instruction is called a statement, even if that line is just one word long. If the above script is executed by your Mac, your Mac beeps once.

To have more beeps than just one, you may provide the beep command with a number, which number indicates the number of beeps you want to hear [2].

[2]
```
beep 2
```

As you can see by comparing scripts [1] and [2], this additional piece of information is optional. If you don't provide a number, AppleScript assumes you want just one beep. So, 1 is the default value.

If you think beeps are PeeCee-ish, why don't we let AppleScript communicate with you the Macintosh way [3], using the following statement:

[3]
```
say "This is a spoken sentence."
```

You may even select another voice, such as "Fred", "Trinoids", "Cellos", or "Zarvox" [4], to replace the default voice "Victoria".

[4]
```
say "This is a spoken sentence." using "Zarvox"
```

#Note: Generally, AppleScript is not case sensitive. That is, it doesn't mind if you use capitals or not. However, the voices, such as "Victoria", and "Zarvox" must be properly capitalized. Grrr.#

As you can see, AppleScript instructions resemble English, making the script quite readable and understandable, even if you have never had any scripting experience. But while the scripts [1-4] are probably fun; they are not very useful. The AppleScript language has a couple more commands, but probably not much to impress you. AppleScript derives its strength from the fact that it allows you to communicate with other programs. This works if these programs are AppleScriptable. Fortunately, many Macintosh programs are. As a result, you have at your disposal not only the, granted, limited command set of the AppleScript component of Mac OS X, but also the vast number of commands provided by your programs.

Some Mac programs are more popular than others. One is used by every Macintosh user: the Finder. Yes, the Finder is a program. When you turn on your Mac, it starts-up automatically

and it is always running. It allows you to move files around, find files on your hard disk, create folders, copy and rename them and much more. For example, if you empty the trash, it is the Finder that does it for you. While you can perform the empty-the-trash operation with the mouse or keyboard, you can do it with an AppleScript [5] too.

[5]
```
tell application "Finder"
	empty the trash
end tell
```

Like a boss, you must tell
- who is to perform a task, and
- which task is to be performed.
It is no use telling, for example, PhotoShop that it is to empty the trash. PhotoShop does not know how to do that. So, the instruction to empty the trash must be conveyed to the Finder.

Like in the real world, the job a boss has ordered you to do may be less than wise, but your Mac is a most faithful employee, and does what it is told. If there were an important file in the trash, once you have executed the above AppleScript [5], you have lost it forever.

The first statement [5.1] is the 'tell' statement where we ask the AppleScript component of Mac OS X to convey one ore more statements to another program, here to the Finder. The AppleScript component of Mac OS X keeps doing that, until it encounters the obligatory 'end tell' statement [5.3]. In the above script [5] we order AppleScript to send the Finder the instruction to empty the trash and then to stop telling the Finder what to do. Taken together, the lines

```
tell application "xyz"

end tell
```

are called a 'tell block'. The instruction to be executed by program 'xyz' is inside the tell block for program 'xyz'. By the way, while the AppleScript language is not very finicky when it comes to notation in comparison with other scripting and especially programming languages, it is not without a couple of rules. One of the rules is that you must use double quotes around the application's name, as in the first statement [5.1].

It is also possible to give the Finder more instructions. In the example [6] below, there are two statements [6.2, 6.3] destined for the Finder. Because they both are to be performed by the Finder, they must be inside the tell block for the Finder.

[6]
```
tell application "Finder"
	empty the trash
	open the startup disk
end tell
```

After emptying the trash, the Finder opens a window showing you the content of your hard disk.

As you can see, we can make the Finder do whatever we want to. We can even tell the Finder to resize the Finder window, put it at a desired position on the screen and way, way more. You will learn how to do that later on.

We can now create a script containing both instruction statements for the Finder, and for the AppleScript component of Mac OS X itself [7].

[7]

```
tell application "Finder"
	empty the trash
	open the startup disk
end tell
beep
```

First, the Finder is given a couple of instructions [7.2, 7.3]. Then the 'beep' instruction [7.5] is executed by AppleScript. In effect, it indicates that the script has been executed.

Interestingly, it doesn't matter whether you put the beep instruction (and any other instruction understood by the AppleScript component of Mac OS X itself) inside or outside the tell block [8].

[8]

```
tell application "Finder"
	empty the trash
	beep
	open the startup disk
end tell
```

While the Finder doesn't know the beep command, the AppleScript component of Mac OS X knows how to deal with it. This makes the script easier to read and understand. Otherwise, you would have to have a first tell block containing the first Finder-executable statement [8.2], then a statement consisting of the beep command, and finally a second tell block for the last Finder-executable statement [8.4].

Mind you, while commands understood by the AppleScript component of Mac OS X may be anywhere in a script, each and every instruction for a particular program, such as the Finder, must be within the tell block for that program. The following script [9] contains a fatal flaw (the last statement [9.5]).

[9]

```
tell application "Finder"
	empty the trash
	beep
end tell
open the startup disk
```

The AppleScript component of Mac OS X does not know how to open the startup disk, and is not willing to look for a program that can do it. The first part of this script (statements [9.2-3] within the tell block) will be executed nicely, but the last statement [9.5] can not be executed.

In a running script, once a problem is encountered, any further statements are not executed [10].

[10]

```
tell application "Finder"
	empty the trash
end tell
open the startup disk
say "I emptied the trash and opened the startup disk for you" using "Victoria"
```

After emptying the trash, the AppleScript component of Mac OS X will stop at the statement [10.4] that should have been addressed to the Finder. You will not hear the sentence of statement [10.5] being spoken, even though there is nothing wrong with the statement.

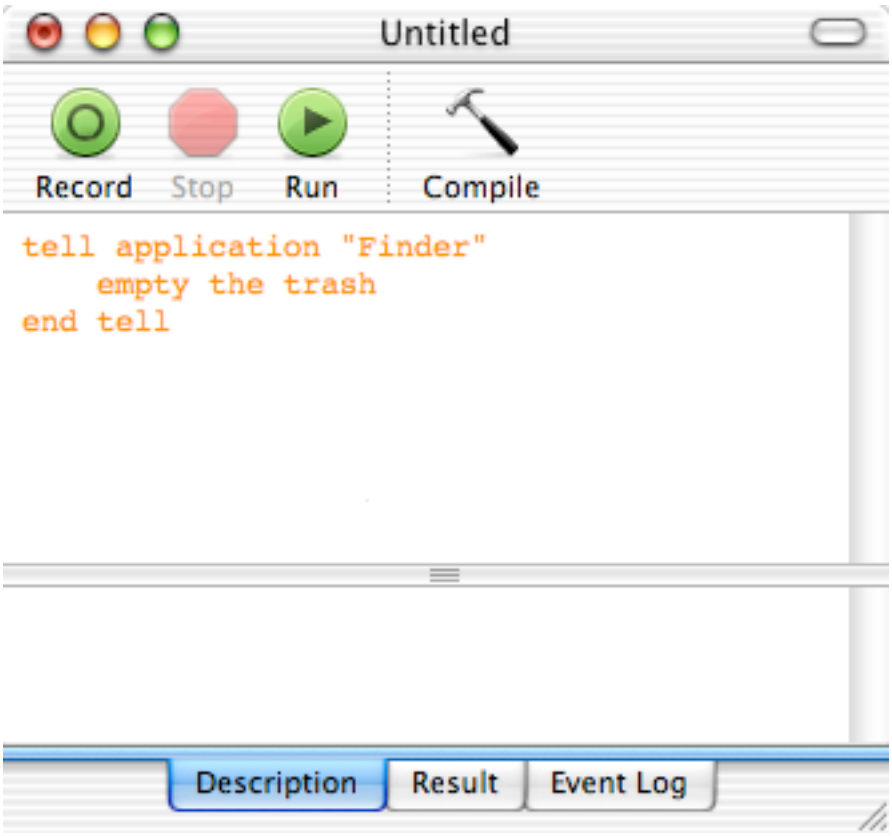
CHAPTER 2

EXECUTING AND SAVING A SCRIPT

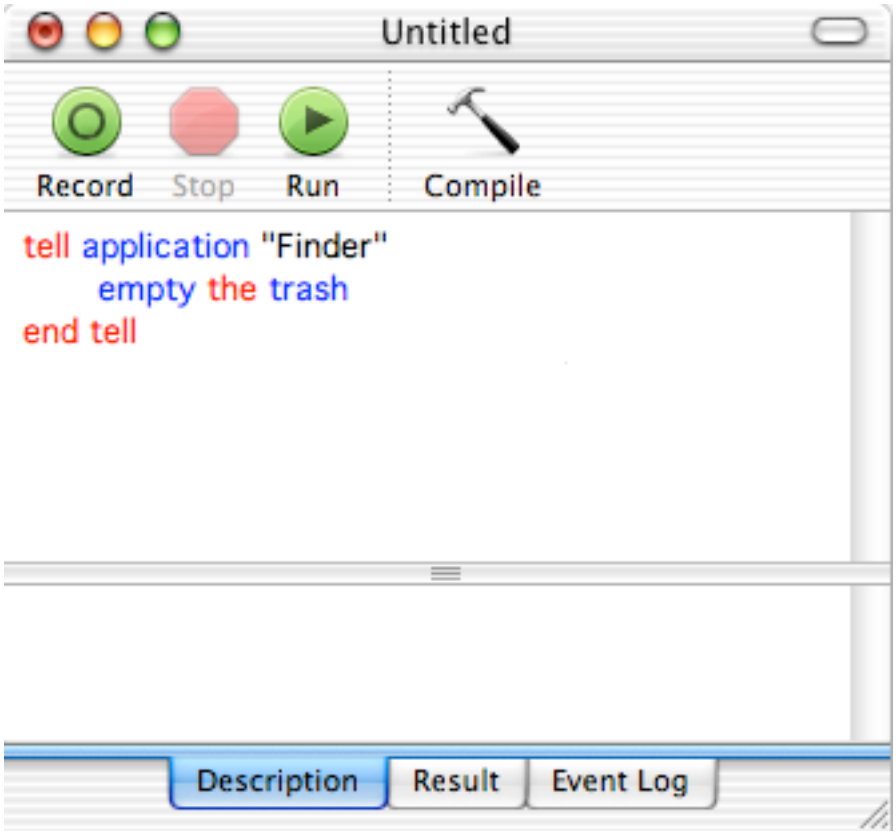
You have seen a couple of scripts now, and there is no denying that they are very similar to English, making the scripts easy to read and understand. You could have performed commands in the script - like empty the trash - yourself, using mouse and keyboard. Let's see how your Mac can do that for you.

The Script Editor is a program where you can type in a script and execute it. You can find the Script Editor in the AppleScript folder that in turn is in the Applications folder. After starting it up, you will see two fields. The upper one is where you should enter the script [1].

[1]



Near the middle you will see a button labeled 'Compile'. While an AppleScript may look like English, the AppleScript language is far from fully versed in English. "Yo Finder! Dump my garbage" or "Hey Finder, clean out the bin" is not what the Finder expects. By a process called 'compilation', the AppleScript component of Mac OS X performs a course check whether it understands your script. If it thinks it does, it formats your script in a nice, colorful way. Uncompiled text is shown in orange, while after compilation the reserved keywords are displayed in red or blue.



If the script does not compile because you made a mistake, you will see a cryptic message indicating that there is something wrong. Try leaving out one of the double quotes in script [2] and see for yourself that the AppleScript component of Mac OS X no longer understands what you mean.

[2]
```
say "I'm learning AppleScript the easy way!" using "Zarvox"
```

If all is well, you may press the Run button, and your script is executed. Now fire up the Script Editor, peck in one of the scripts you've seen here, and try it!

#You may hit the Enter key as a shortcut to compile your script. The Enter key is the key at the right next to your spacebar (for laptops) or numerical keypad (desktop Macs). The Return key (above the right shift key) works as you would expect, and creates a new line after the current line. You can not use the Return key to compile your script.

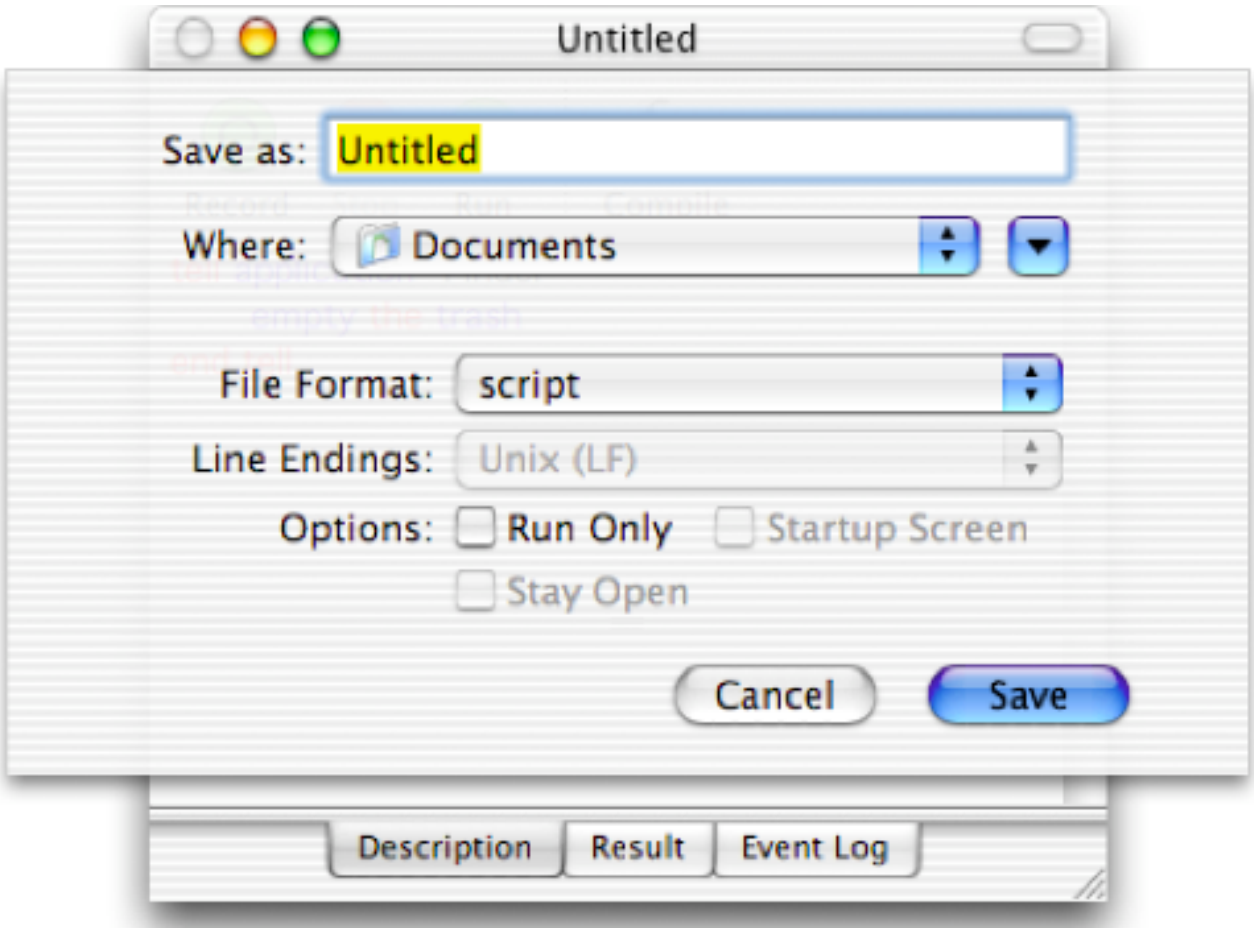
It is not really necessary to press the Compile button before you run the script. If you press the Run button, the script's syntax is checked and, if OK, the script is immediately executed.

Instead of clicking the Run button, you may press Command-R.

