Basics of AppleScript

NAYAN SETH

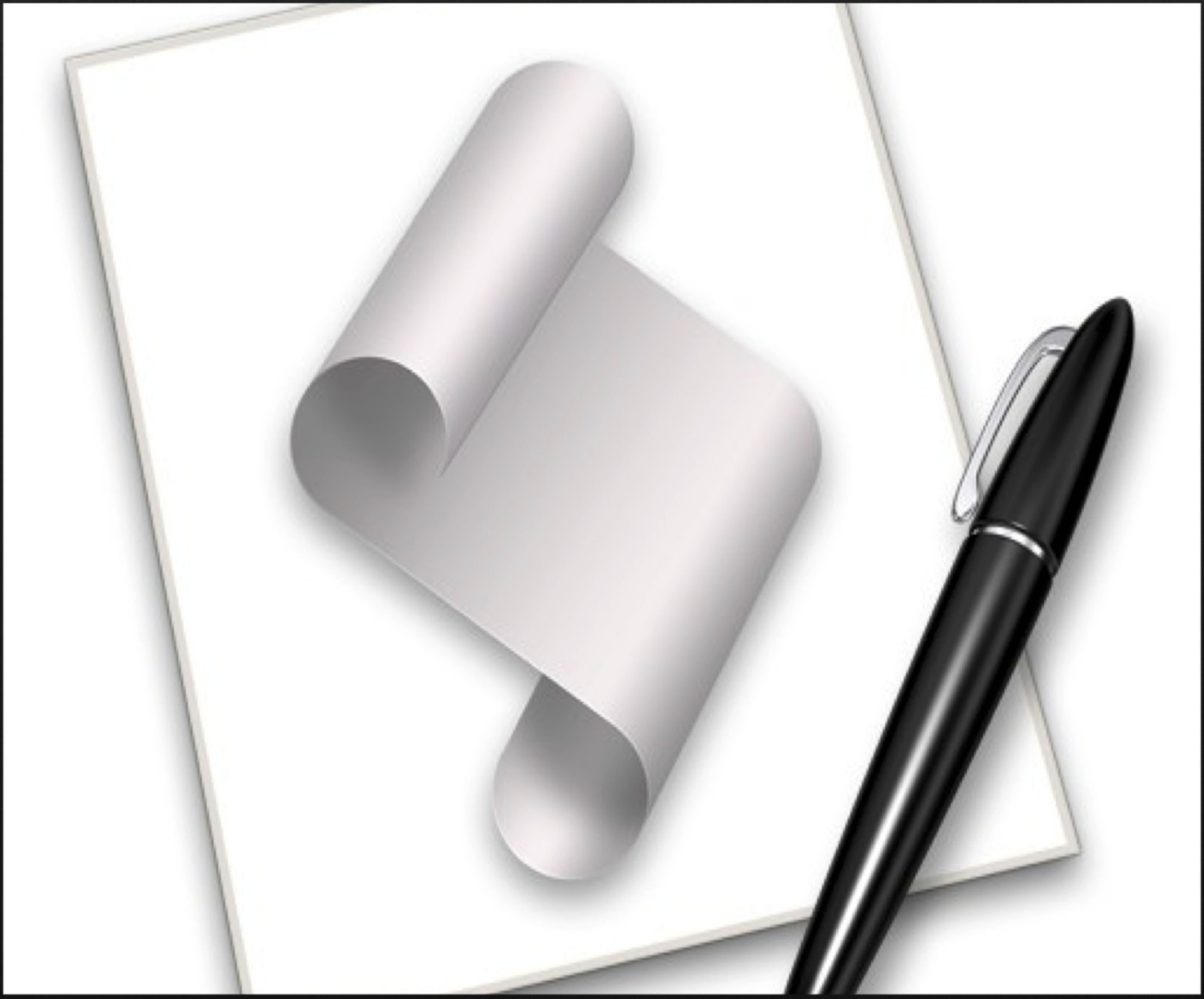


Table of Contents

Basics of AppleScript

This book is for all those Mac users who want to learn about AppleScript i.e. the scripting language for OS X. This book covers the basics. You can use the skills you learn from this book, to write complex scripts.

ISBN (13 for iBooks) - 9788192978420

ISBN (13 for EPUB) - 9788192978437

Scripts - Download on Gists

iTunes (iBooks Format) - http://itun.es/us/G3CvZ

iTunes (EPUB) - https://itun.es/us/sAOc3

Google Play - http://goo.gl/HA7YKD

Gitbook - Click Here

Website View - View on Tech Barrack Solutions

Dedicated To

My revered ... father and mother

My trusted ... family and friends

Copyright

About The Author

Nayan Seth is currently pursuing Computer Science in Dwarkadas J. Sanghvi College of Engineering, Mumbai, India. He is the founder and CEO of Tech Barrack Solutions LLP. You can connect with him at Twitter, Facebook, Google Plus, YouTube and Linkedin. Click here to send me an email.

About The Book

AppleScript is a very cool scripting language which can be used to automate tasks on your Mac.

It is not case sensitive and uses simple English terms. It provides a simple and effective way for communication between your Mac and the Applications running on it. Here are few examples:

1. Rename thousands of files
2. Compress images
3. Add watermark to documents, etc.

For me, the difference between scripting and programming is the nothing but the simplicity. Writing scripts is comparatively simpler than programming. This book contains Script as well as screenshots. To download all the scripts used in this book, click here. The scripts can be read online on Gists.

Make sure you have a Mac. We will use AppleScript Editor. It is an application which comes along with Mac OS X.

Basic Commands

In this chapter, we will go through the most basic commands used in AppleScript. These commands will include how to add *comments*, *say* command, *beep* command and *tell* command.

Comments

Script [1.1.1]: `--This is a single line comment`

Explanation: Comments are used to explain something in the script or give information about the script. Comments are ignored by AppleScript during execution. Using -- we can have single line comments.

Script [1.1.2]: `#This is another single line comment`

Explanation: Using # we can write single line comments

Script [1.1.3]:

```
(*
this is
a multi
line comment
*)
```

Explanation: Multi line comments begin with (*and end with*). It can contain multiple lines of explanation. Usually used to specify author and copyright status.

Say

Script [1.2.1]: `say "Hi I am a Mac"`

Explanation: This command will make your Mac say anything you want. Whatever you want the Mac to say has to go inside ""

Script [1.2.2]: `say "Hi I am a Mac" using "Zarvox"`

Explanation: This command will make your Mac say anything you want. Whatever you want the Mac to say has to go inside “”. However, this time Mac will use the voice specified. Though AppleScript is case sensitive, the name of the voice should match.

Beep

Script [1.3.1]: `beep`

Explanation: A beep sound will be played. The beep sound can be changed in System Preferences.

Script [1.3.2]: `beep 10`

Explanation: Beep sound will be played 10 times.

tell

Script [1.4.1]:

```
tell application "Finder"
    -- insert actions here
end tell
```

Explanation: This command will tell application Finder to execute the specified commands

Script [1.4.2]:

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

Explanation: This Script will ask Finder to empty the trash first and then open the startup disk i.e. Macintosh HD.