Note: In reality, compiling involves more than just checking the syntax, but that is not your concern.
#

Saving your script
There are several ways to save your script. If the script could not be compiled successfully, your only option is to save your script as mere text.

If there were no problems during compilation, the dialog window below will appear, and you can save your text as a compiled script or as an application.



COMPILED SCRIPT: If you double-click the icon of your saved, compiled AppleScript,

the Script Editor is opened and you can run the script by pressing the Run button.

APPLICATION: If you double-click the icon of your saved, AppleScript application,

the script will be executed immediately. That is, the Script Editor is not opened. Saved as an application, you may use the script as a log-in item (in your System Preferences). After logging in, your Mac will perform the tasks detailed in your script. If you need to edit a script saved as an application, startup the Script Editor, and open the script by selecting Open from the File menu.

#WARNING: Setting the appropriate tick in the Save dialog allows you save your script: as run-only. Make sure you have a backup of your script, because a run-only script can not be opened and edited again. #

CHAPTER 3

EASIER SCRIPTING (I)

In Chapter 1 you came across the following script:

[1]
```
tell application "Finder"
	empty the trash
end tell
```

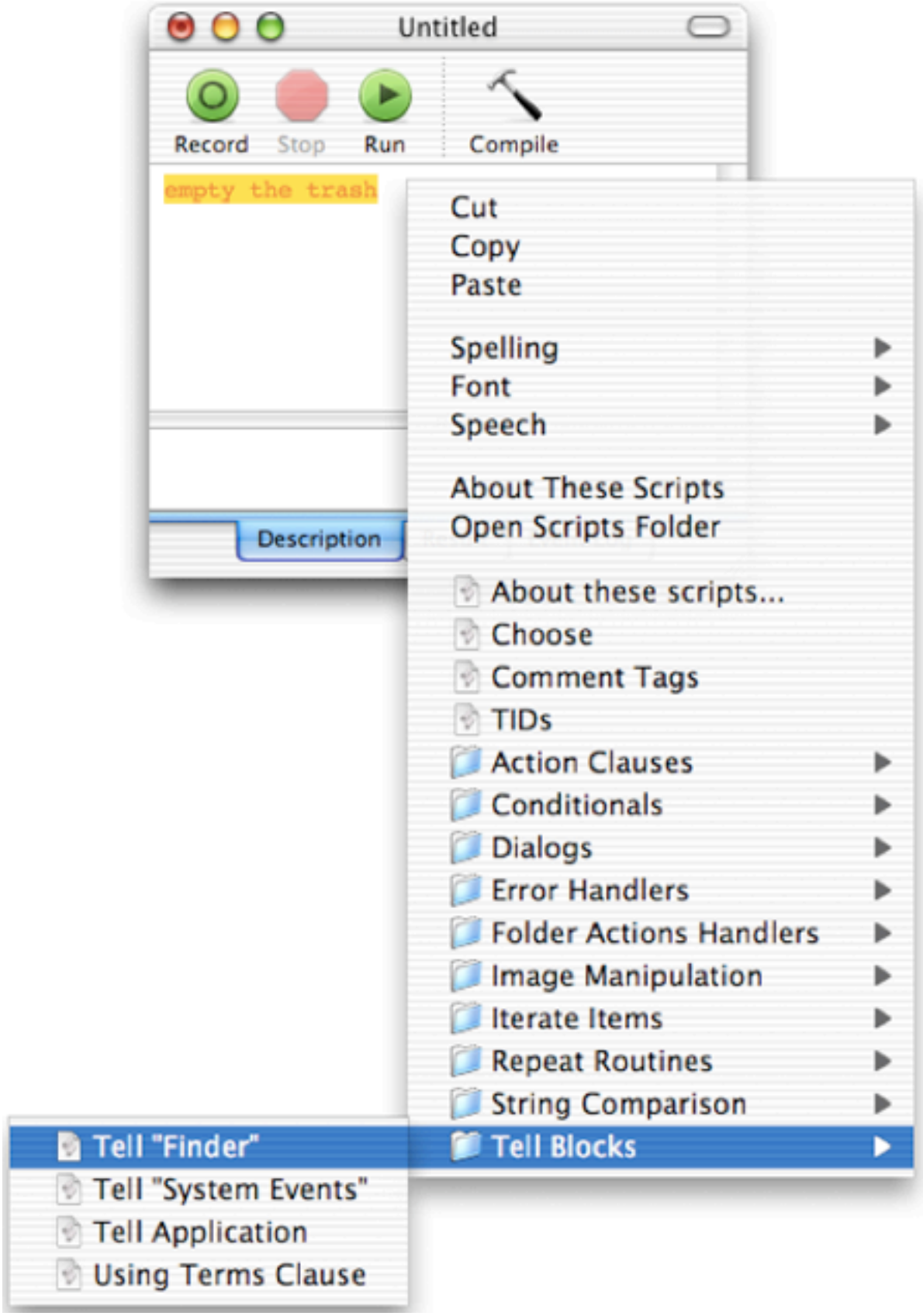
Let us see how AppleScript and the Script Editor try to make it easy for you to script.

In the first line of a tell block, instead of typing the word ‘application’ in full, you may write

```
tell app “xyz”
```

Upon compilation, the Script Editor expands ‘app’ to ‘application’. Even better, you do not need to type or know how to spell the name of the application ‘xyz’ either. Just type anything (provided it is not the name of another application), such as ‘pqr’. If you compile the script, AppleScript will provide you with a list of all scriptable applications on your Mac. You just pick the appropriate application, and AppleScript will replace ‘pqr’ with the correct name of the application, and in effect finish writing the tell statement for you.

Actually, the Script Editor allows you to create a tell block without any typing, by using a contextual menu. That is the kind of menu that appears if you hold down the control-key while clicking. This trick can be performed in two ways:
1) Control-click the upper field of the Script Editor. A menu appears (see picture on the next page), and near the lower end of this menu, you will see a menu item named ‘Tell Blocks’. Click it and a submenu appears from which you can select ‘Tell “Finder”’.
2) If your script already contains one or more statements for the Finder - such as ‘empty the trash’ - which are not yet enclosed by the required tell block, select the statement(s), and then do as under 1). You can see this in action in the picture below. Your statements are automatically enclosed by the tell block.


Untitled
Record
Stop
Run
Compile
empty the trash
Description
Cut
Copy
Paste
Spelling
Font
Speech
About These Scripts
Open Scripts Folder
About these scripts...
Choose
Comment Tags
TIDs
Action Clauses
Conditionals
Dialogs
Error Handlers
Folder Actions Handlers
Image Manipulation
Iterate Items
Repeat Routines
String Comparison
Tell Blocks
Tell "Finder"
Tell "System Events"
Tell Application
Using Terms Clause

CHAPTER 4

DEALING WITH NUMBERS

In primary school you had to do calculations, filling in the dots:

2 + 6 = ...
... = 3 * 4

In secondary school, dots were out of fashion and variables called 'x' and 'y' were all the hype. Looking back, you may wonder why people felt so intimidated by this very small change in notation.

2 + 6 = x
y = 3 * 4

AppleScript uses variables too. Variables are nothing more than convenient names to refer to a specific piece of data, such as a number. Variable names are often referred to as 'identifiers', because they identify a piece of data. Here are two examples [1] of AppleScript statements where a variable is given a particular value, using the 'set' command.

[1]
```
set x to 25
set y to 4321.234
```

While the variable names themselves have no special meaning to AppleScript, to us human beings descriptive variable names can make a script much easier to read and hence easier to understand. That is a big bonus if you need to track down an error in your script (errors in scripts and programs are traditionally called 'bugs'). Hence, avoid using non-descriptive variable names like 'x'. For example, the variable name for the width of a picture could be called "pictureWidth" [2].

[2]
```
set pictureWidth to 8
```

Please note that a variable name consists of a single word (or single character, at a pinch). After checking the syntax, the variable name is displayed in green, so you can immediately see it is not a reserved keyword of AppleScript, which are shown in blue or red. Also, note that data (such as the number '8' in script [2]) is shown in black.

#While you have plenty freedom choosing variable names, there are several rules which a variable name has to conform with. While I could spell them all out, it would be boring. The prime rule you must obey is that your variable name may not be an AppleScript command or any other reserved keyword. For example, 'set', 'say','to', and 'beep' are words that have a special meaning to AppleScript. By compositing a variable name as contracted words, like 'pictureWidth', you are always safe. To keep the variable name readable, the use of capitals within the variable name is recommended.

If you insist on learning a couple of rules, finish this paragraph. Apart from letters, the use of digits is allowed, but a variable name is not allowed to start with a digit. Also allowed is the underscore _.#

Now we know how to give a variable a value, we can perform calculations. AppleScript is capable of performing basic math operations, so there is no need to tell a particular program that it should perform the calculation to determine the surface area of a picture. Here is the script [3] that does just that.

[3]
```
set pictureWidth to 8
set pictureHeight to 6
set pictureSurfaceArea to pictureWidth * pictureHeight
```

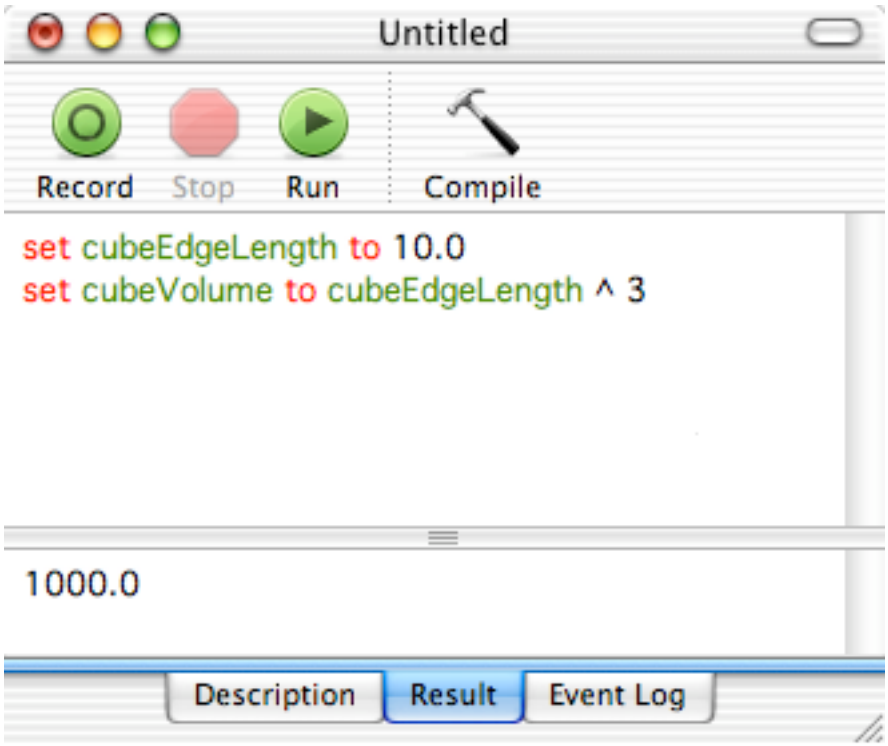
Use the following symbols, officially known as operators, for doing basic mathematical calculations.

+	for addition
-	for subtraction
/	for division
*	for multiplication

The exponent is written using the exponent symbol ^. Here is a script [4] that calculates the volume of a cube.

[4]
```
set cubeEdgeLength to 10.0
set cubeVolume to cubeEdgeLength ^ 3
```

If you run this script [4] in the Script Editor, the result is displayed in the lower field. If you don't see the result, move the horizontal bar near the two tabs up. The result field displays the result of the last statement executed. If your script consists of statement [4.1] only, the result field displays 10.0. For the full script [4], the result displayed is 1000.0. That is, the expression 'cubeEdgeLength^3' is evaluated, and the result is displayed.



Numbers can be distinguished into two types: integers and fractional numbers. You can see an example of each in the statements [1.1] and [1.2], respectively. Integers are used for

counting, which is something we will do when we have to repeat an series of instructions a specified number of times (see Chapter 13). You know fractional or real numbers ('reals' for short), for example, from baseball hitting averages. By the way, both integers and reals can be negative, as you know for example from your bank account.

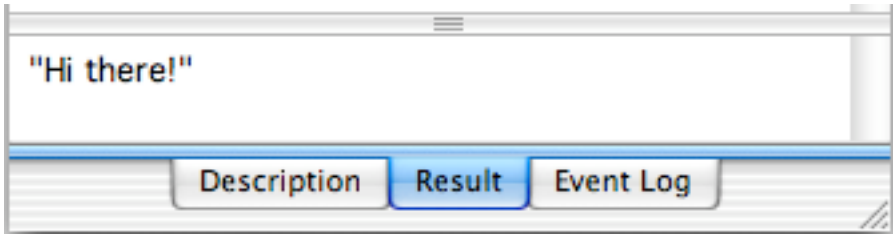
CHAPTER 5

DEALING WITH TEXT

Variables are not only used for storing numbers. They can be used to store text as well. A chunk of text, even if it is only zero or a single character long, is called a string. Strings must be placed between double quotes. Here are three examples [1] of variables with descriptive names set to a corresponding string value.

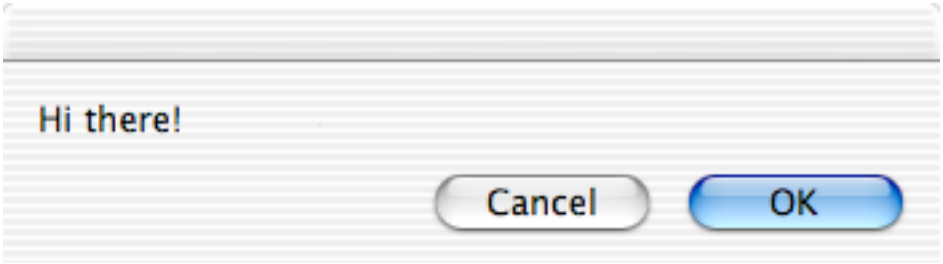
[1]
```
set emptyString to ""
set notEmtpyContainsASpace to " "
set greeting to "Hi there!"
```

After running script [1], the result field displays the string between double quotes. So, the result field not only communicates the value, but also the type of data (numbers without quotes; strings with quotes) to you.



Because the result field can show the result of the last executed statement only, just the string comprising the greeting of statement [1.3] is displayed.

In addition to the result field, AppleScript offers a convenient alternative to communicate with the world: a dialog window. It looks like this:



You can evoke it using the command: 'display dialog' followed by the data (number or string) you want to show. The above dialog was created using the following script [2].

[2]
```
display dialog "Hi there!"
```

Why do strings have double quotes? AppleScript may comprise a very limited vocabulary only, but reading your script and deciphering what is an instruction and what is not, is still pretty

hard for a computer. So, AppleScript relies on clues to help it understand the meaning of each element of a statement of a script. For that reason, we have to put a string between double quotes. Otherwise, AppleScript could mistake a string for a variable name. Check out the following script [6]:

[6]
```
set stringToBeDisplayed to "Hi there!"
display dialog "stringToBeDisplayed"
display dialog stringToBeDisplayed
```

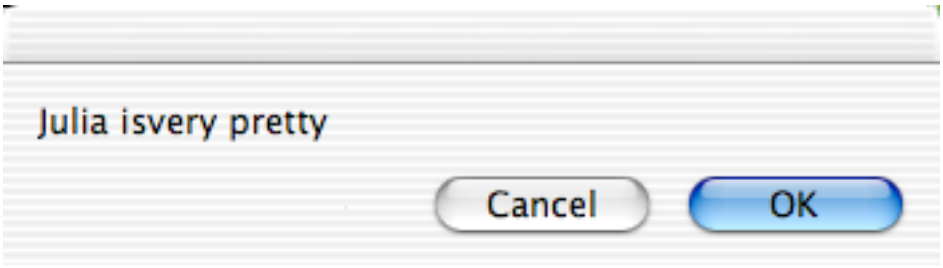
Run the script yourself to see what it does. Statement [6.2] displays the text 'stringToBeDisplayed', whereas statement [6.3] displays 'Hi there!' Because the Script Editor shows compiled scripts in color, it is easy to see that while in statement [6.3] 'stringToBeDisplayed' is a variable name, for which reason it is shown in green, in statement [6.2] 'stringToBeDisplayed' is shown in black, indicating that this word is data (a string). At times, the colorful formatting will help you to track down bugs quicker.

As stated before, AppleScript needs clues to decipher the English-like script into something a Mac can understand. Here is another example of why these clues are important: If we write 'thirty' as a number between double quotes, i.e. as "30", it is not a number anymore but a string. Recognizing the data type is very important, because some operations can only be performed on a specific data type. For example, while you can divide two numbers, you can't divide a string by another string. Let's look at a couple of operations that can be performed on strings.

Like numbers (and voters), strings can be manipulated. You can glue strings together, which operation is called concatenation, using an ampersand [7].

[7]
```
set nameOfActress to "Julia"
set actressRating to "very pretty"
set resultingString to nameOfActress & " is" & actressRating
display dialog resultingString
```

In the third statement [7.3], we concatenate three strings, two of which are referred to by variables.



Please note that the number of spaces between a string and an ampersand is of no consequence for the resulting string contained by the variable 'resultingString'. After compiling, the Script Editor reduces that number of spaces to 1 if you had added more than one. If you need one or more spaces to separate the words of the sentence to be displayed, you will have to provide them between the double quotes of a string. In statement [7.3], apart from the space left of the word 'is' , there should have been another space next to the 's' of the word 'is' .

There are more commands acting on strings available. Some of them require stuff we will cover in later chapters, so we will leave that for the time being. But we can give you one other example of a command relating to strings. You can ask for the length of a string [8].

[8]

```
set theLength to the length of "I am"
```

If you run this script, the result displayed in the result field is 4. So, please remember that when the length of a string is calculated, spaces count too.

Because double quotes are used to signal the beginning and the end of a string, you might think that it is impossible to have a string containing quotes. Of course, the AppleScript language offers a way out, called 'escaping'. Just put a backslash before the double quote, and AppleScript no longer tries to interpret the double quote as the end of a string [9].

[9]

```
set exampleString to "She said: \"Hi, I'm Julia.\""
```

#If you ponder over it, that does leave the problem for the rare occasion that your string must contain a double quote preceded by a backslash. Suppose you want to display the following text:

blah blah \" blah.

First, we put a backslash in before of the backslash. This means AppleScript will ignore any special meaning of the next character, i.e. the second backslash. Of course, we still need to escape our double quote, otherwise AppleScript would think our string ends there. Hence, the double quote must be preceded by a backslash, like we have seen before. Taken all together, we arrive at the following statement [10].

[10]

```
display dialog "blah blah \\\" blah"
```

For your convenience, I have shown the escaping backslashes in bold, something the Script Editor does not do. Care should be taken with backslashes, because they can have special meaning before several other characters as well. For example, \n designates a new line (Return), and \t designates a Tab.#

As indicated above, numbers and strings are different data types. You can not subtract three from a string [11].

[11]

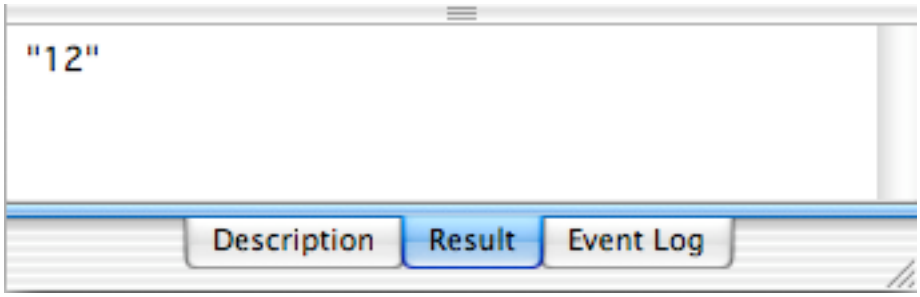
```
set nonsensical to "fifteen" - 3
```

If you try to execute script [11], you get an idea how friendly a scripting language AppleScript is. It actually tries to convert the string into a number. If the string had been "15", instead of "fifteen", this conversion would have worked. The conversion of one data type into another is called coercion. You can enforce this as shown by the following two example statements [12]

[12]

```
set coercedToNumber to "15" as number
set coercedToString to 12 as string
```

The result field of the Script Editor shows the result of the last statement, that is [12.2]. The data held by 'coercedToString' is a string, as the double quotes clearly indicate (see picture below).



For a script ending with the first statement [12.1], the result field would have shown 15 without double quotes, indicating that the result is a number and not a string.

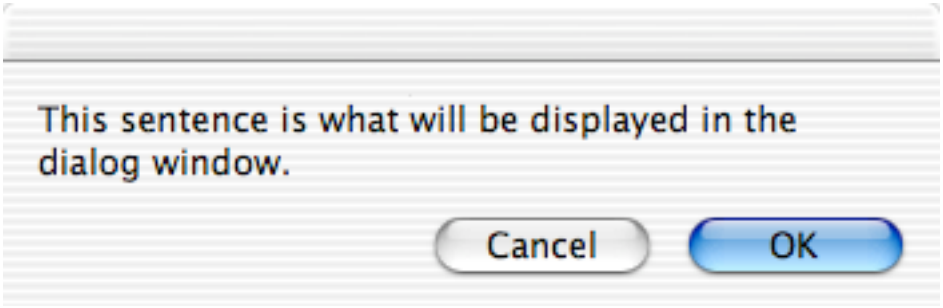
CHAPTER 6

LISTS

In the previous chapters, you have seen how to write very simple scripts to perform basic calculations and string operations. The user of the script could be presented the result with the 'display dialog' command [1]. In this and the next chapter we will use the 'display dialog' command to learn more about various aspects of the AppleScript language.

[1]
```
display dialog "This sentence is what will be displayed in the dialog window."
```

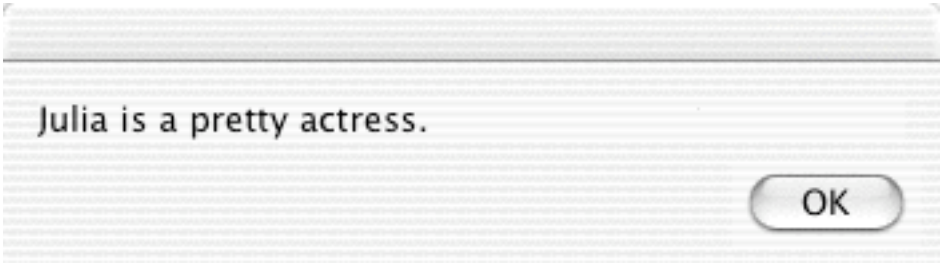
If you run script [1], you will see a dialog window that by default has two buttons: A Cancel button and an OK button.



The Cancel button halts further execution of the script. Because the above script does not have any further statements, the Cancel button is pretty superfluous. Let's get rid of it by defining the buttons of the dialog window ourselves. The display dialog command allows you to specify a list of buttons. In our case, we need only one, a button that we want to read 'OK' [2.2].

[2]
```
set stringToBeDisplayed to "Julia is a pretty actress."
display dialog stringToBeDisplayed buttons {"OK"}
```

If you run the script now, you will see that the Cancel button is gone.



As shown in the second statement [2.2], the list consists of the string "OK" placed between curly braces. Now why are those curly braces there? As we have seen before, AppleScript needs clues to help it understand the meaning of each element of a statement of a script. The

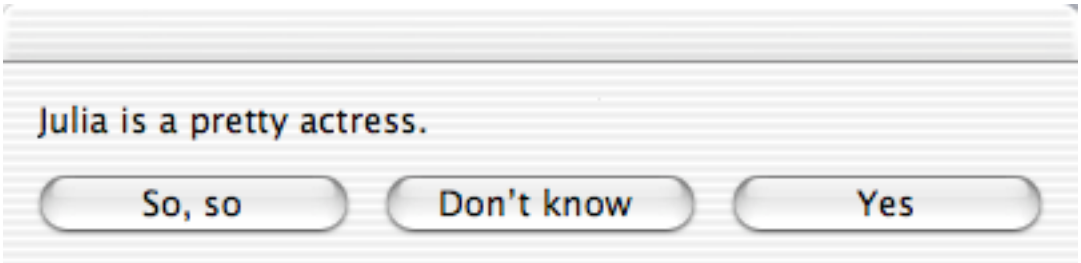
clue necessary to help AppleScript recognize a list, consists of the curly braces.

The list in statement [2.2] contains only one item, the string “OK”. If a list contains more items, they are separated by comma’s [3] .

[3]
```
set exampleList to {213.1, 6, "Julia, the actress", 8.5}
```

The list in the above statement [3] contains 4 items: One string and three numbers. Let's return to the display dialog command and create a dialog window with multiple buttons. The AppleScript display dialog command allows for one, two, or three buttons, with a (quite short) text of your own choice. So, to create a dialog window with three buttons, we have to specify a list with three items [4.2].

[4]
```
set stringToBeDisplayed to "Julia is a pretty actress."
display dialog stringToBeDisplayed buttons {"So, so", "Don’t know ", "Yes"}
```



#You will notice that no button is highlighted if you specify buttons yourself [2.2, 4.2] That means that the person who runs the script can’t hit the Enter button to dismiss the dialog window. Because that person is a Macintosh user who appreciates user-friendliness, we will now take care that a button is highlighted [5].

[5]
```
set stringToBeDisplayed to "Julia is a pretty actress."
display dialog stringToBeDisplayed buttons {"So, so", "Don’t know ", "Yes"}
                                                    default button "Yes"
```

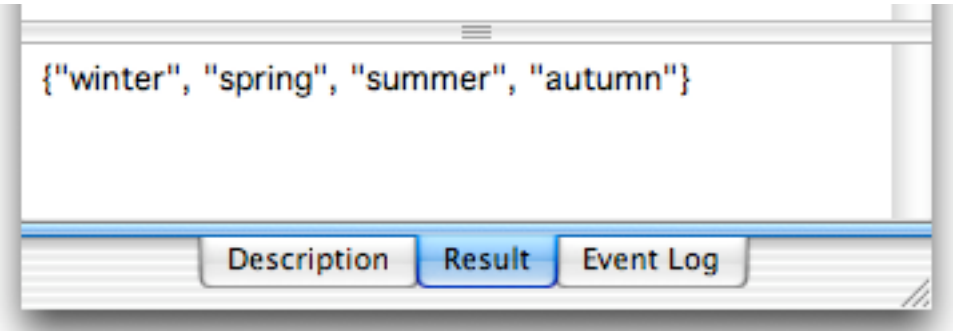
Instead of writing the name of the button to be highlighted out in full, you can refer to the number of the button [6], i.e. the third item in the specified list.

[6]
```
set stringToBeDisplayed to "Julia is a pretty actress."
display dialog stringToBeDisplayed buttons {"So, so", "Don’t know ", "Yes"}
                                                    default button 3
#
```

In the next chapter you will learn how to figure out which button has been pressed. For now we continue with learning more about lists. Lists can be used to store a series of data. So, you will need to know how to edit a list and retrieve data from the list. It is easy to add data to the beginning or end of a list. To add items to a list, we use the ampersand, like we have seen for strings.

[7]
```
set addToBegin to {"winter"}
set addToEnd to {"summer", "autumn"}
set currentList to {"spring"}
set modifiedList to addToBegin & currentList & addToEnd
```

In statement [7.4] we create a list consisting of four items. The result field displays the curly braces, characteristic of the data type 'list'.



Please note that 'addToBegin' and 'addToEnd' are just variable names [7.1-2], chosen to help you understand the script and they do not do anything except referring to data. The green color is a clear indication of they being variable names.

You can refer to each item in a list by a number. The leftmost item is item 1, the next is item 2 etc. This allows you to retrieve a particular value from a list, or change the value (such as a string or a number). Here is an example [8].

[8]
```
set myList to {"winter", "summer"}
set item 2 of myList to "spring"
get myList
```

The 'get' command in the last statement [8.3] allows us to show the value of the variable myList in the result field. This field will show the value of the variable myList as {"winter","spring"}

Focussing on the second statement [8.2], the same result would have been obtained with any of the following two statements of script [9] as the second statement in script [8].

[9]
```
set the second item of myList to "spring"
set the 2nd item of myList to "spring"
```

The first statement [9.1] shows elegantly AppleScript's English-like nature. This verbal numeration works up to the tenth item. After that, you have to resort to 'item 11', etc. Alternatively, you may write '11th item' , etc similar to [9.2]. Apart from referring to verbally numerated list items, there is also the 'last' item [10].

[10]
```
set myList to {"winter", "summer"}
set valueOfLastItem to the last item of myList
```

So, you don't need to know how long a list is to retrieve the value of the last item of a list (or to set it to another value).

AppleScript allows you to refer to items by counting in the opposite direction, i.e. from right to left, as well. To this end, use negative numbers, where item -1 is the last item, item -2 is the item before last etc. Script [11] yields exactly the same result as script [10].

[11]
```
set myList to {"winter", "summer"}
set valueOfLastItem to item -1 of myList
```

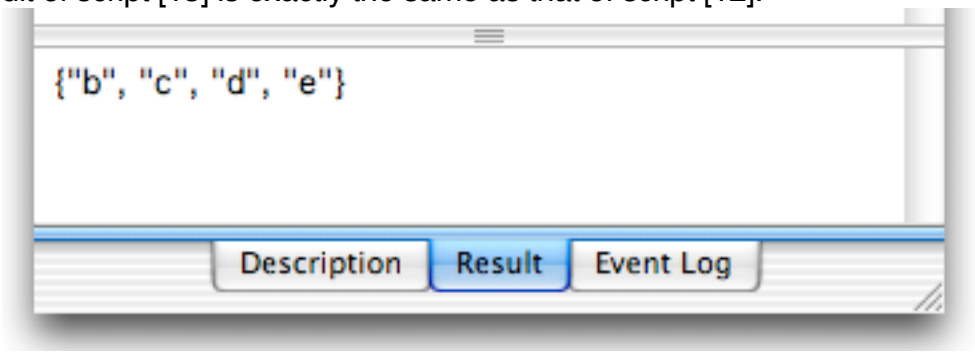
You now know how to create a list, how to add items to it, and how to change the values of list items. You also know how to retrieve a single value from a list. You will probably want to know how to create a part of a list too [12].

[12]
```
set myList to {"a", "b", "c", "d", "e", "f", "g", "h"}
set shortList to items 2 through 5 of myList
```

In statement [12.2], you may use 'thru' instead of 'through' if u r that type of person. If you reverse the order of the item numbers [13.2] in the specified range, your list will not be reversed [13].

[13]
```
set myList to {"a", "b", "c", "d", "e", "f", "g", "h"}
set shortList to items 5 through 2 of myList
```

So, the result of script [13] is exactly the same as that of script [12].

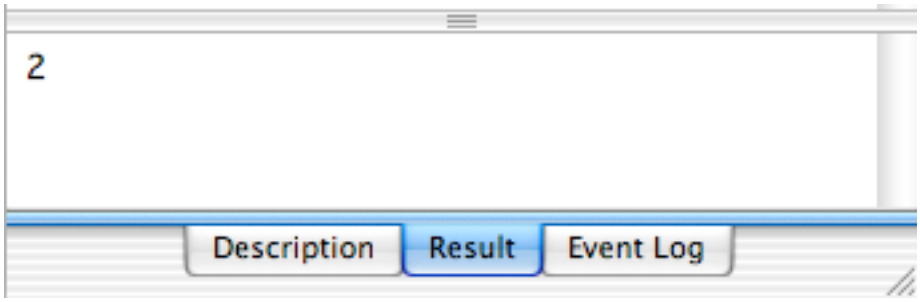


To reverse the order of the items in a list, you have the 'reverse' command at your disposal [14].