Script [1.4.3]:

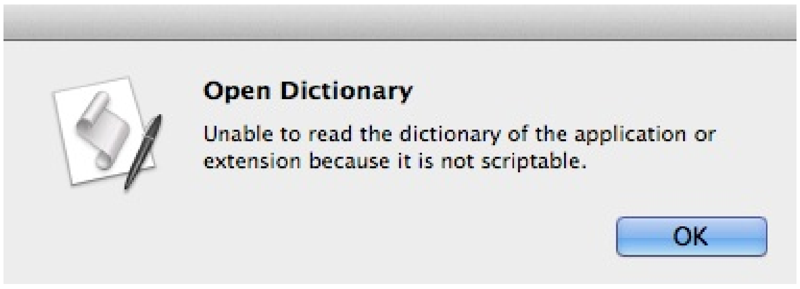
```
tell application "Finder"
    empty the trash
    beep 10
end tell
```

Explanation: This Script will ask Finder to empty the trash first and then beep 10 times. Beep can be used inside the the tell application command. Finder does not know anything about the beep command, but AppleScript component of Mac OS X knows about it and hence will execute it.

Script [1.4.4]:

```
tell application "Finder"
    empty the trash
    beep 10
end tell
open the startup disk
```

Explanation: Finder will empty the trash and beep 10 times. Thereafter runtime error will occur because open the startup disk command is not known to AppleScript component of Mac OS X, it is known to Finder application. An error notification will pop up in dialog.



Open Dictionary

Unable to read the dictionary of the application or extension because it is not scriptable.

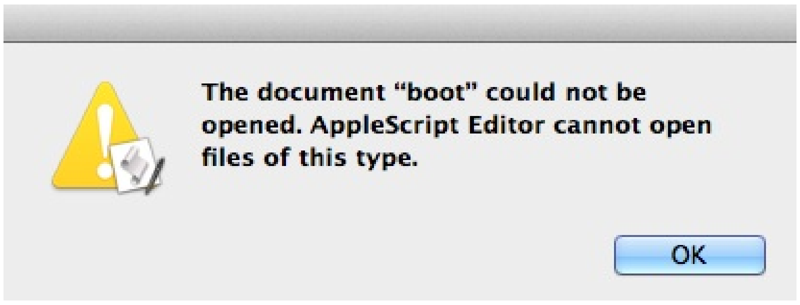
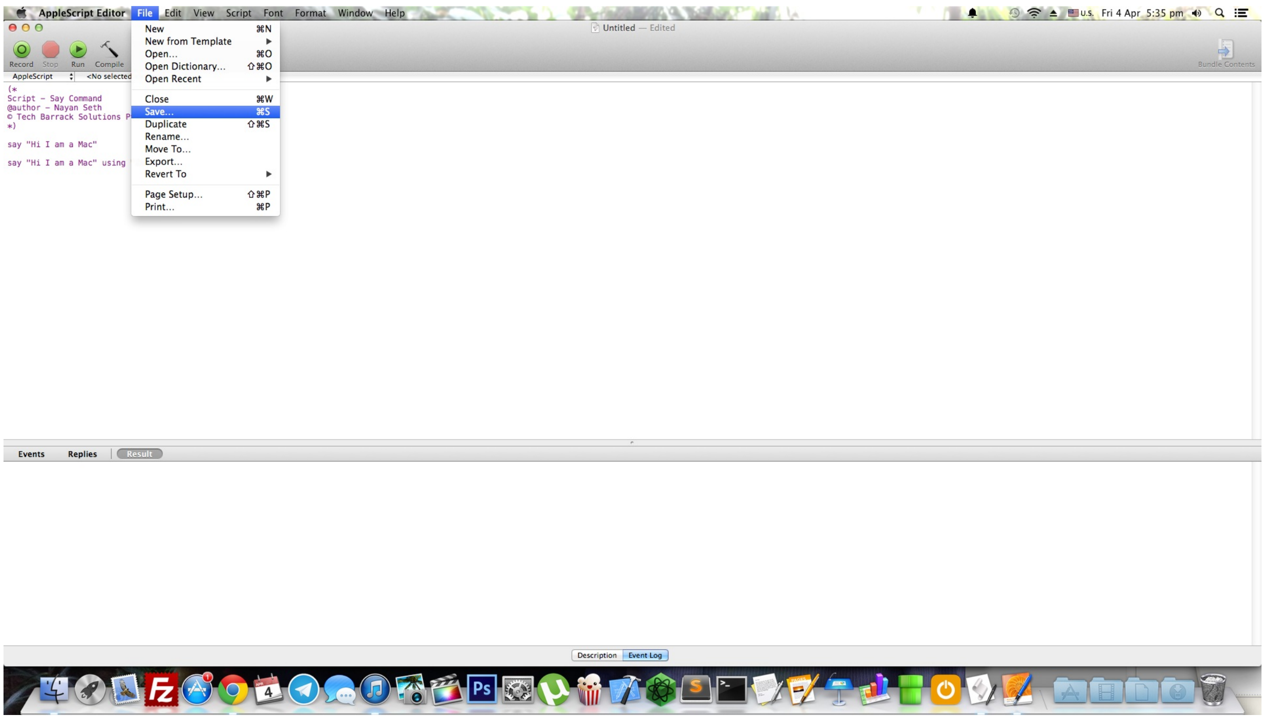

The document “boot” could not be opened. AppleScript Editor cannot open files of this type.

Figure 1.4.4 Runtime Error

Save Compile Run

In this chapter, we will learn how to *save*, *compile* and *run* the script. We will also learn how to *export* the script.

Save



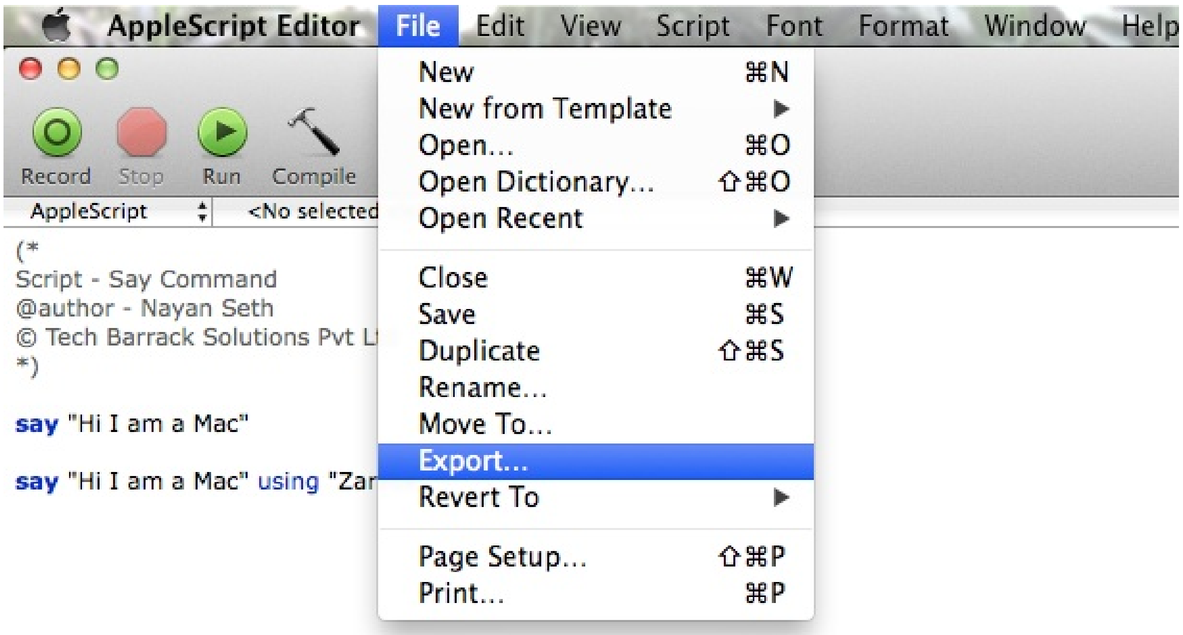
```
(*
Script – Say Command
@author – Nayan Seth
© Tech Barrack Solutions P
*)

say "Hi I am a Mac"

say "Hi I am a Mac" using
```

Figure 2.1.1 Saving a Script

Explanation: Go to File and click on Save. Give name to your script. And in file format, you can either select Script/Script Bundle/Application/Text. Currently we will save it as Script only. Later we will see how to save the file as an Application. As soon as you save the file as a script, the contents of the script gets colored. Now you are ready to run the script.