[14]
```
set reversedList to reverse of {3, 2, 1}
```

At times you must know how long your list is. The answer is easy to get [15], using any of the following two statements.

[15]
```
set theListLength to the length of {"first", "last"}
set theListLength to the count of {"first", "last"}
```



Finally, you can refer to a random item [16].

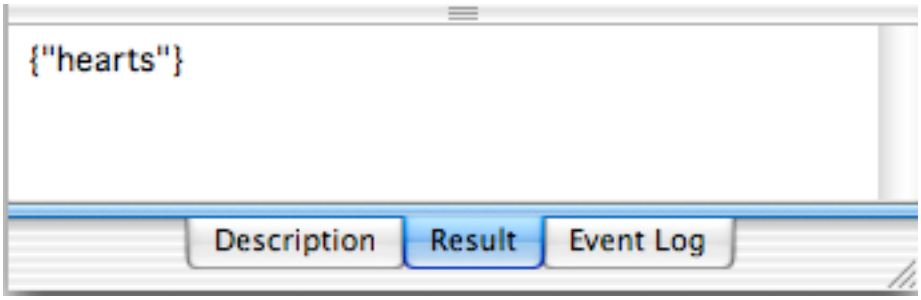
[16]
```
set x to some item of {"hearts", "clubs", "spades", "diamonds"}
```

I know, this is a long chapter, but we really need to discuss some issues having to do with both lists and strings. We couldn't discuss these in Chapter 5, where we discussed strings, because we had not discussed lists yet. Hold on or take a short break first.

In previous chapters you have learned that it is possible to coerce one data type into another. Now we will show you how to turn a string (or a number) into a list [17].

[17]
```
set cardType to "hearts"
set stringAsList to cardType as list
```

After coercing a string into a list, the result is a list containing a single item (a string):



When dealing with both lists and strings, coercion is an important safety measure, as will be demonstrated here.

As you will recall, an ampersand is used to concatenate strings. What happens if you use the ampersand to add a string to a list [18]?

[18]
```
set myList to {"a"}
set myString to "b"
set theResult to myList & myString
```

The data type of the variable theResult depends on the data type which occurs first in the expression to be evaluated [18.3]. Because the expression starts with the variable myList which is a list, the result is a list. Try this for yourself and check the result field. It reveals the data type by displaying the curly braces. If we had reversed the order of the variable names myList and myString, and written [19.3]:

[19]
```
set myList to {"a"}
```

```
set myString to "b"
set theResult to myString & myList
```

the result would have been a string. It goes without saying that, if you don't pay close attention, this behavior can easily lead to a bug in your script. To prevent anything unexpected, perform a coercion [20].

[20]
```
set myList to {"a"}
set myString to "b"
set theResult to (myString as list) & myList
```

In statement [20.3], the string "b" referred to by the variable 'myString' is coerced into a list {'b"}, irrespective of the data type 'myString' referred to. Because the variable 'myString ' refers to a list now, theResult will be a list too [20.3].

To add one or more items to a list, and instead of concatenation, you may write:

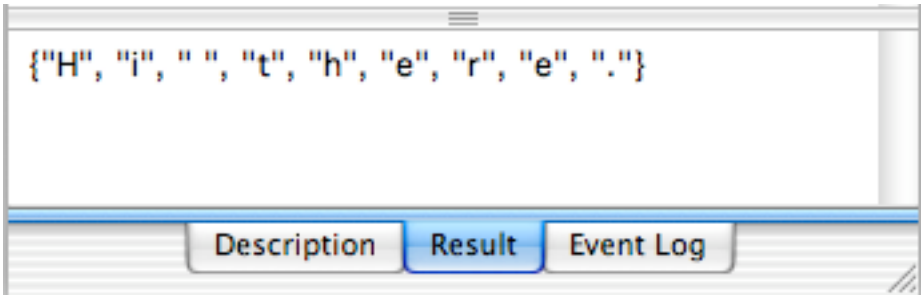
```
set mylist to {1, 2, 3, 4}
set the end of mylist to 5
get mylist
```

This is believed to be faster.
#

Apart from coercion to turn a string into a list, is also possible to create a list containing every character of a string [21]. (Note: While using 'item' instead of 'character' works, it is advised against under Mac OS X).

[21]
```
set itemized to every character of "Hi there."
```

The resulting list, visible in the result field, looks like this:



Instead of itemizing as single characters, you may rather want to cut a sentence up into words. This can be done using AppleScript's text item delimiters. You specify a character that should serve as a separator for defining the items that should eventually constitute the list. To cut up a sentence into words, the separator to use will be a space. Check out script [22]. Good scripting style requires that if you change the AppleScript text item delimiters [22.3], you change it back once you're done [22.5].

[22]
```
set myString to "Hi there."
set oldDelimiters to AppleScript's text item delimiters
set AppleScript's text item delimiters to " "
```

```
set myList to every text item of myString
set AppleScript's text item delimiters to oldDelimiters
get myList
```

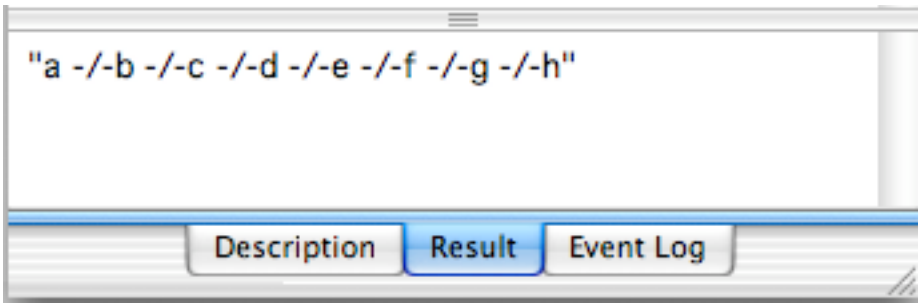
Please take a very good look at statement [22.4], where it says 'text item', not just 'item'. This is an often made mistake. text item text item text item. Got it?

It is easy to coerce a list into a string [23].

[23]
```
set myList to {"a", "b", "c", "d", "e", "f", "g", "h"}
set myList to myList as string
```

If you need a particular character or series of characters in the string to separate the original list items, you should set the AppleScript's text item delimiters accordingly [24].

[24]
```
set myList to {"a", "b", "c", "d", "e", "f", "g", "h"}
set oldDelimiters to AppleScript's text item delimiters
set AppleScript's text item delimiters to " -/-"
set myList to myList as string
set AppleScript's text item delimiters to oldDelimiters
get myList
```



Combining the teachings of script [22] and script [24], you now have the possibility to perform find and replace operations on strings.

#The reason why you should restore the previous AppleScript text item delimiters is that other scripters may be less careful. Their script may assume a particular value of the AppleScript's text item delimiters. A change by your script is remembered by the Mac OS X component of AppleScript even after your script has finished. #

Let us recapitulate: There are several data types, such as number, string and list. Each data type has its own operators you can use to perform operations on your data. A lot of operators are available to manipulate lists. In this chapter on lists we have also learned a thing or two on strings, which we could not discuss before.

CHAPTER 7

RECORDS

In the previous chapter, we used the 'display dialog' command as an introduction to the list data type. Now we will use it to introduce another data type.

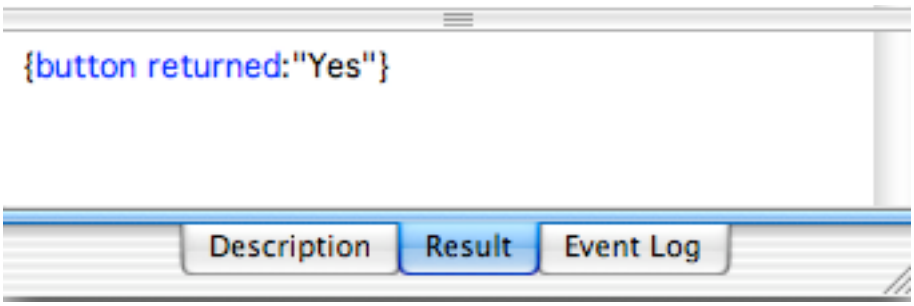
The display dialog command allows you to specify a number of buttons by providing a list of up to three strings [1.2].

[1]
```
set stringToBeDisplayed to "Julia is a pretty actress."
display dialog stringToBeDisplayed buttons {"So, so", "Don't know", "Yes"}
```

But how do we know which button has been pressed? Check out this script [2]

[2]
```
set stringToBeDisplayed to "Julia is a pretty actress."
set tempVar to display dialog stringToBeDisplayed buttons {"So, so", "Don't
                                                          know", "Yes"}
```

If you run this script [2], you can see the result in the result field. Depending on which button you pressed, it will look like this:



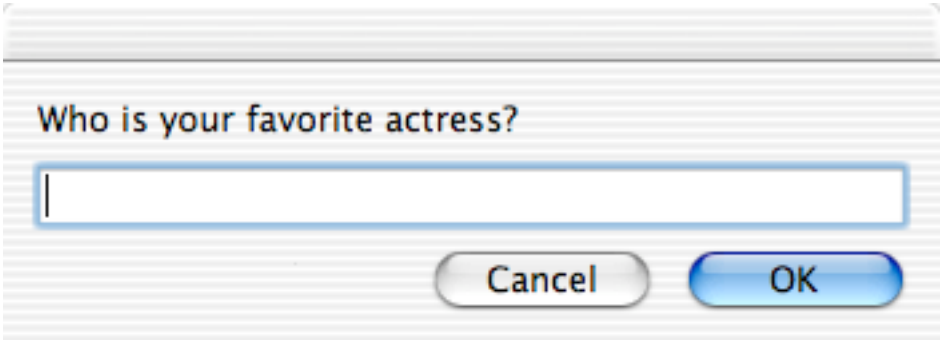
This result represents yet another data type, called a record. Like a list, there are curly braces, but unlike a list every element consists of two parts, separated by a colon. An element of a record is referred to as a property, but also often as a name/value pair. The first part of the property is a label (or name; here 'button returned'), and the second part is the actual value of that property (here the value is the string "Yes"). Because the second part is a value, the Script Editor displays it in black.

To find out which button has been pressed, all we have to do is ask for the value of the property having the label 'button returned' [3.3].

[3]
```
set stringToBeDisplayed to "Julia is a pretty actress."
set tempVar to display dialog stringToBeDisplayed buttons {"So, so", "Don't
                                                          know", "Yes"}
set theButtonPressed to button returned of tempVar
display dialog "You pressed the following button " & theButtonPressed
```

In statement [3.3] we specify that we want to set the variable ‘theButtonPressed’ to the value kept by the property labeled ‘button pressed’ of the record ‘tempVar’. And there we are! We now know which button of the dialog window has been pressed.

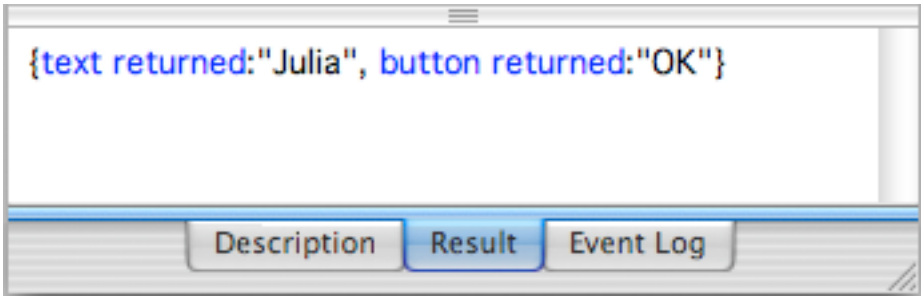
The dialog window is limited in that it can show numbers and (short) strings only. It can not display lists and records. In contrast, the result field can show all the data types we have seen so far. However, the display dialog can do something very useful: It may allow a user to enter numbers or text, which your script can process.



To make the entry box appear, you have to provide a default answer as a string, for example an empty string "" [4].

[4]
```
set temp to display dialog "Who is your favorite actress?" default answer ""
```

If you run the script [4], the result is a record with two properties, i.e. two name/value pairs, which record may look like this in the result field (depending on what has been entered by the user of the script):



Note that, like the items of a list, the properties are separated by a comma. AppleScript doesn’t mistake a record for a list, because of the colons. In contrast to a list, where you may need to remember which item number contains which piece of information, the fact that data can be extracted from a record by referring to the property’s label makes life easy. To extract the name of the actress, all we have to do is ask for the value of the property labeled ‘text returned’ [5.2]

[5]
```
set temp to display dialog "Who is your favorite actress?" default answer ""
set textEntered to text returned of temp

#Please note that the value of text returned is a string, that is, a chunk of text, even if the user entered a number. For example, if the user entered 30, value of the variable textEntered is not 30 but “30”. If you want to perform a calculation with the data entered, you are in luck. AppleScript will try to coerce the string into a number automatically [6.3]. So, you don’t have to include a coercion statement [7.1] after [6.2].
```

[6]
```
set temp to display dialog "What is your age?" default answer ""
set ageEntered to text returned of temp
set ageInMonths to ageEntered * 12
display dialog "Your age in months is" & ageInMonths
```

[7]
```
set ageEntered to ageEntered as number
```

Coercion works if the user entered a number, such as 30. But if the user entered 'thirty' or '30 years', AppleScript can no longer perform the coercion, and the script will fail. Because you can't trust a user to do what you had in mind, you will have to learn to write scripts that take various user behavior into account. In Chapter 10 you will learn how to deal with this issue.#

You can create your own records by setting a variable to a name/value pair [8].

[8]
```
set personalData to {age:30}
```

Note that the color of the property 'age' is green, i.e. defined by you. As usual, the data is shown in black.

#With reference to scripts [3, 5], please note that the property label's 'button returned' and 'text returned' are AppleScript specials. You can see that by their blue color, but what sets them apart is that they consist of two words. If you create your own records, you are not allowed to use two or more words to identify the property [9.1]. The label (or name) of the property must be a single word. [9.2]

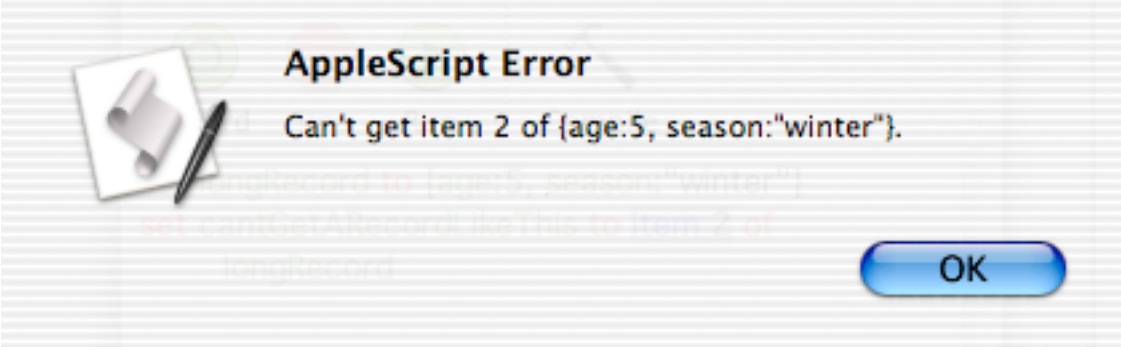
[9]
```
set improperlyNamedProperty to {my property: "This is not correct"}
set properlyNamedProperty to {myproperty:"This is correct"}
```

#

You may concatenate records like lists, but beware: If the records you concatenate have similar property labels, the result can be different from what you expect.

Please do not call the properties of a record 'items', because even though the following script [10] passes the syntax check, AppleScript does not allow you to refer to each record name/value pair as an item [10.2].

[10]
```
set longRecord to {age:5, season:"winter"}
set cantGetARecordLikeThis to item 2 of longRecord
```



You may count how many properties there are in a record though [11]:

[11]
```
set longRecord to {age:5, season:"winter"}
set theNoOfProperties to the count of longRecord
```

To create a new record containing the property of another record, you have to create the record as follows [12].

[12]
```
set longRecord to {age:5, season:"winter"}
set temp to the age of longRecord
set newRecord to {age:temp}
```

The result field shows the new record as being {age:5}. It is possible to write script [12] more succinctly, but you may find it harder to read at first [13].

[13]
```
set longRecord to {age:5, season:"winter"}
set newRecord to {age:age of longRecord}
```

Unfortunately it is not possible to determine which name/'value pairs are present in a record. That is, you can not create a list of all property labels. Similarly, it is not possible to change the labels in a record. If you want to use another label, you should create the record anew, as shown in script [13].

#Let's finish this chapter with one nasty pitfall. Here is a script which should not give you any surprises [14].

[14]
```
set firstValue to 30
set rememberFirstValue to firstValue -- A copy is made and stored by
                                                    'rememberFirstValue'.
set firstValue to 73  --Change the value of original variable.
get rememberFirstValue -- We ask for the value of 'rememberFirstValue'.
```

The result is 30. For records (and lists!) this behavior is completely different, which may result in very hard to trace bugs. Check out script [15]

[15]
```
set personalData to {age:30}
set rememberPersonalData to personalData
set age of personalData to 73
get rememberPersonalData
```

The result is {age:73} ! ! ! The set command does not make a copy if the variable contains a record or a list. To make sure the data is copied, you must use the copy command [16].

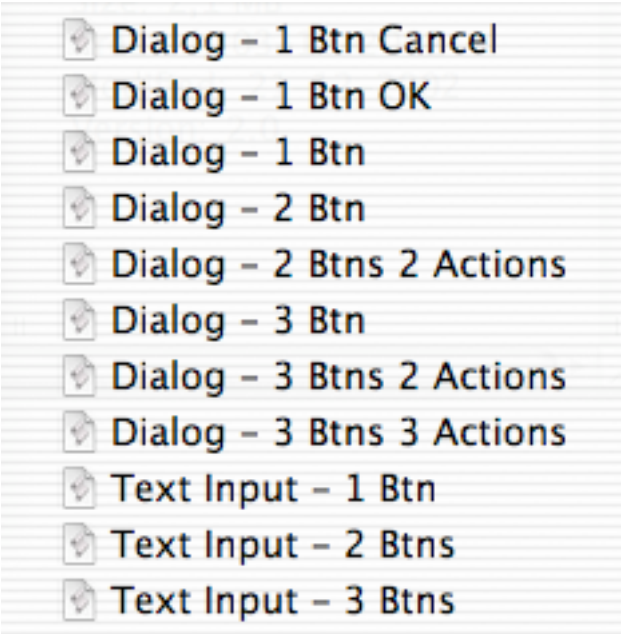
[16]
```
set personalData to {age:30}
copy personalData to rememberPersonalData
set age of personalData to 73
get rememberPersonalData
```

#

CHAPTER 8

EASIER SCRIPTING (II)

In the previous chapter we have seen various ways to display a dialog window. If you feel like remembering all the options, be my guest. If you prefer the easy way, just remember that control-clicking in the upper field of the Script Editor brings up a menu. One of the menu items reads 'Dialogs'. Click it and you are presented with the following submenu.



Here, 'Btn' stands for button. The number preceding it represents the number of buttons. For now, forget about the menu items containing the word 'Actions'. You will understand those by the time we reach Chapter 10.

The last three menu items in the above submenu allow the user to input text. In the Script Editor, type the following:

```
set temp to
```

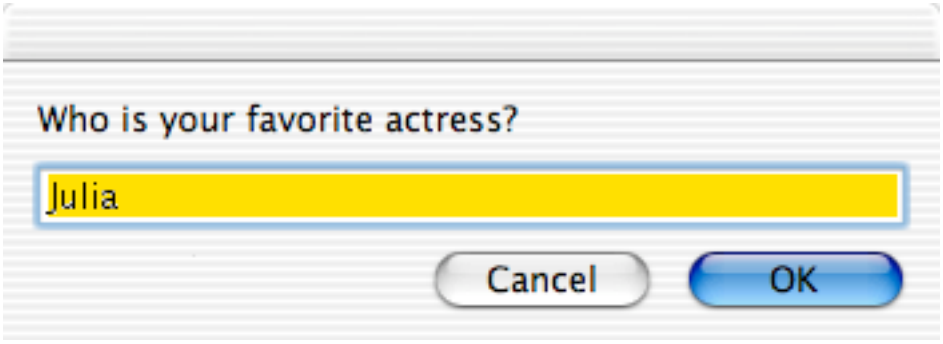
Make sure there is a space after 'to'. Now, control-click right after the space, and select 'Text Input - 2 Btns'. This creates the statement needed to provide a user with a field where he or she can enter data, as discussed in Chapter 7 when we discussed the property 'text returned'.

If you want to, display dialog can provide a default answer, and the user of the script can replace it with something else if necessary. If a particular answer is likely, it makes your script more user-friendly if you provide that answer as the default answer [1.1]. And you know it, Macintosh users do love user-friendliness. Even if the default answer is not likely to be the answer, it still gives the user a clue about the type of answer expected.

[1]

```
set temp to display dialog "Who is your favorite actress?" default answer
                                                                    "Julia"
```

If you run script [1], you are presented with the following dialog. If you agree, you just hit Enter. Otherwise, you can start typing another name.



In short, there is no need to remember the exact commands for all incarnations of display dialog. The Script Editor does it for you, and you can add them to your script without typing.

CHAPTER 9

NO COMMENT? UNACCEPTABLE!

Several factors make AppleScripts easy to read, write and maintain. Some are beyond your control, such as the English-like nature of this scripting language. However, other factors are your department, such as the use of descriptive variable names. In this chapter we will discuss another important factor.

While we currently stuck to examples only a couple of lines long, the AppleScripts you can write will get longer and longer. Of prime importance when writing scripts is not just worrying that your script actually does what you want it to do, but also that it is properly documented. Later, if you haven't seen the script for a while and want to change it, you really need comments to help you understand what a particular part of a script does and why that part is there in the first place. You are strongly advised to take some time commenting your script. We can assure you that you will gain back the time spent manifold in the future. Also, if you share your script with someone else, that person will be able to customize it quickly if you have provided comments.

To create a comment, start the comment with two hyphens.

```
-- This is a comment
```

After compilation (syntax checking), the comment is shown in gray.

```
-- This is a comment
-- This is a comment spanning more than
      one line
```

In the old days, multi-line comments were put between (* *)

```
(* This is a comment
extending over two lines.*)
```

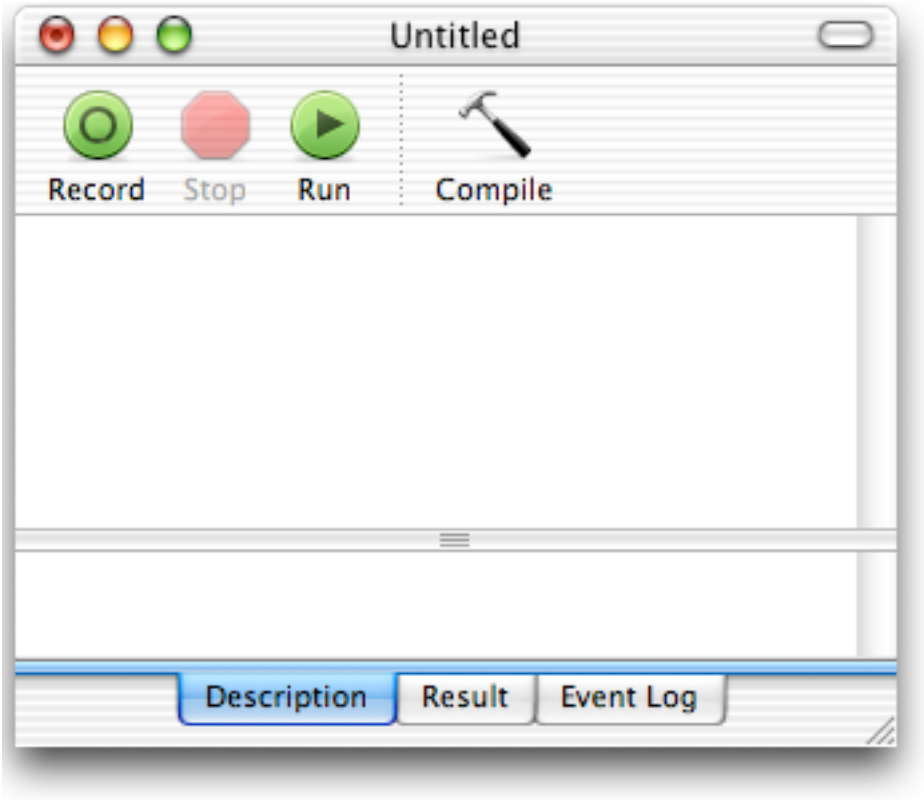
With the advent of the contextual menu, that is not recommended anymore. You should stick with the two hyphens. That is because of a practice called outcommenting.

By placing part of your script between (* *), you can temporarily disable ('outcomment') part of the script, to see if the rest works as expected. This allows you to hunt down a bug. If the outcommented part should result in, for example, a value for a particular variable, you can include a temporary line where you set the variable to a value suitable for testing the remainder of your script.

The contextual menu available in the Script Editor allows for easy outcommenting. If you want to outcomment a part of your script, selected that part, for example by dragging your mouse. Then control-click the upper field and select 'Comment Tags'. You are offered to add or remove comment tags in the selected text. Click 'Add' and you are done. To remove the

comment tags, you make sure the tags are within the selected part of the script. It is not necessarily to do this with pinpoint precision. Then select ‘Remove”. If your explanations were placed after two hyphens, they survive this procedure.

In the Script Editor, click the Description tab (if it isn’t already highlighted). In the lower field you can put a general explanation of the purpose of your script. In addition, or alternatively, you should start the upper part of your script with an explanation (using the two hyphens).



The importance of comments can not be overstated. The scripts in this book do not contain as many comments as we would ordinarily have written in, because they are already surrounded by explanations.

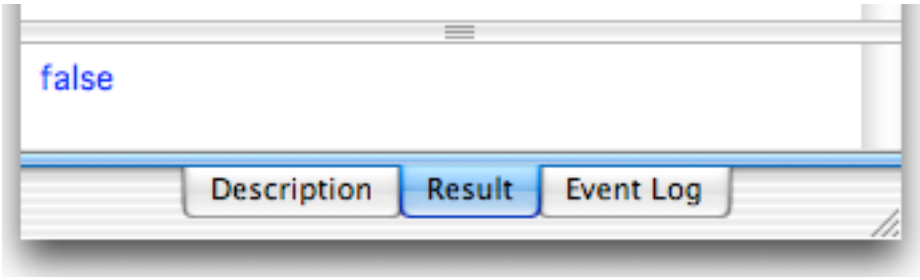
CHAPTER 10

CONDITIONAL STATEMENTS

At times, you will want your script to perform a series of actions only if a particular condition is met. Type the following line [1] in the Script Editor and click Run.

[1]
```
73 = 30
```

This is what you will see in the Result field.



AppleScript evaluates the comparison in script [1] and concludes it is false , i.e. not true. If you enter something like '30=30', you will see that the result is 'true'.

It is this capability of AppleScript to compare two values that is used in the 'if...then' statement [2] to execute statements only if the condition is met. The 'if...then' statement is called a conditional statement. Please note that this statement requires an 'end if' statement. More on that later.

[2]
```
if true then
	-- actions performed
end if

if false then
	-- these actions are not performed
end if
```

So, what we need to do is to replace the word 'true' in statement [2.1] with a comparison. If that comparison yields 'true', the statements of line [2.2] are performed.

[3]
```
set ageEntered to 73
set myAge to 30
if ageEntered is myAge then
	display dialog "You are as old as I am."
end if
```

If the comparison ‘ageEntered is myAge’ [3.3] would have been true, the dialog window would have been displayed. With the above values of the variables in the first two statements, you will not see the dialog window [3.4].

If there are more instructions that have to be executed if the condition is met, they all have to be included within the ‘if...then...end if’ block statement [4].

[4]
```
set ageEntered to 73
set myAge to 30
if ageEntered is myAge then
	display dialog "You are as old as I am."
	beep
end if
say "This sentence is spoken anyway."
```

You will recognize the similarity between a tell block and an ‘if.. then’ statement. In both cases a statement containing the word ‘end’ is required. ‘end if’ allows AppleScript to identify which are the statements that have to be executed if the comparison yields ‘true’. In script [4], you will hear the last statement [4.7] irrespective whether the condition [4.3] is met or not.

We may also provide an alternative set of instructions if the condition is not met, using an if...then...else statement [5]

[5]
```
set ageEntered to 73
set myAge to 30
if ageEntered = myAge then
	display dialog "You are as old as I am."
else
	display dialog "You are not as old as I am." -- [5.6]
end if
```

Here, the dialog of statement [5.6] will be displayed, because the comparison in statement [5.3] yields ‘false’. There are many more comparisons you can perform. Here are the comparison operators for the various data types.

Apart from the equal sign in statement [5.3], the following comparison operators for numbers are at your disposal.

For numbers

=	is (or, is equal to)
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

The second column does not merely show the explanations of the symbols in the first column, but you can use them verbally as well! [6]

[6]
```
if a is greater than b then
	display dialog "a is larger"
end if
```

If you type >= and check the syntax, AppleScript automatically converts this comparison operator into the official symbol ≥. Similarly, the 'smaller than or equal to' comparison operator <= is replaced with ≤. You do have to type the symbols in the right order, or AppleScript will complain. So, it isn't always as user-friendly as it could be.

Negative formulations are possible as well, for example, 'is not greater than'. If you type '/=', it is automatically reformatted at compilation as ≠, which is short for 'is not'.

In Chapter 7 we came across the following script [7], which allowed us to determine which button has been pressed.

[7]
```
set stringToBeDisplayed to "Julia is a pretty actress."
set tempVar to display dialog stringToBeDisplayed buttons {"So, so", "Yes"}
set theButtonPressed to button returned of tempVar
```

Using the 'if...then' statement, we can perform a desired action [8].

[8]
```
set stringToBeDisplayed to "Julia is a pretty actress."
set tempVar to display dialog stringToBeDisplayed buttons {"So, so", "Yes"}
set theButtonPressed to button returned of tempVar
if theButtonPressed is "Yes" then
	say "I agree entirely!"
else
	say "Didn't you see the movie 'Pretty Woman'?"
end if
beep
```

If the user presses the button "Yes", then the sentence of statement [8.5] is spoken. In any case, the script continues with the beep in the last statement.

In the previous script, the values of the strings had to be identical. However, AppleScript can do much more than that. Here are four examples [9]. You will notice that 'end if' is not required if the instruction to be executed is on the same line as the 'if...then' statement.

[9]
```
set textString to "Julia is a beautiful actress."
if textString begins with "Julia" then display dialog "The first word is Julia"
if textString does not start with "Julia" then beep
if textString contains "beau" then set myVar to 5
```

Below an overview of the comparison operators for strings.

<u>For strings</u>
begins with (or, starts with)
ends with
is equal to
comes before
comes after
is in
contains

You may phrase things negatively too:

does not start with
does not contain
is not in
etc.

If you write ‘doesn’t’, it is automatically reformatted to ‘does not’.
If you write ‘does not begin with’, it is automatically reformatted to ‘does not start with’.