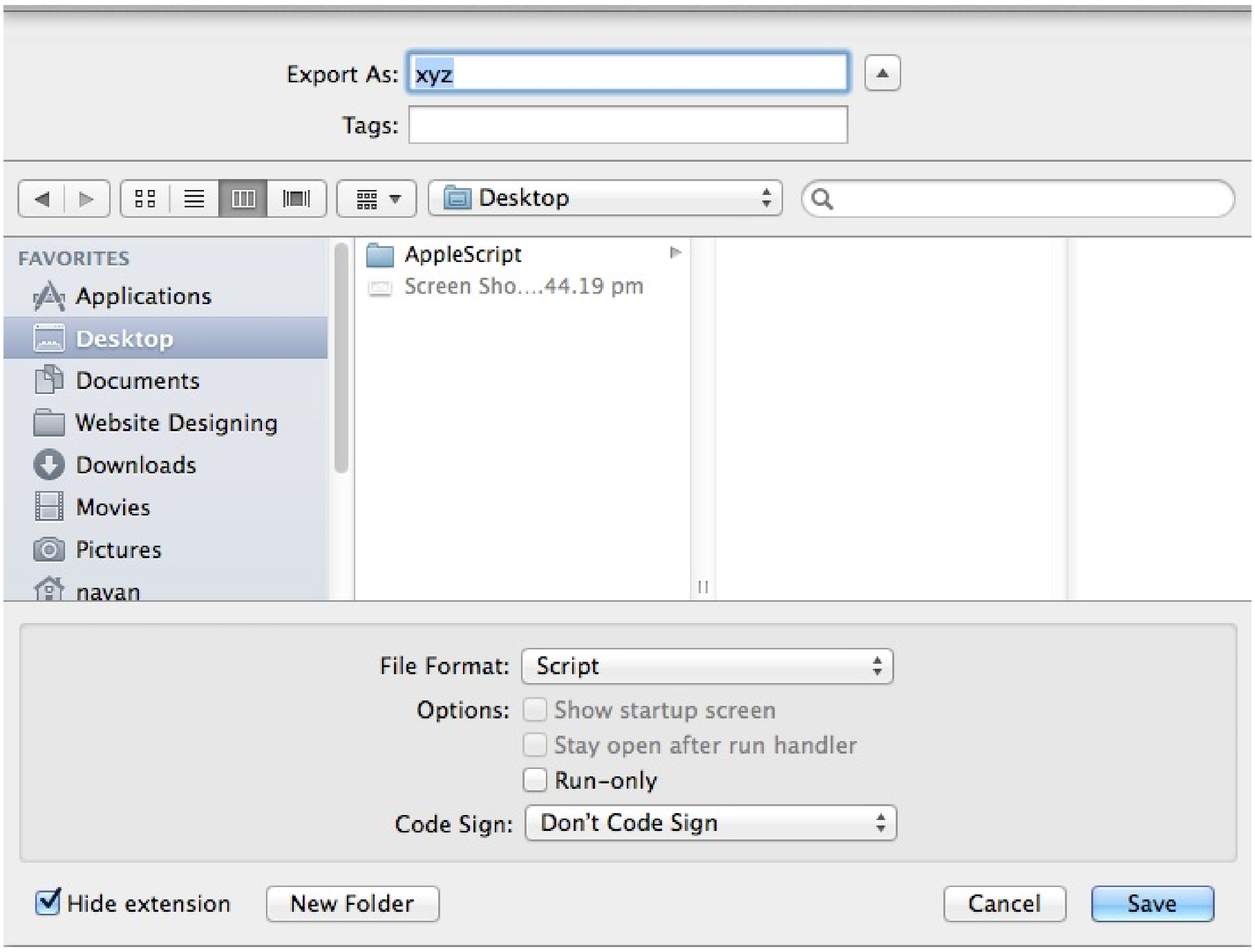
Export

What if I have already saved the file as a script and now I want to save it as a text/application/script bundle. How do I do that? Well it's simple.



```
(*
Script - Say Command
@author - Nayan Seth
© Tech Barrack Solutions Pvt Lt
*)

say "Hi I am a Mac"

say "Hi I am a Mac" using "Zar
```

Figure 2.2.1 Exporting a Script



Export As: xyz

Tags:

File Format: Script

Options: Show startup screen

Stay open after run handler

Run-only

Code Sign: Don't Code Sign

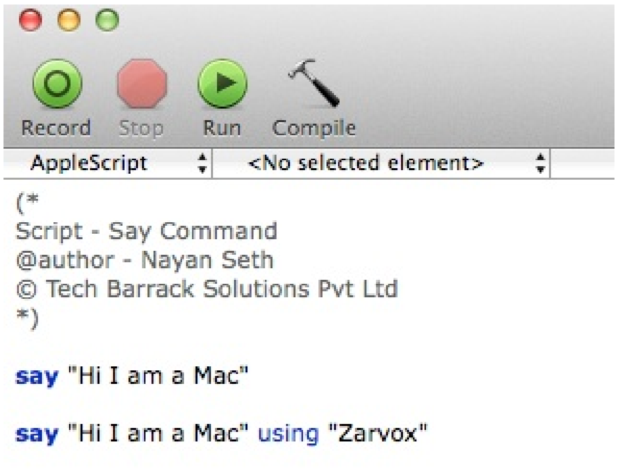
Figure 2.2.2 Selecting File Format

Explanation: Go to File and click on Export. Give name to your script. And in file format, you can either select Script/Script Bundle/Application/Text.

As soon as you save the file as a script, the contents of the script gets colored. Now you are ready to run the script.

Compile

Next step is to compile the script. We just need to click on the Compile button. The compiler will check for errors. If there is something wrong with the script, it will throw an error.

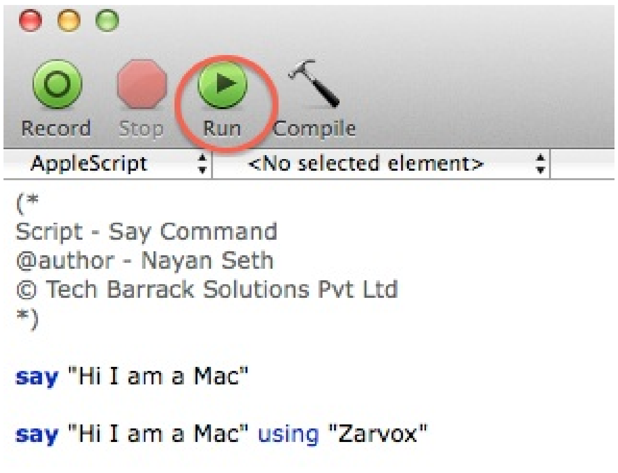


```
(*
Script - Say Command
@author - Nayan Seth
© Tech Barrack Solutions Pvt Ltd
*)

say "Hi I am a Mac"

say "Hi I am a Mac" using "Zarvox"
```