The ‘comes before’ and ‘comes after’ comparison operators work alphabetically. So, the following statement will result in a beep [10].

[10]
```
if "Steve" comes after "Jobs" then
	beep
end if
```

#When comparing strings, capitalization may be important to you. Just signal it to AppleScript. Of course, you will have to turn it off when you’re done [11].

[11]
```
set string1 to "j"
set string2 to "Steve Jobs"
considering case
	if string1 is in string2 then
		display dialog "String2 contains a \"j\""
	else
		display dialog "String2 does not contain a \"j\""
	end if
end considering
```

By default, white space (space, return, tab) are taken into account. If you don’t want that, you type ‘ignoring white space’, as demonstrated by script [12]. Please note that you have to include an ‘end ignoring’ or ‘end considering’ statement.

[12]
```
set string1 to "Stev e Jobs"
set string2 to "Steve Jobs"
ignoring white space
	if string1 = string2 then beep
end ignoring
```

You may also instruct AppleScript to ignore punctuation, and to ignore diacriticals. #

For the list data type, less comparison operators are available than for strings, but otherwise they are the same. So, you do not have to learn more comparison operators than necessary.

<u>For lists</u>
begins with
ends with
contains
is equal to
is in

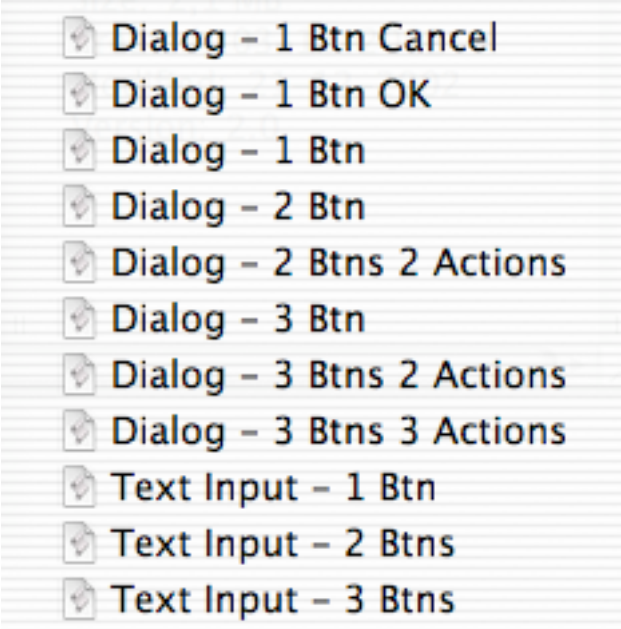
Often, you will compare individual items of a list (or of different lists). Let’s look at a practical

example. Look at script [8]. What if we have three buttons in our dialog? By nesting 'if...then' statements [13], all options are considered.

[13]

```
set stringToBeDisplayed to "Julia is a pretty actress."
set tempVar to display dialog stringToBeDisplayed buttons {"So, so", "Who?",
"Yes"}
set theButtonPressed to button returned of tempVar
if theButtonPressed is "Yes" then
	say "I agree entirely!"
	beep
else
	if theButtonPressed is "Who?" then
		say "Didn't you see the movie 'Pretty Woman'?" -- [4.8]
	else
		say "I don't agree with you."
	end if
end if
```

#As you can see, the indentation facilitates reading the script, but it still is a bit awkward to read. Plus, it is a lot of typing and you may easily forget or misplace an 'end if' statement, as a result of which your script won't compile. Script Editor's contextual menu's to the rescue! Do you recall the Dialog menu items where it read Actions?



Click the one for 3 buttons, 3 Actions and you are presented with the full set of statements needed. You just need to fill out the values of the list [14.1] and type out the actions needed.

[14]

```
display dialog "" buttons {"", "", ""} default button 3
set the button_pressed to the button returned of the result
if the button_pressed is "" then
	-- action for 1st button goes here
```

```
else if the button_pressed is "" then
	-- action for 2nd button goes here
else
	-- action for 3rd button goes here
end if
```

The first two lines may need an explanation. As you can see, statement [15.1] differs from statement [8.2] in that we don't set a variable to the record resulting from the display dialog command. Just like the result field automatically displays the result of the last statement executed, the result of a previous statement is available to you if you refer to it using the keyword 'result'. By putting the if statement on the same line as 'else' [14.5], a separate 'end if' is not required. All in all, this script is easier to read. #

For records the number of comparison operators is even more limited than for lists.

For records
contains
is equal to -- or =

[15]
```
set x to {name:"Julia", occupation:"actress"}
if x contains {name:"Julia"} then display dialog "Yes"
```

#Did you notice that in statement [15.1] one label is displayed in blue, and the other one in green? You are warned that one of the labels is a keyword. While AppleScript does not stop you from using property labels identical to reserved keywords, such as 'to', 'set', string etc., this may get you into trouble at times, so do avoid reserved keywords using the tips in Chapter 4 for variable names.#

The limited number of comparison operators for records is not a problem because usually you will not compare full records but the individual values of properties of the record [16].

[16]
```
set aRecord to {name:"Julia", occupation:"actress"}
if name of aRecord is "Julia" then display dialog "OK"
```

As we have seen in the first part of this chapter, if you compare two values (whatever their data type), you are actually trying to see if a comparison expression is true or false. Indeed, they represent a special data type called Boolean. Variables holding a value of this type can have only one of two values: true and false. For numbers, you have operators like '+' and '-'. If you use such an operator, the result is a number. For Boolean operators, we have 'and', 'or' and 'not'. The result of such an operation is again Boolean.

'not' is the simplest of the three. 'not true' is false, and 'not false' is true.

[17]
```
set x to not true -- So, x is false
```

As demonstrated in the following script [18], for the 'and' operator, both x and y have to be true for z to be true.

[18]
```
set x to true
set y to true
set z to (x and y) -- z is true
```

In contrast, for the ‘or’ operator, it suffices if one of x and y is true [19].

[19]
```
set x to true
set y to false
set z to (x or y) -- z is true
```

Why did I tell you all this? Well, in some circumstances you want a set of instructions to be executed only if multiple conditions are met. The following script [20] will display the dialog of the last statement only.

[20]
```
set x to 5
set y to 7
set z to "Julia"
if x = 5 and y = 6 then display dialog "Both conditions met."
if x = 5 or z = "actress" then display dialog "At least one condition met."
```

In statement [20.4] the first comparison is true, but the second is not. ‘and’ requires both to be true, so the dialog is not displayed.
In statement [20.5], ‘or’ is already happy if one of the conditions is met.

In script statements like [20.4, 20.5] you will want to use parentheses, as this enhances readability (and possibly reliability).

CHAPTER 11

TRYING WITHOUT FAILING

In all the scripts we have discussed so far, if AppleScript encounters a problem during execution, the script is terminated.

[1]
```
beep
set x to 1 / 0
say "You will never hear this!"
```

If you check the syntax of the above script [1], AppleScript will not report any problems. However, if you run the script, you will not hear the sentence [1.3] being spoken, because all further execution of the script, however valid statement [1.3], is stopped at statement [1.2].

Of course, termination of a script is not necessarily what you want. For example, if your script requires the presence of a folder with files to process, and the folder has been deleted, you may want to give the user the option to select another suitable folder.

When writing a script, you have to identify statements that are susceptible to problems during execution. Enclose these statements between a try ... end try block statement as demonstrated here [2].

[2]
```
try
	beep
	set x to 1 / 0
	say "You will never hear this!"
end try
say "The error does not stop this sentence being spoken"
```

Now, if you run the script, you will hear the second sentence [2.6], as AppleScript will continue after the ‘end try’ statement [2.4].

In Chapter 7, which dealt with records, we encountered the following script [3]

[3]
```
set temp to display dialog "What is your age in years?" default answer ""
set ageEntered to text returned of temp
set ageInMonths to ageEntered * 12
display dialog "Your age in months is " & ageInMonths
```

The problem with this script is, that if a person enters something else than a number, the script fails. We can avoid abortion of the script, and provide the user with useful feedback [4].

[4]

```
set temp to display dialog "What is your age in years?" default answer ""
set ageEntered to text returned of temp
try
	-- First we check if the user entered a number
	set ageEntered to ageEntered as number
	set ageInMonths to ageEntered * 12
	display dialog "Your age in months is " & ageInMonths
on error
	-- If it is not a number, the entry must have been text."
	display dialog "Instead of a number, like 30 , you entered text."
end try
```

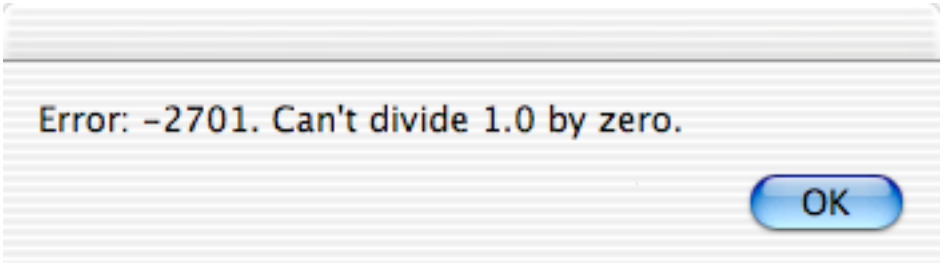
Now, if the user does not enter a number, he is provided with feedback. An inconvenience is that the user has to start the script again. In Chapter 13 we will solve that problem.

#The Script Editor makes it very easy for you to include try blocks. Just select the statement or statements you want to include within the try block, and use the contextual menu to select the try block of your choice, such as the one used in script [17].

When a script is diverted to the 'on error' part of a try block, AppleScript has a couple of goodies for you: the number of the error, as well as an (often cryptic) explanation of the problem.

[17]
```
try
	set x to 1 / 0
on error the error_message number the error_number
	display dialog "Error: " & the error_number & ". " & the error_message
buttons {"OK"} default button 1
end try
```

If you put a variable name after 'on error', the explanation of the problem with be put inside that variable. If you have a variable name preceded by 'number', the number of the error will be put into that variable. In statement [17].3 we have both. Here is what the dialog looks like.

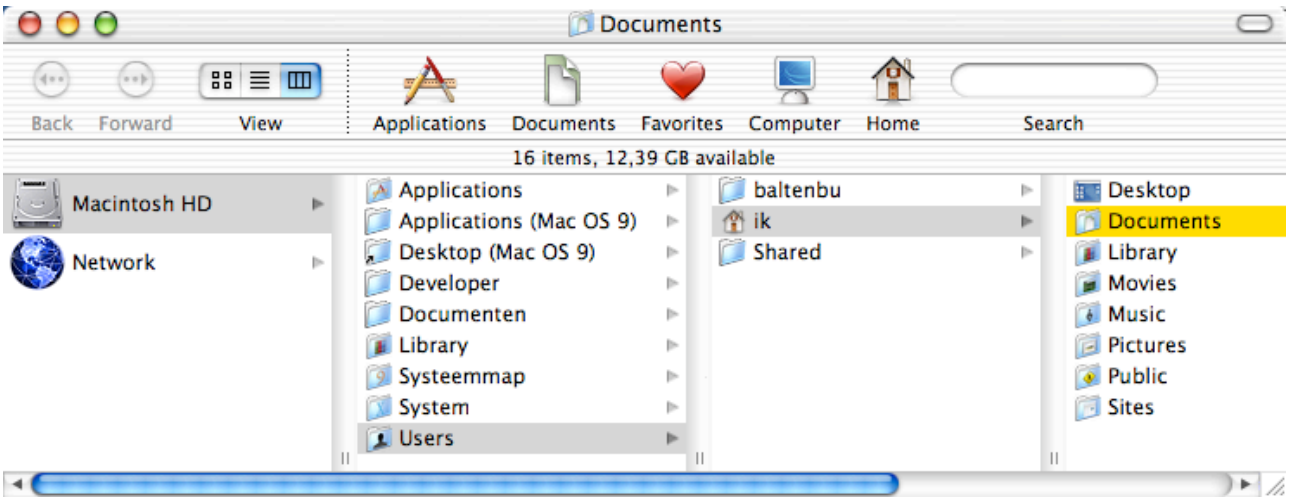


#

CHAPTER 12

PATHS TO FILES, FOLDERS AND APPLICATIONS

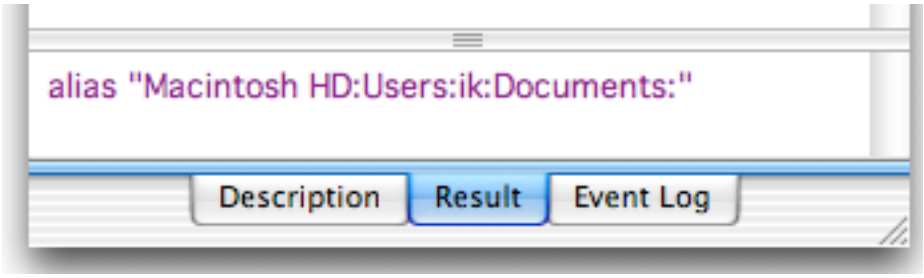
Let's take a look at how folders and files are organized on your hard disk. If you open a Finder window in column view, it looks like this.



There is a hard disk, which contains folders, applications and files (the above pictures does not show files and applications). All these elements are organized in a hierarchical manner. This allows us to define the location of a file (or folder or application) by means of a path. Let's see what such a path looks like [1].

[1]
```
choose folder
```

Here is what the result field shows if I select my Documents folder.



You can see a path that basically looks like this:

hard disk name:folder name:subfolder name:subfolder name:

Trace this path in the top picture (screenshot of Finder window) to the folder ‘Documents’. Then shift your attention to the picture with the Result field. Please note that in the path colons are used as separators. That is why colons are not allowed in file or folder names. Try that by

creating a new folder on your desktop and giving it a name containing a colon. Finally, you can see that the path ends with a colon, which indicates that the path refers to a folder.

We can use the path to tell the Finder to open the folder [2].

[2]
```
tell application "Finder"
	open folder "Macintosh HD:users:ik:Documents"
end tell
```

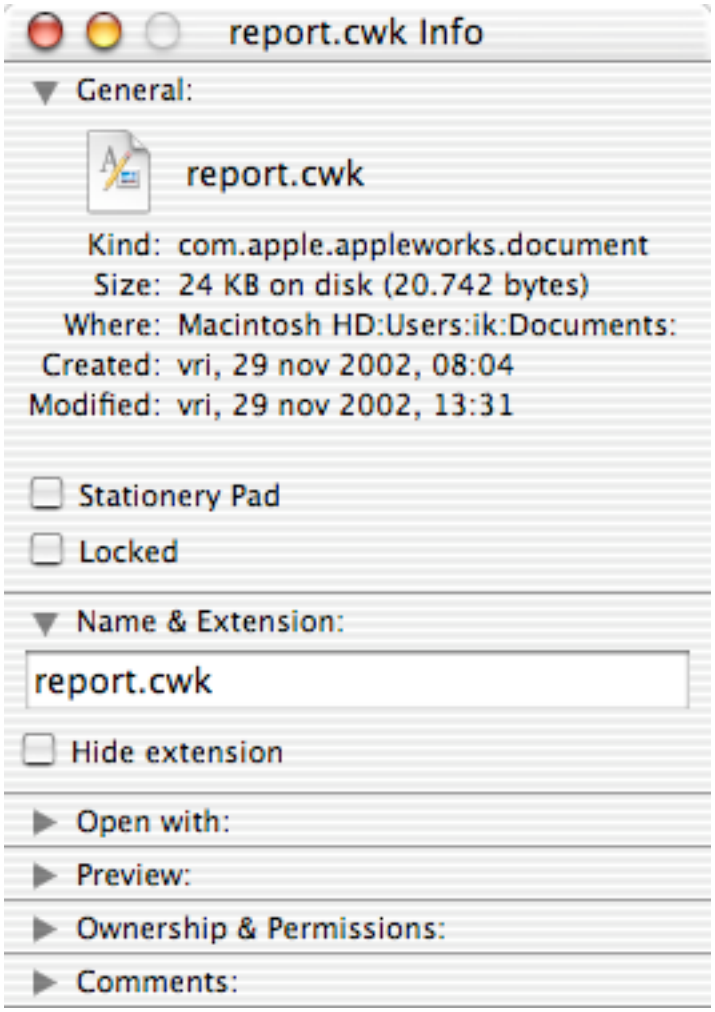
As you can see when you run script [2] (don't forget to replace 'ik' for your own login name), AppleScript is quite forgiving and does not care if you forget the colon at the end of the path. However, please note that the name of the hard disk must be properly capitalized. Tsk, tsk, Apple!

My folder 'Documents' contains an AppleWorks file named "report". Let us open that file [3].

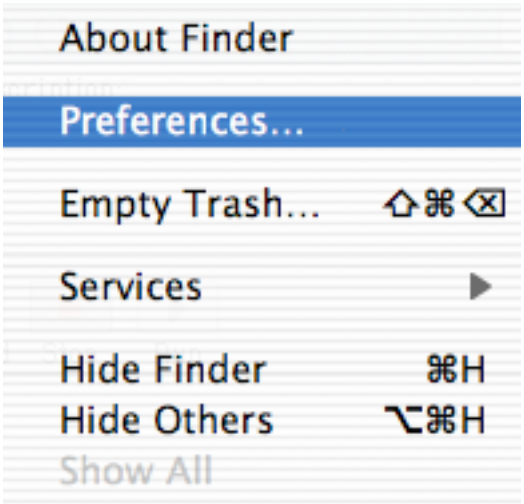
[3]
```
tell application "Finder"
	open file "Macintosh HD:users:ik:Documents:report.cwk"
end tell
```

The first thing to note is that statement [3.2] specifies 'file', not 'folder', because we are referring to a file. A second thing to note is that the file name at the end of the path must specify the extension. Here the extension is 'cwk', derived from the original name of AppleWorks, which was ClarisWorks.

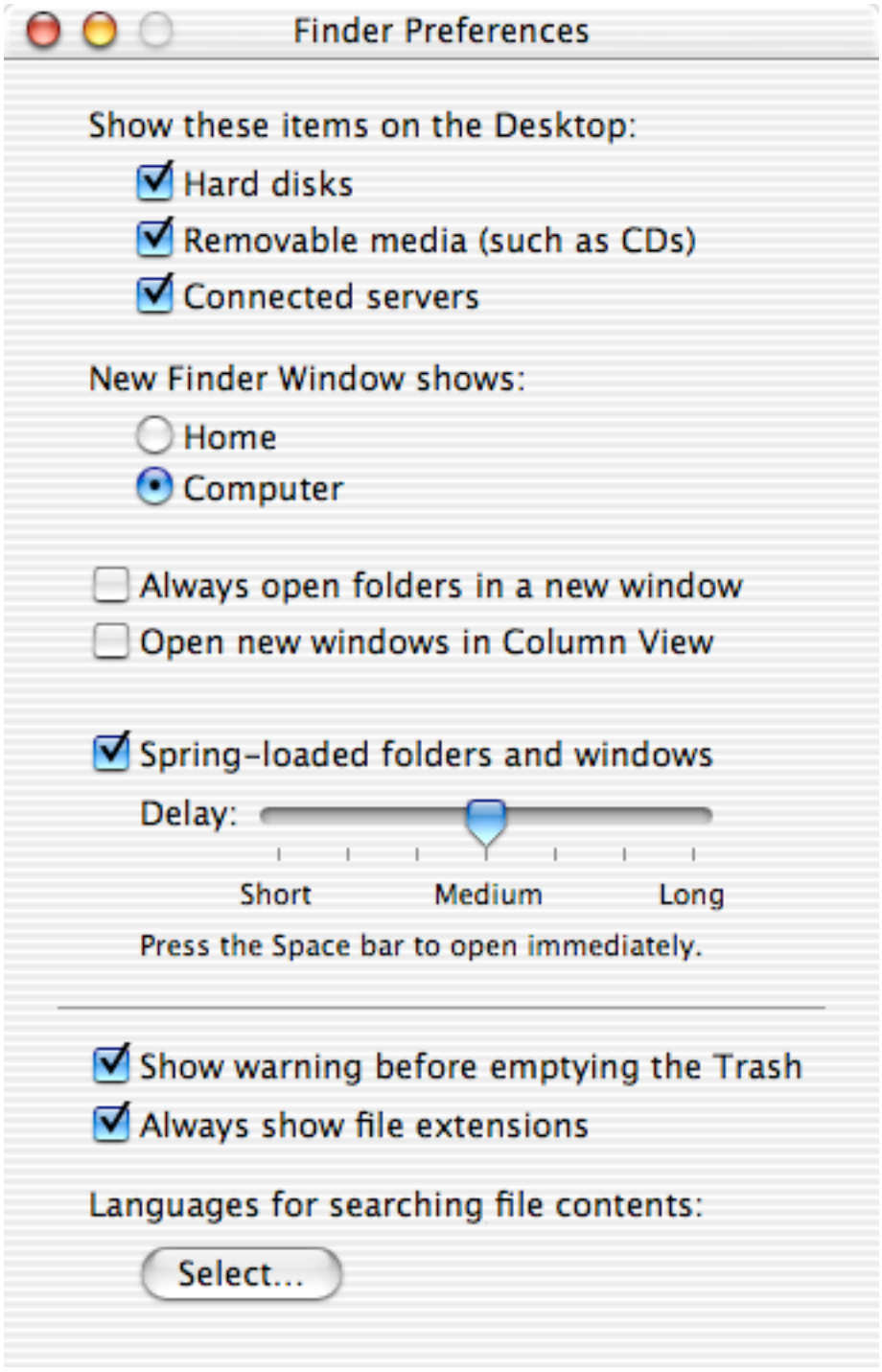
#In Mac OS X the extension of a file is hidden by default. You can find out the extension by selecting a file and pressing command-i, to get info on that file.



Instead, you may opt to make file extensions visible always. You can do this by setting the Finder preferences accordingly. Click the ‘Finder’ item of the menubar of the ‘Finder’. A menu appears. Select ‘Preferences’.



In the window that appears, set the tick near the bottom.



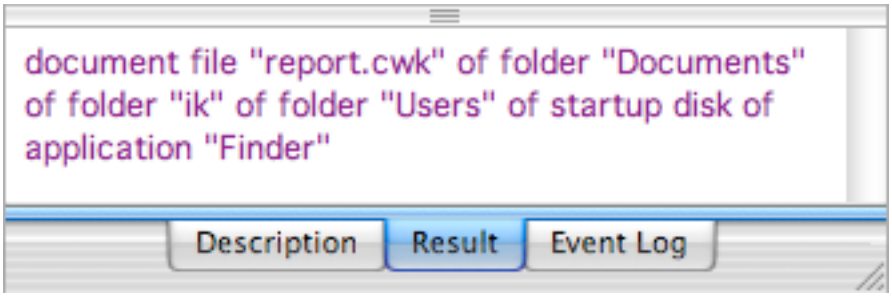
#

If you want to store the path to the file 'report.cwk' in a variable, you may be inclined to do that like this [4]

[4]

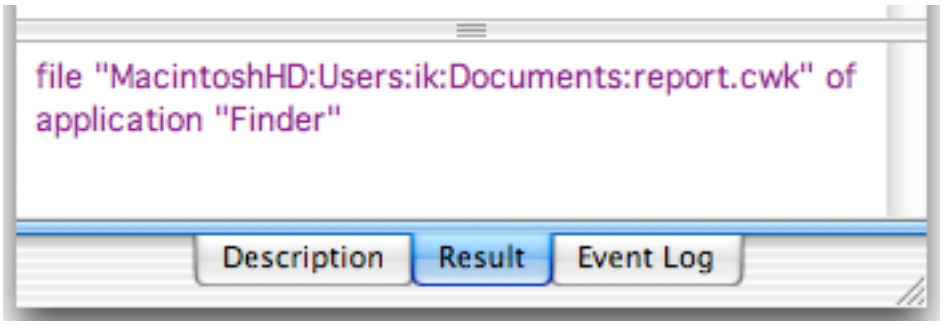
```
tell application "Finder"
	set thePath to file "Macintosh HD:Users:ik:Documents:report.cwk"
end tell
```

However, if you run this script, this does not result in the path format we've seen above. It now looks like this:



This style is a bit awkward to read, especially for longer paths, and worst of all, it is in a format is recognized by the Finder only. You may well prefer, or usually need, the more abstract notation of a path using colons. You can force the Finder to return that, by using the 'a reference to' command [5].

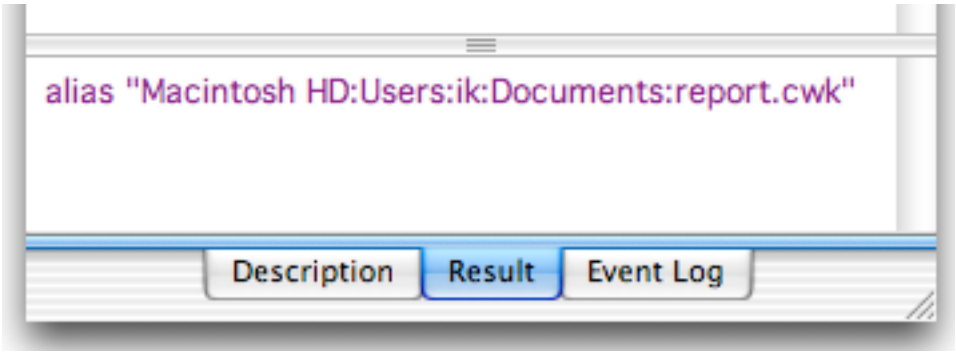
[5]
```
tell application "Finder"
	set thePath to a reference to file "Macintosh HD:Users:
		ik:Documents:report.cwk"
end tell
```



Note that the result field says that we have a file. Please run the script [6] below and select the same file report.cwk.

[6]
```
choose file
```

The result field now looks like this.



It reads 'alias', instead of 'file'! To explain the difference to AppleScript, let us first discuss the alias as you will be familiar with from working in the Finder.

Suppose I have created an alias on my desktop of the file 'report.cwk' which file is in my folder 'Documents'. Now, if I were to move the file 'report.cwk' from the folder 'Documents' to another location, double-clicking the alias would still open the file. Great! I may even rename the original file 'report.cwk' to something else, such as 'funny_story.cwk'. That is because the alias does not store the location (and name) of the file 'report.cwk' as

"Macintosh HD:users:Documents:report.cwk"

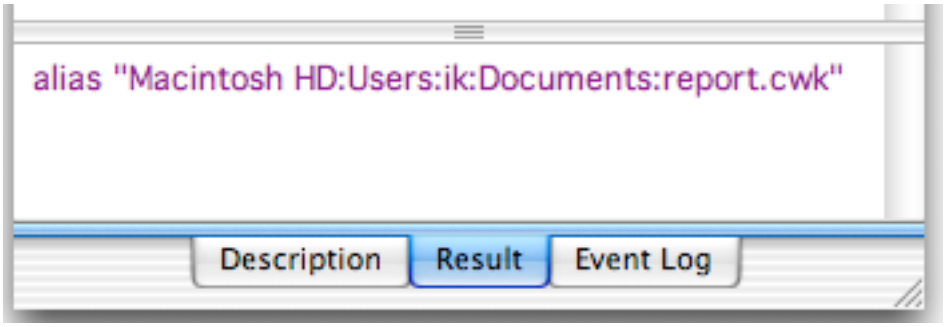
but rather as a kind of ID. The Finder maintains a database of these IDs together with the current whereabouts of the corresponding files (and folders and applications, for that matter). So, if I move the file 'report.cwk', its unique ID remains the same, but the Finder changes its internal database to reflect the new location of the file. When I double-click the alias, the ID contained by it is used by the Finder to find out which file corresponds to the ID, and the Finder will open the correct file.

To have a script that does not break if the file (or folder) a script refers to is removed or renamed, our script should contain the ID of the file (or folder), instead of a 'hard-coded' path. AppleScript allows for this [7].

[7]
```
set thePath to alias "Macintosh HD:Users:ik:Documents:report.cwk"
```

It is very important to note that the statement of script [7] refers to the original file present in the folder 'Documents', and not some user-created alias of that file, like the one I purported to have on my desktop. In the statement of script [7], 'alias' is a keyword indicating that, after compilation (i.e. checking of the syntax), the script should remember the ID of the file and, upon execution, should not ask the Finder for a file at the location as specified by the defined path, but based on the ID.

If you run the script [7], and check its result in the result field, it looks like this:



So, you do not get to see the ID itself. However, the word 'alias' before the path tells you that internally the script works with the ID, and not with a hard-coded path. Now, with the script being compiled, move the file 'report.cwk' to another location and run the script again. I moved the file into a folder named 'Miscellaneous' inside my folder 'Documents'. Even though the script, and in particular the path in script [7], has not changed, the result now is

```
alias "Macintosh HD:Users:ik:Documents:Miscellaneous:report.cwk"
```

Try this yourself (again, your path looks different because of your login name that is probably not 'ik' and perhaps because you may use different folder names)! If you save the script as a compiled script or an AppleScript application, the ID is stored, and next time you run the script, it will perform flawlessly even if you moved or renamed the original file. It should not surprise you that the script will fail if the file has been deleted.

In summary, you have two ways of dealing with paths in scripts. You can either specify a file

location (hard-coded, using file "*path here*"), or you can use the 'alias' keyword to make the script insensitive to movement or renaming of the file/folder/application after compilation of the script.

While the use of 'alias' is often the way to go, the file you refer to has to be there when you compile the script. Also, you can not give the script to someone else in the form of an AppleScript application, because even if a file with that same name is present at a similar location on that person's computer, the file will have a different ID.

#Just a note about script [7]. Personally, I was quite surprised that this works without a Finder tell block. Afterall, the internal database containing IDs and file locations is maintained by the Finder. Because the Finder is the only program that can supply the information, and the AppleScript component of Mac OS X knows that, AppleScript apparently asks the Finder for the ID behind the scenes or queries the database directly. This behavior is an exception, though. For example, opening a file using a path requires a tell block (See script [3]). That is because it is not just the Finder that could open a particular file, but also the program which created that file, and possibly another program as well. For example, a picture in jpeg format, can be opened by PhotoShop and by your browser. The tell block is needed to indicate which program is to open the file.#

Now we know how to address files and folders, we can move them around or copy them [8].

[8]

```
tell application "Finder"
	move file "Macintosh HD:Users:ik:Documents:report.cwk" to the trash
end tell
```

CHAPTER 13

REPEATING

In all the scripts we have discussed so far, each statement was executed just once. At times, you will need to execute a one or more of statements several times. AppleScript offers several ways to achieve that.

If you know the number of times the statement (or group of statements) has to be repeated, you may specify that by including that number in the repeat statement of script [1]. The number must be an integer, because you can not repeat an operation, say, 2.7 times.

[1]
```
repeat 2 times
	say "Julia is a beautiful actress"
end repeat
say "This sentence is spoken only once"
```

Similar to what we have seen for the 'tell', 'try' and 'if...then' statements, an 'end repeat' statement is compulsory to indicate to AppleScript which statements belong to the group of statements to repeat

Instead of specifying the number, you may use a variable instead [2].

[2]
```
set repetitions to 2
repeat repetitions times
	-- Statements to repeat here
end repeat
```

Here [3] is a more life-like example of script [2]. When script [3] is run, the user of the script is allowed to enter a number in a dialog window. Because anything entered there ends up in 'text returned', we must convert the value entered into an integer. That is not possible if the user entered text or a real (fractional) number. So, we have to take some precautions.