Figure 2.3.1 Compiling the Script

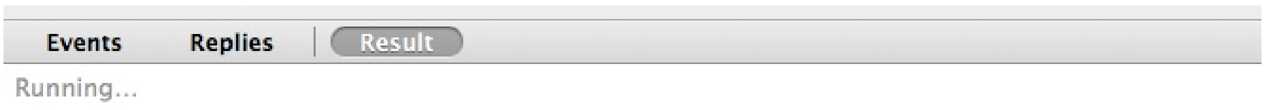
Run

If the script is successfully compiled then the next step is to Run the script. It can be done by clicking on the Run button. On doing so, the Result tab will show running and print the results if necessary. *Result tab shows output of last executed statement only.*



```
(*
Script - Say Command
@author - Nayan Seth
© Tech Barrack Solutions Pvt Ltd
*)

say "Hi I am a Mac"

say "Hi I am a Mac" using "Zarvox"
```

Figure 2.4.1 Running the Script



```
Running...
```

Figure 2.4.2 Result Tab

Simplifying Scripting

AppleScript is super awesome. It has got some cool features too. Consider the tell command. You would simply type:

Script [3.1]:

```
tell application "Finder"
    -- insert actions here
end tell
```

This is too much to write. Well in AppleScript, you can right click on the window and select Tell Blocks > Tell “Finder”.

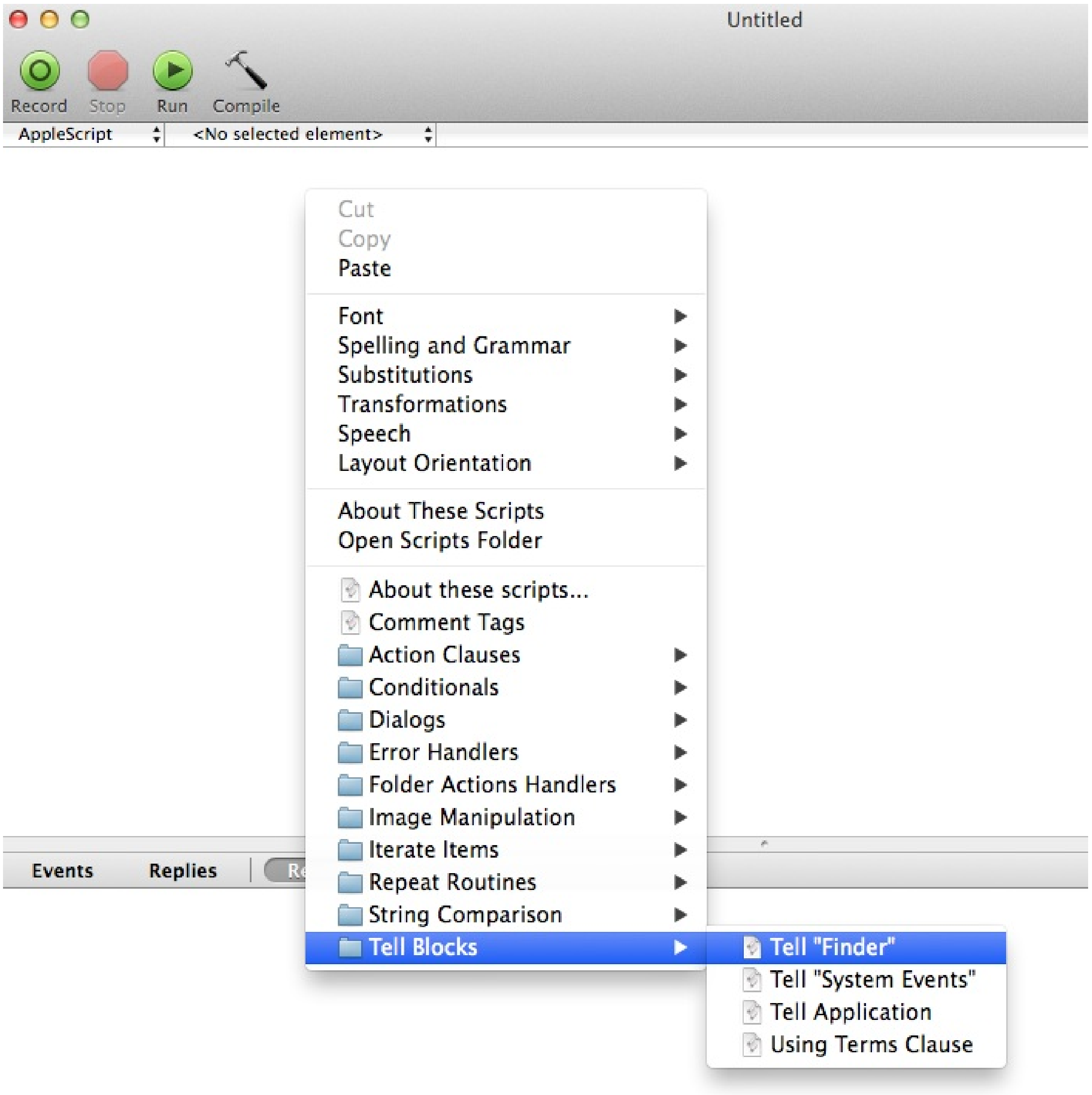


Figure 3.1 Tell Blocks

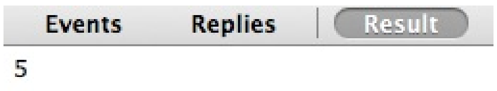
Variables & Arithmetic

In school, we use to make use of variables like x and y for algebraic operations. Similarly we make use of variables in AppleScript too.

Declaring Variables

Script [4.1.1]: `set x to 5`

Explanation: This will create a variable with the name *x* and assign it a value of 5. The result tab will print 5.



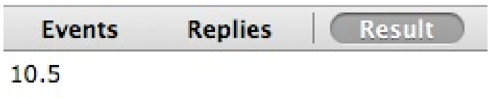
```
5
```

Figure 4.1.1 Printing Integer

Script [4.1.2]:

```
set x to 5
set y to 10.5 --creates a floating point number
```

Explanation: This will create a variable with the name *x* and *y* and assign a value of 5 and 10.5 respectively. However the result tab will print 10.5 (the last value).



```
10.5
```

Figure 4.1.2 Printing Floating Point Number

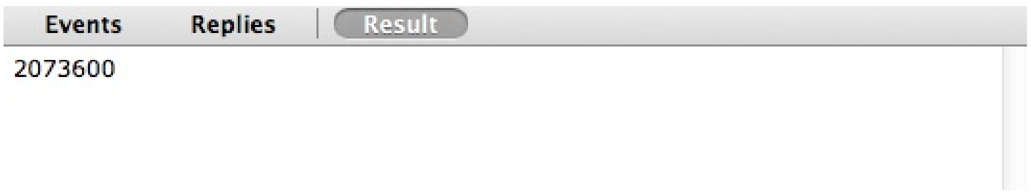
Another important thing here is naming of variables. There is pre-defined set of rules that needs to be followed. Click here to learn more.

Just remember one thing, the variable name should not be same as a keyword used in AppleScript like say, beep, etc. It is a good practice to composite words. It is a good practice to keep the second and additional word in title case (e.g. pictureWidth, getDivisionResult, etc).

Script [4.1.3]:

```
set pictureWidth to 1920
set pictureHeight to 1080
set resolution to pictureWidth * pictureHeight
```

Explanation: This will create a variable with the name *pictureWidth* and *pictureHeight* and assign them with a value 1920 and 1080 respectively. As specified resolution = pictureWidth x pictureHeight i.e. 1920*1080 (2073600). Hence in the result tab 2073600 will be printed.



```
2073600
```

Figure 4.1.3 Output of Multiplication

Strings & Dialog

Strings can make your program short...Dialog can be used for adding GUI tinge.