[3]
```
-- The user is allowed to determine how often the text string is spoken
set textToDisplay to "How often has the sentence to be repeated?"
-- The number '2' is displayed to give the user a hint of the type of
	answer expected
display dialog textToDisplay default answer "2"
set valueEntered to text returned of the result
try
	-- valueEntered is a string (not an integer), which may look like this: "2"
	-- Here we try to coerce the value entered by the user into an integer. If
		that doesn't work, the try statement prevents the script
```

```
            from being aborted
        set valueEntered to valueEntered as integer
end try
-- If valueEntered is of the appropriate class (i.e. integer), we can perform
        the repeat block. If not, we provide a dialog.
if class of valueEntered is integer then
        -- The repeat block is executed if the coercion did not fail
        repeat valueEntered times
                say "Julia is a beautiful actress"
        end repeat
else
        display dialog "You did not enter a (valid) number."
end if
```

After the else statement [3.21], the user of the script is reproached without being given a hint how to get it right, and the user has to start the script again. That does not earn you many points in the Macintosh community. There are two alternative repeat statements [4, 5] that allow you to repeat operations until a condition is met.

[4]

```
set conditionMet to false
repeat while conditionMet is false
        -- if (some test is passed) then execute the following statement
        --      set conditionMet to true
end repeat
```

[5]

```
set conditionMet to false
repeat until conditionMet is true
        -- if (some test is passed) then execute the following statement
        --      set conditionMet to true
end repeat
```

Let us use the repeat statement of script [4] to alleviate the inconvenience of script [3]. We will repeat the request until the value entered can be coerced into an integer. In case of a successful coercion, we will set the variable 'correctEntry' to true, as a result of which the repeat loop is exited and the remainder of the script is executed [6]. If the value entered by the user can not be coerced into an integer, we will give detailed feedback to the user.

[6]

```
set correctEntry to false
repeat while correctEntry is false
        -- The user is allowed to determine how often the text string is spoken
        set textToDisplay to "How often has the sentence to be repeated?"
        -- The number '2' is displayed to give the user a hint of the type
        answer expected
        display dialog textToDisplay default answer "2"
        set valueEntered to text returned of the result
        try
```

```
                -- The valueEntered is always string.
                -- Here we try to coerce the value entered by the user
                        into an integer. If that doesn't work, we jump to the
                        on error section
                set valueEntered to valueEntered as integer
                -- Setting correctEntry to true will end the loop
                set correctEntry to true
        on error
                -- We will give detailed feedback.
                try
                        -- First we check if the user entered a fractional number
                        set valueEntered to valueEntered as number
                        display dialog "You entered a fractional number instead
                                                                of an integer."
                on error
                        -- If it is not a number, the entry must have been text."
                        display dialog "Instead of an integer, like 9 , you
                                                                entered text."
                end try
        -- Because the value of correctEntry is still false, the loop continues.
        end try
end repeat

-- The script can make it here only if correctEntry is true
        CorrectEntry is only true if valueEntered is successfully coerced
        into an integer
repeat valueEntered times
        say "Julia is a beautiful actress"
end repeat
```

Please note that the location of the statement ‘set correctEntry to true’ is very important. It must be
- within the (first) try block; and
- after the statement that may result in an error (here the attempt at coercion)
Otherwise, correctEntry would be set to true irrespective of whether the requirement (successful coercion to an integer) is met.

#
While script [3] can perform the desired action (speaking a sentence an indicated number of times) exactly like script [6], it is neither user-friendly nor robust. I.e., it can fail if the user enters the wrong data, and does not provide proper feedback in case the user makes a mistake. However, script [6] could be expanded by putting a restriction on the upper limit of the value of valueEntered (for example using the script fragment [7]), so as to prevent the sentence being spoken 10,000 times. On the other hand, script [6] may already be overkill for your purposes.

[7]
```
if valueEntered > 5 then
        set valueEntered to 5
end if
```

If you really need a robust script, you should test it extensively. Try to enter text, fractional numbers, extreme values etc. to verify that the script is well-behaved. One thing not covered by script 6 is the occasion where the user enters a negative number. You could either turn it into a positive number, or just signal the user that a positive number is expected. Interestingly, script [6] does not fail with a negative number as the entry. Try it yourself by modifying script [1].
#

The repeat statements of scripts [4] and [5] can be used for just about any purpose. You may perform a loop to make sure
- a user chooses a file or folder,
- a word is present in a particular file,
etc.

In contrast to the general purpose repeat statements of scripts [4] and [5], scripts [1] and [2] are meant for number-related conditions. There are a couple more of such repeat statements.

[8]
```
repeat with counter from 1 to 5
	say "I drank " & counter & " bottles of coke."
end repeat
```

As you can see, you can use the variable of statement [8.1], i.e. ‘counter’, within your script. However, you can not change the variable to another value within the repeat block.

[9]
```
repeat with counter from 1 to 5
	say "I drank " & counter & " bottles of coke."
	set counter to counter + 1
end repeat
```

If you run the script, no spoken sentences are skipped and all bottles from 1 to 5 are consumed.

In statement [9.1], a step size of 1 is used by default. If you want a different step size, you can [10.1].

[10]
```
repeat with counter from 1 to 5 by 2
	say "I drank " & counter & " bottles of coke."
end repeat
```

In script [10], the step size is 2, and you will hear a sentence spoken three times (for the counter values of 1, 3 and 5).

If you have a list, and each item has to be used by or subjected to some operation, you could count the number of items in the list, and perform a repeat loop as in script [11].

[11]
```
tell application "Finder"
	set refToParentFolder to alias "Macintosh HD:Users:ik:Documents:"
	set listOfFolders to every folder of refToParentFolder
	set noOfFolders to the count of y
	repeat with counter from 1 to noOfFolders
```

```
            -- actions here
        end repeat
end tell
```

However, AppleScript has an elegant alternative, a demonstration of which is given below. The script [12] below allows you to determine the number of folders in a folder selected by a user. Then a repeat statement is used to create a list of the names of all the folders present.

[12]

```
set folderSelected to choose folder "Select a folder"
-- To find out which folders are present in the selected folder, we have to ask
        the Finder to give us the answer.
-- Note: “every folder” does NOT include folders inside other folders. It is just
        the folders you would see if you’d open the folder in the Finder.
tell application "Finder"
        set listOfFolders to every folder of folderSelected
end tell
-- The result is a list of folder references (paths), which can be processed
        outside the Finder tell block
-- Outcomment all the following statements and use the result field to see
        that the list ‘listOfFolders’ contains items that look like this:
        folder “reports” of folder “Documents” of folder “ik” of folder “Users”
        of startup disk of application “Finder”.
-- Only the Finder and the AppleScript component of Mac OS X can deal with
        such  references.

-- The folder names are to be stored in a new list, which is created here
set theList to {}
repeat with aFolder in listOfFolders
        -- We have references to folders, and because a reference contains the
                name of the folder, the AppleScript component of Mac OS X can
                obtain the name without having to rely on the Finder
        -- If you want to find out another property of the folders, e.g., the size
                (in bytes) of the folders, a trip to the Finder is required (i.e. a
                tell block for the next statement)
        set temp to the name of aFolder
        -- Here we add the name to the list
        set theList to theList & temp
end repeat
```

CHAPTER 14

HANDLING EVERYTHING

AppleScript's English-like nature helps to make scripts easy to read and write. We have seen that it is your responsibility too. For example, it is up to you to choose descriptive variable names, and to provide useful comments in your script. AppleScript can help you to keep your scripts more readable by providing 'handlers'. Imagine that you have the same set of statements at more locations in your script. If there is a bug, you have to correct it at each of those places. AppleScript offers the possibility to group these statements and give this group a name. If you call the name, the set is executed.

Here is how to define a handler [1].

[1]
```
on warning()
	display dialog "Don't do that!" buttons {"OK"} default button "OK"
end warning
```

To use it, your script must call the handler, like this [2].

[2]
```
warning()
```

It does not matter whether the handler is defined in your script before or after the handler is called.

The handler of script [1] is pretty inflexible. It would be nice if we could tell the handler what text to display. Guess what, this is what handlers were designed for [3].

[3]
```
on warning(textToDisplay)
	display dialog textToDisplay buttons {"OK"} default button "OK"
end warning
warning("Don't do that!")
warning("Go fishing!")
```

In statement [3.1], the variable 'textToDisplay' accepts the value passed on when the handler is called (in statements [3.4] and [3.5], each of which contains a value between parenthesis, which value is passed on to the handler). When the script [3] is executed, two dialog windows are shown consecutively.

Instead of specifying the data when calling a handler, you may use a variable instead [4].

[4]
```
on warning(textToDisplay)
	display dialog textToDisplay buttons {"OK"} default button "OK"
end warning
set someText to some item of {"Don't do that!", "Go fishing!"}
```

```
warning(someText)
```

Note that the variable name used when calling the handler [4.5] differs from the one in the handler definition [4.1]. So, you do not need to know (look up) the variable name used by the handler. Of course, you will need to know what data type the handler expects.

You not only can pass information on to a handler, but it can also return information.

[5]
```
on circleArea(radius)
	set area to pi * (radius ^ 2)
end circleArea
set areaCalculated to circleArea(3)
```

The handler 'circleArea()' performs the calculation pi *3^2 and automatically returns the result. However, just like the 'get' command, this automatic returning goes for the last statement executed by a handler only. To ensure that you obtain the result you want, even if it is generated somewhere in a series of statements [6.3], use the 'return' keyword [6].

[6]
```
on older(a)
	if a > 30 then
		return "older"
	end if
	return "not older"
end older
set theAge to older(73)
```

If the comparison in statement [6.2] is true, statement [6.3], which is not the last statement in the handler, returns the result.

You are not limited to passing on single values [7.7] to a handler.

[7]
```
on largest(a, b)
	if a > b then
		return a
	end if
	return b
end largest
set theLargest to largest(5, 3)
```

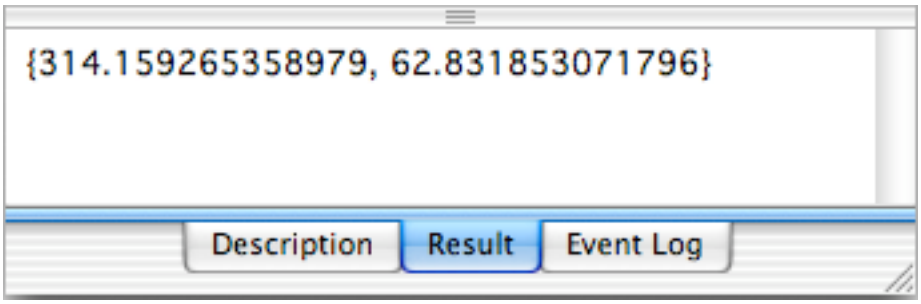
The above script returns the largest of two values. Note that in statement [7.1] two variables are provided to accept the values. Accordingly, when calling the handler, two values must be supplied [7.7]. In a real-world script, at least one of these values will be supplied as a variable.

It is also possible to return more than one value. To this end, provide the data to be returned as a list [8.4].

[8]

```
on circleCalculations(radius)
	set area to pi * (radius ^ 2)
	set circumference to 2 * pi * radius
	return {area, circumference}
	set testVar to 3
end circleCalculations
set circleProperties to circleCalculations(10)
```

The above script returns a list containing both the values for the surface area and the circumferential length of a circle with a specified radius.



Writing your script as a series of handlers and couple of statements that make up the body of your script, may well save you trouble and time thanks to an important property of handlers. Here is a question for you (which I'll answer myself): What if your script uses a variable name which is already present for a different purpose in a handler? No worries, there is no interference. To prove the point, run the following script [9].

[9]

```
set x to 5
on example()
	set a to 5 * x
end example
set y to example()
```

You will get the following error:

Within the handler, the variable x is not defined. If you want the value of x [9.1] to be known inside the handler, you will have to pass it on, as demonstrated in scripts such as [4] .

Conversely, changing a variable within a handler has no effect outside the handler. Here is the proof [10].

[10]

```
set x to 5
on example()
	set x to 6
end example
example()
get x
```

The result field shows that x is 5. So, there is no interaction between handlers and the script they are in except for what is explicitly passed on to the handler and returned by the handler. I should say there is a way to make the variable 'x' of statement [9.1] or [10.1] functional in a handler without passing it on. However, this makes the script harder to read and to debug. In addition, it negates the great advantage of handlers discussed in the next paragraph.

#
If you are curious how to create a variable that is valid both inside and outside handlers, here is how to do it: Preceed the variable name with the keyword 'global'. It is recommended to do this in the first line(s) of your script, so you can easily find out which variables display this behavior.

```
global x

set x to 5
on example()
	set x to 6
end example
example()
get x
```

Now, the result field shows that x is 6. From the above discussion, that variables are local by default.

While I can't say I'm fond of defining variables globally, for the reasons explained above and below, there is one type of global variables that come in quite handy. They come with the peculiar name 'property'.

```
property x : 1

on example()
	set x to x + 1
end example
example()
get x
```

Do run the above script several times and pay close attention to the result. The value of the property is remembered between several runs of the script. You will note from the above script, that the property is valid both inside and outside handlers, in other words, it behaves like a global variable.
#

Apart from the above important advantages (readability etc.), an additional big advantage is that you can reuse handlers in other scripts you create. Because the handler has been used

and was tested for another script, you can be pretty sure it will work fine in the other scripts. This saves you time to create the new script. Don't think you can copy statements that are not in the form of handlers from another script as easily. You would have to go through the script to figure out what to copy, which takes a lot of time. In addition, you run the risk that what you copy is incomplete and/or does not function properly in your new script. Trust me, handlers are the way to go.

#
Ok, so you can now enjoy effectively reusing handlers in other scripts. But what if you find a bug in the handler or think of an enhancement later on? You would have to change all your scripts. AppleScript has a solution for this problem, the 'load script' command. All you have to do is
1) save one or more handlers as a compiled script.
2) include in the script needing the handlers the statement

```
set aVariableName to (load script "path here")
```

To use a handler of the compiled script, you must use a tell block.

```
tell aVariableName
	handlerName()
end tell
#
```

One peculiar aspect of handlers is that if you use tell blocks, without an additional measure, the handler appears not to be defined within the tell block. If you run script [11]

[11]
```
on showProgress()
	display dialog "Job done!"
end showProgress

tell application "Finder"
	empty the trash
	showProgress()
end tell
```

the following error message pops up.

The cure is simple. Just indicate that the handler is of the script itself [12.7].

[12]
```
on showProgress()
```

```
        display dialog "Job done!"
end showProgress

tell application "Finder"
        empty the trash
        showProgress() of me
end tell
```

This is necessary for handlers only, not for variables [13], as you can see below.

[13]

```
set x to 4
tell application "Finder"
        set x to 5
end tell
get x
```

The result is 5.

CHAPTER 15

SOURCES OF INFORMATION

The only goal of this book was to teach you the basics of AppleScript. If you have been through this book twice, and tried examples with your own variations thereof yourself, you are ready to learn how to script the applications which are important to you.

An important advice for those who want to start writing their own scripts: Don't do it! Somebody else may already have done what you need. So, can save time by looking for such a script, and modifying it to suit your needs. Where do you find these scripts? On the Internet, of course. Your first visit should be Apple's site at

http://www.apple.com/applescript

which has many valuable links.

I strongly recommend that you also bookmark

www.AppleScriptSourcebook.com

and

http://macscripter.net

Macscripter has a forum where you can post your questions. Helpful members as well as the macscripter staff will do their best to help you. Yes, we are talking Macintosh community here. The site also has a large series of links to other sites and sources of information.

I hope you enjoyed this book and will pursue with AppleScript. Oh, and, please, do not forget chapter 0.

Bert