Strings are used to store text. It can be empty or it can contain just a single letter.

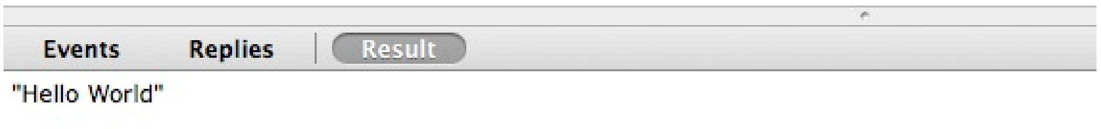
Declaring Strings

Strings are declared in double quotes. AppleScript makes use of few symbols to understand exactly what the programmer is trying to script.

Script [5.1.1]:

```
set emptyString to ""
set spaceString to " "
set hello to "Hello World"
```

Explanation: This will create 3 strings. First an empty String which contains nothing. Second a string which contains just a space and third a string which contains Hello World. Result tab will print Hello World



```
"Hello World"
```

Figure 5.1.1 Output of String

Dialogs

Script [5.2.1]: `display dialog "Hello World"`

Explanation: It will display a dialog with text Hello World. It is similar to javascript pop up in web browsers.

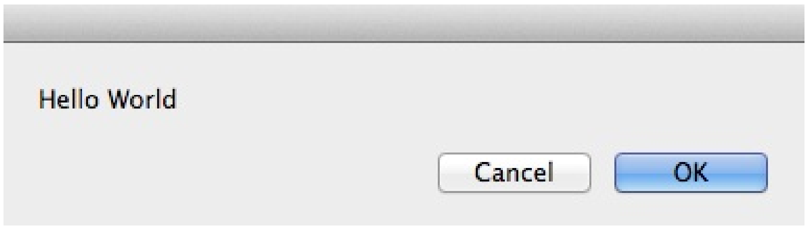

Hello World

Figure 5.2.1 Displays a Dialog

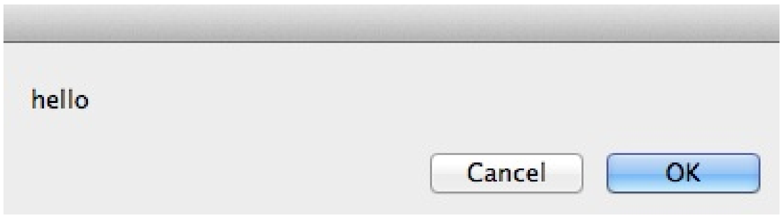
Script [5.2.2]:

```
set hello to "Hello World"
display dialog "hello"
display dialog hello
```

Explanation: This will create a variable hello with text Hello World. A dialog will be displayed with text hello and not the text of variable hello.

This is because we have mentioned the text in double quotes. If we want Hello World to be printed in dialog, then we need to mention the variable name without double quotes or Hello World in double quotes.

The third line will print contents of hello variable in the dialog.


hello

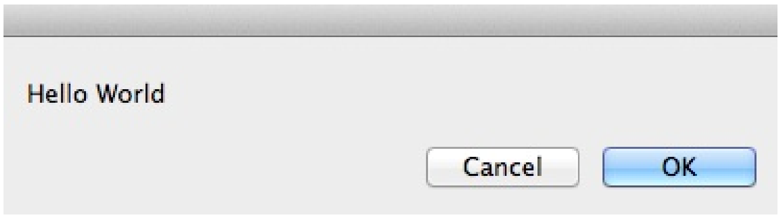

Hello World

Figure 5.2.2 Strings in Dialog

Merging Strings

Script [5.3.1]:

```
set firstName to "Nayan"
set lastName to "Seth"
set myName to firstName & " " & lastName
display dialog myName
```

Explanation: Variable *firstName* will store Nayan and variable *lastName* will store Seth. Variable *myName* will store Nayan + space + Seth that is Nayan Seth. And a dialog will pop up displaying myName.

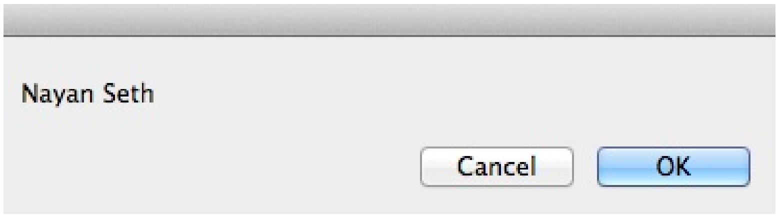
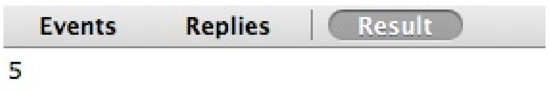

Nayan Seth

Figure 5.3.1 Output of Merged Strings

Length of String

Script [5.4.1]: `set strLength to the length of "Nayan"`

Explanation: This will create variable *strLength* and assign it with a value equal to the length of String “Nayan” that is 5.



```
5
```

Figure 5.4.1 Length of a String

Escape Sequences

What if I want to add double quotes inside the string declaration. Escape Sequences allow us to add new line, double quotes, etc in the string itself. It is always declared using a .

Script [5.5.1]:

```
set exampleString to "Nayan is \"Awesome\""
display dialog exampleString
```

Explanation: In output we will have Nayan is "Awesome"

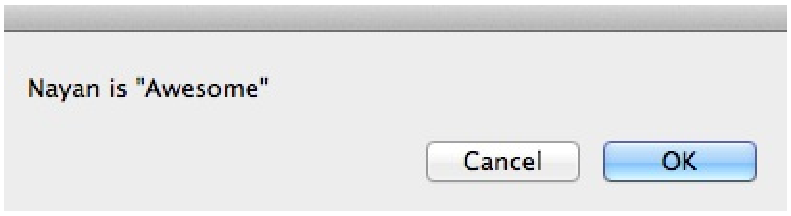

Nayan is "Awesome"

Figure 5.5.1 Escape Sequence Output

Script [5.5.2]:

```
set exampleString2 to "Nayan is     Awesome"
display dialog exampleString2
```

Explanation: In output we will have Nayan is Awesome. There will be tab space between is and Awesome as \t signifies tab space.

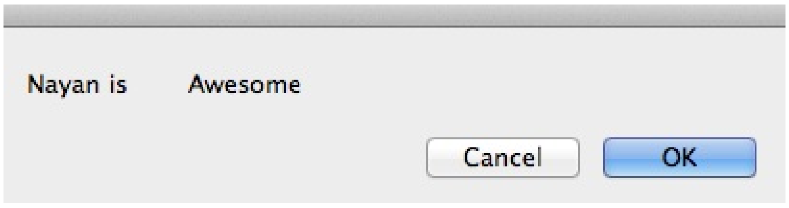

Nayan is   Awesome

Figure 5.5.2 Adding Tab Space

Another thing to note is that if you add an escape sequence and then compile the program, the escape sequence will get converted to what it actually means.

e.g. \t will get converted to (tab space)

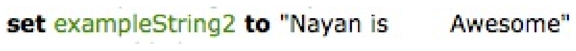

```
set exampleString2 to "Nayan is      Awesome"
```

Figure 5.5.2-2 Compiled Program with Escape Sequence

Other Escape Sequences include \n for new line, \? for question mar, \’ for single quotes, etc.

Script [5.5.3]: `display dialog "Nayan \\\" Seth"`

Explanation: In output we will have a dialog. The dialog will print Nayan \" Seth. \ gets converted to \ and \" gets converted to "

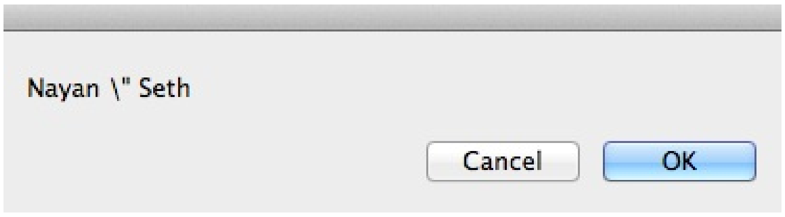

```
Nayan \" Seth
```

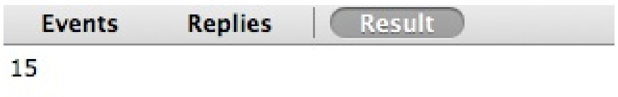
Figure 5.5.3 Printing \ and "

Coercion

Script [5.6.1]:

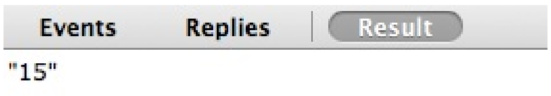
```
set input to "15" as number
set numToString to 15 as string
```

Explanation: input variable was string but as we specified it to be number, it becomes of type number and stores 15. numToString was number type but as we specified string, it stores “15”. This is coercion.



```
15
```

Figure 5.6.1 Output of Number as String



```
"15"
```

Figure 5.6.1-2 Number as String

Dialog With Custom Buttons

Script [5.7.1]:

```
display dialog "Yo!!!" buttons {"Click Me"}
```

Explanation: Displays a dialog with text Yo!!! and button Click Me.

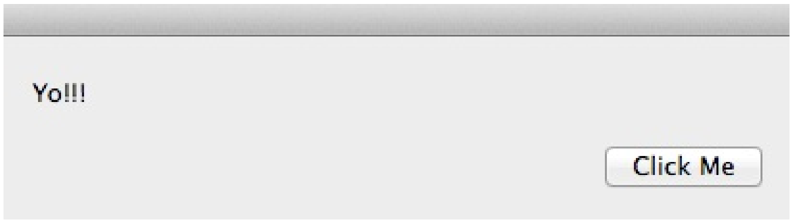
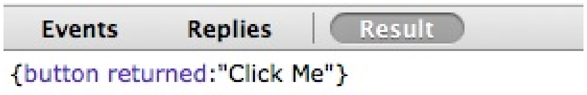


Figure 5.7.1 Custom Dialog Buttons



```
{button returned:"Click Me"}
```

Figure 5.7.1-2 Result Tab Shows Button Returned

Script [5.7.2]:

```
display dialog "Yo!!!" buttons {"Click Me", "Don't Click Me", "Test Me"} default button 1
```

Explanation: Displays a dialog with text Yo!!! and buttons Click Me, Don’t Click Me and Test Me. We have also specified that the default button to be 1 that is Click Me and hence it will be highlighted.

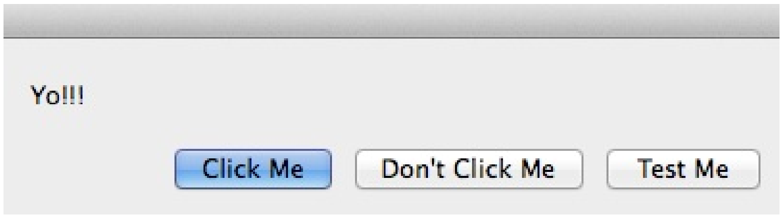


Figure 5.7.2 Dialog with Custom Button and Default Button Set

Lists

A list contains number of items. We usually make lists to remember multiple items.

If you have ever made websites using HTML, then you must have come across ul and ol which allows us to create lists. AppleScript enables us to declare lists too.

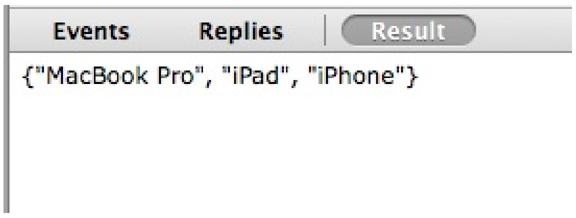
Declaring Lists

AppleScript is a scripting language and hence can perform limited tasks. But this makes it really simple to Script as the number of keywords in AppleScript are less in comparison to normal programming languages.

Script [6.1.1]:

```
set myList to {"MacBook Pro", "iPad", "iPhone"}
```

Explanation: The above statement will create a list named *myList* and it will contain the following items i.e. MacBook Pro, iPad and iPhone.



```
{"MacBook Pro", "iPad", "iPhone"}
```

Figure 6.1.1 Declaring a List

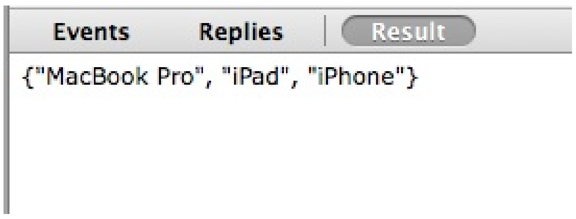
Print List

Script [6.2.1]:

```
set myList to {"MacBook Pro", "iPad", "iPhone"}
get myList
```

Explanation: The above statement will create a list named *myList* and it will contain the following items i.e. MacBook Pro, iPad and iPhone.

The command get is used to print the contents of list specified in the result tab.



```
{"MacBook Pro", "iPad", "iPhone"}
```

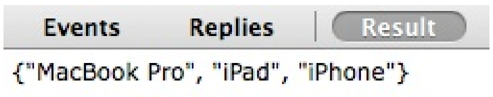
Figure 6.2.1 Print List Using get Keyword

Merge Lists

Script [6.3.1]:

```
set laptop to {"MacBook Pro"}
set tablet to {"iPad"}
set phone to {"iPhone"}
set devices to laptop & tablet & phone
get devices
```

Explanation: The first three statements will create three lists named *laptop*, *tablet* and *phone* which will contain Macbook Pro, iPad and iPhone respectively. The fourth statement will create a list named devices which will merge the contents of laptop, tablet and phone. & is used to merge lists. It is the same symbol used to merge Strings. Finally the get command is used to print the devices list in result tab.



```
{"MacBook Pro", "iPad", "iPhone"}
```

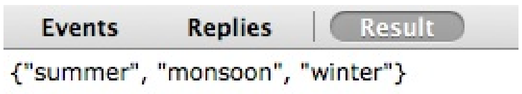
Figure 6.3.1 Merged Lists

Modifying Lists

Script [6.4.1]:

```
set seasons to {"summer", "spring", "winter"}
set item 2 of seasons to "monsoon"
get seasons
```

Explanation: Here a list named seasons is created. It contains summer, spring and winter. However I want to modify the 2nd item. So with the help of second statement, I modified spring to monsoon. Finally I used the get command to print the new modified list.



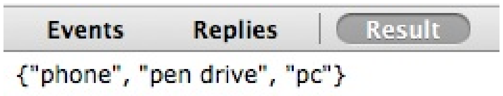
```
{"summer", "monsoon", "winter"}
```

Figure 6.4.1 Modified List

Script [6.4.2]:

```
set buy to {"phone", "usb", "pc"}
set 2nd item of buy to "pen drive"
get buy
```

Explanation: Here a list named buy is created. It contains phone, usb and pc. However I want to modify the 2nd item. So with the help of second statement, I modified usb to pen drive. Finally I used the get command to print the new modified list. This is another way of modifying a list. The only difference is in item 2 and 2nd item.



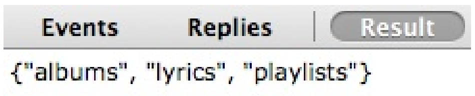
```
{"phone", "pen drive", "pc"}
```

Figure 6.4.2 Modified Lists (Different Method)

Script [6.4.3]:

```
set musicList to {"songs", "lyrics", "artists"}
set first item of musicList to "albums"
set last item of musicList to "playlists"
get musicList
```

Explanation: This is another method to modify the first and last item of the list by simply using the command first item or last item. Here the list is named musicList. It contains songs, lyrics and artists. The second and third statement sets the first and last item of the list to albums and playlists respectively. Finally the get command is used to print the list.



```
{"albums", "lyrics", "playlists"}
```

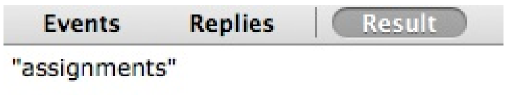
Figure 6.4.3 Modifying First & Last Item of the List

Extracting Items From Lists

Script [6.5.1]:

```
set bag to {"books", "assignments"}
set engineering to the last item of bag
```

Explanation: Here a list named bag contains books and assignments. What if you want to obtain last item of list and you don’t know the size of list?? It’s simple. Just create a new variable and set it to the last item of listName. Here engineering is the variable which will contain the last item of bag that is assignments.



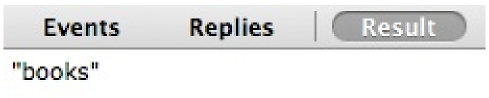
```
"assignments"
```

Figure 6.5.1 Getting Last Item From List

If the second statement was,

`set engineering to the last item of bag`

then output would be



```
"books"
```

Figure 6.5.1-2 Getting First Item From List

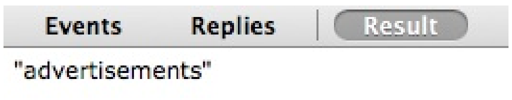
Script [6.5.2]:

```
set newspaper to {"articles", "author", "advertisements"}
set itemValue to item -1 of newspaper
```

Explanation: AppleScript allows us to retrieve value of items in the list using the keyword item -n where n is the number from 1 to the length of list.

Note: Last item is item -1. Subsequently item -2 is the item before last item and so on.

In the above example the list is named *newspapers* and contains articles, author and advertisements. The above Script will retrieve the last item and store it in itemValue variable.



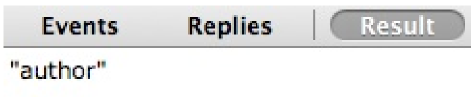
```
"advertisements"
```

Figure 6.5.2 Retrieving Last Item of List

If the second statement of Script was,

`set itemValue to item -1 of newspaper`

then output would be,



```
"author"
```

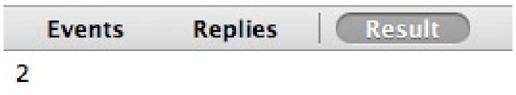
Figure 6.5.2-2 Retrieving Items From List

Size of List

Script [6.6.1]:

```
set college to {"departments", "classes"}
set listSize to the length of college
```

Explanation: Here a list named *bag* contains books and assignments. What if you want to obtain information about the size of the list? It’s simple. Just create a new variable and set it with to the length of listName. Here listSize is the variable which will contain size of list (college) that is 2.



```
2
```

Figure 6.6.1 Size of List

Part of List

What if you just want to retrieve just a part of the list?

Script [6.7.1]:

```
set vowels to {"a", "e", "i", "o", "u"}
set letters to items 2 through 4 of vowels
```

Explanation: Here I have a list named *vowels* which contains a, e, i, o and u. But I want only e, i and o.

So with the help of second statement, I am specifying that letters should contain items of vowels from 2 to 4 only.

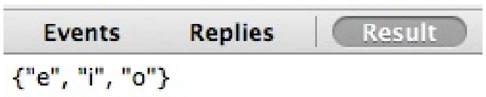
through is the keyword which helps in creating partition in the list.

However if you are a person who sends a number of messages and uses SMS language then you can use thru instead of through. The output will remain same.

Even if the second statement is changed to,

`set letters to items 4 through 2 of vowels`

the output, remains same.



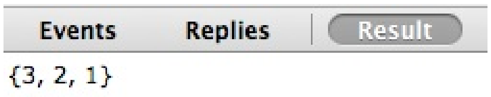
```
{"e", "i", "o"}
```

Figure 6.7.1 Partition of List

Reverse of List

Script [6.8.1]: `set reverseList to reverse of {1, 2, 3}`

Explanation: With the help of reverse keyword, we can reverse the list. The above statement will create a list named reverseList which will contain 3, 2 and 1.



```
{3, 2, 1}
```

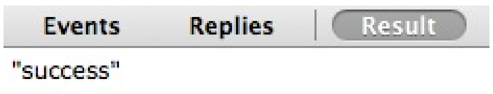
Figure 6.8.1 Reverse a List

Random Value

Script [6.9.1]:

```
set randomValue to some item of {"success", "failure", "life", "fun"}
```

Explanation: Using the keyword some item of listName or some item of {list items....}, we can obtain a random value. This random value can be anything from the list. Output will differ, every time you run the script.



```
"success"
```

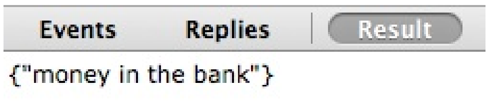
Figure 6.9.1 Random Value

Coercion

Script [6.10.1]:

```
set bank to "money in the bank"
set myBankList to bank as list
```

Explanation: Coercion allows us to convert variables from one type to another. In the above example I have created a variable bank which of String type. I then went on to create another variable named myBankList which contains the contents of bank. However a condition is specified. The condition says, when the contents of bank are being copied to myBankList, copy them as a list.



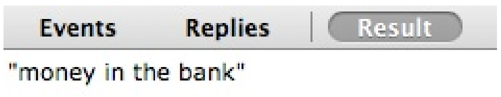
```
{"money in the bank"}
```

Figure 6.10.1 String To List

If the second statement did not specify as list that is,

`set myBankList to bank`

then output would be,



```
"money in the bank"
```

Figure 6.10.1-2 Output Without Coercion

The output no longer shows curly braces. This indicated no coercion has been performed.

Merging Different Type

Script [6.11.1]:

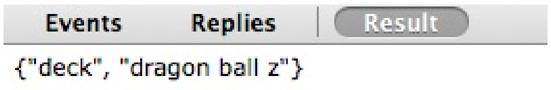
```
set cards to {"deck"}
set cartoon to "dragon ball z"
set playingCards to cards & cartoon
```

Explanation: This is an example where we merge *cards* and *cartoon* to *playingCards*. The fundamental question here is, what is the data type of playingCards.

cards is of type list and cartoon is of type string.

If you observe closely the third statement says cards & cartoon

As cards has been specified first, playingCards will inherit the data type of cards that is list.



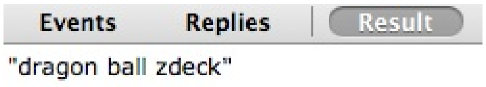
```
{"deck", "dragon ball z"}
```

Figure 6.11.1 Meging Different Types

If the third statement of Script was,

`set playingCards to cartoon & cards`

then output would have been,



```
"dragon ball zdeck"
```

Figure 6.11.1-2 Merging Different Types

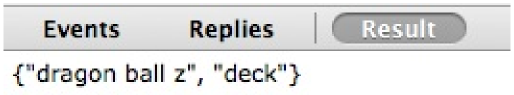
The data type of playingCards is String. This is because cartoon was the first term used in merging. And because it is String type, the contents of cards gets added with contents of cartoon.

Script [6.11.2]:

```
set cards to {"deck"}
set cartoon to "dragon ball z"
set playingCards to (cartoon as list) & cards
```

Explanation: This is an example where we merge *cards* and *cartoon* to *playingCards*. But there is a problem. I want data in the variable playingCards to be of type list.

cards is of type list and cartoon is of type string. If you observe closely the third statement says (cartoon as list). This command will convert cartoon from string to list (coercion). Now it does not matter what data type the subsequent variables are of. playingCards will be of type list.



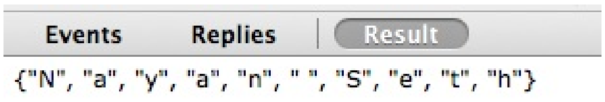
```
{"dragon ball z", "deck"}
```

Figure 6.11.2 Merging With Coercion

Character List

Script [6.12.1]: `set myLetters to every character of "Nayan Seth"`

Explanation: AppleScript enables us to convert the text of String into a character list by using command to every character of. Consider this as a character array. In Java we use the command toCharArray().



```
{"N", "a", "y", "a", "n", " ", "S", "e", "t", "h"}
```

Figure 6.12.1 Character List

Get List By Splitting Test

You have a text and you want to create a list by splitting the text by a delimiter. AppleScript allows us to specify the delimiter too.

Script [6.13.1]:

```
set myName to "Nayan Seth"
set oldDelimiters to AppleScript's text item delimiters
set AppleScript's text item delimiters to " "
set studentData to every text item of myName
set AppleScript's text item delimiters to oldDelimiters
get studentData
```

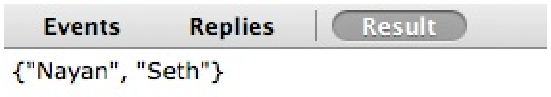
Explanation: Wow! That’s a pretty big Script for a tyro. Don’t worry its really very simple to understand.

First I created a variable named *myName* which contains String data that is "Nayan Seth".

By default AppleScript’s text item delimiters is "". So I created a variable oldDelimiters which stores default AppleScript’s text item delimiters. Now I change AppleScript’s text item delimiters to " ". Notice the space in delimiter.

Next I ask AppleScript to create a list *studentData* which will contain data of *myName*. But look closely it says use the text item in *myName*. text item is delimiter (" "). Now I change the AppleScript’s text item delimiters to default.

Finally I use get command to print *studentData*.



```
{"Nayan", "Seth"}
```

Figure 6.13.1 Split Text Using Delimiters

Custom Delimiters To String

Script [6.14.1]:

```
set myData to {"Nayan", "Seth"}
set oldDelimiters to AppleScript's text item delimiters
set AppleScript's text item delimiters to "|"
set myName to myData as string
set AppleScript's text item delimiters to oldDelimiters
get myName
```

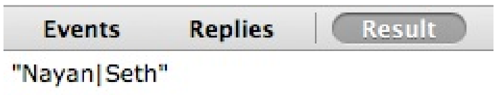
Explanation: The above Script will add custom delimiter that is | in my case.

First I created a list named myData which contains Nayan and Seth.

By default AppleScript’s text item delimiters is "". So I created a variable *oldDelimiters* which stores default AppleScript’s text item delimiters.

Now I change AppleScript’s text item delimiters to "|". Next I ask AppleScript to create a variable *myName* which will contain data of *myData* and store it as String. Just before executing this statement, I changed AppleScript’s text item delimiters. So instead of "", | will be printed between two items of list. Now I change the AppleScript’s text item delimiters to default.

Finally I use get command to print studentData.



```
"Nayan|Seth"
```

Figure 6.14.1 Adding Custom Delimiters To String

Note: It is a good practice to reset AppleScript’s text item delimiters because if you want to use these delimiters in your script again, you don’t want the custom delimiter to cause issues in your output.

More On Dialogs

In this chapter, we will learn how to find out which button has been pressed by the user, how to take user input using dialogs and how to perform coercion and display it in a dialog.

Button Pressed

Script [7.1.1]:

```
set myName to "Nayan Seth"
display dialog myName buttons {"Cancel", "Ok"} default button 2
```

Explanation: This will display a dialog with contents of myName and 2 buttons Cancel and Ok.

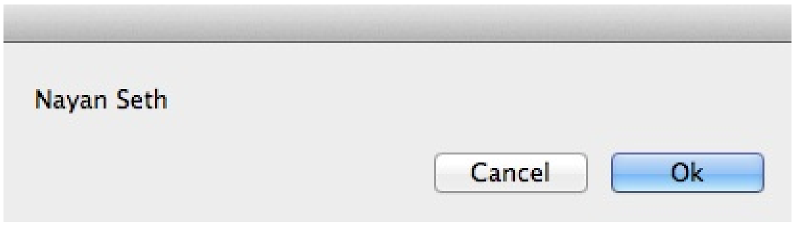

Nayan Seth

Figure 7.1.1 Dialog Output

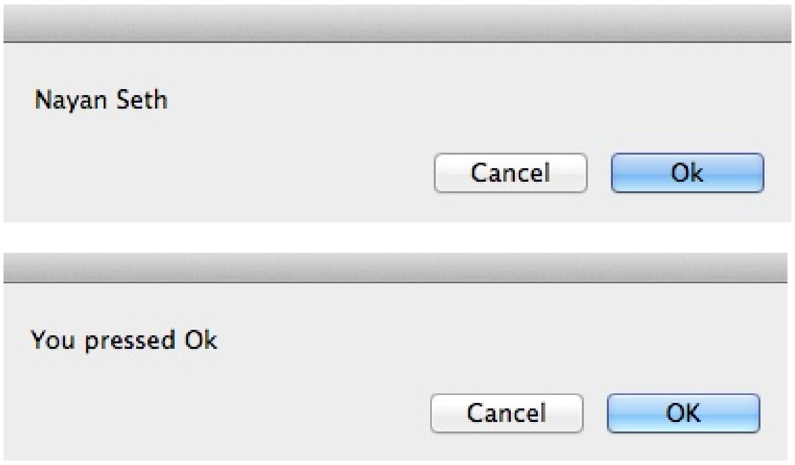
However how do I find out about the button which was pressed. Let’s modify the Script

Script [7.1.2]:

```
set myName to "Nayan Seth"
set dialogName to display dialog myName buttons {"Cancel", "Ok"} default button 2
set buttonName to button returned of dialogName
display dialog "You pressed " & buttonName
```

Explanation: Here we have the same variable myName but instead of displaying dialog directly, I assign the dialog to a variable named dialogName.

I create another variable *buttonName*. This will return the value of button which was pressed by dialogName. Then I display a dialog which prints the value of button pressed by the user.


Nayan Seth

You pressed Ok

Figure 7.1.2 Button Pressed

Dialogs can display numbers and short strings. They cannot display lists. However they can be used to take user input.

User Input

Script [7.2.1]:

```
display dialog "What is your name?" default answer ""
```

Explanation: The above statement will display a dialog and small text area where user can provide his/her input. Here the keyword default answer "" is necessary, if user input is required.

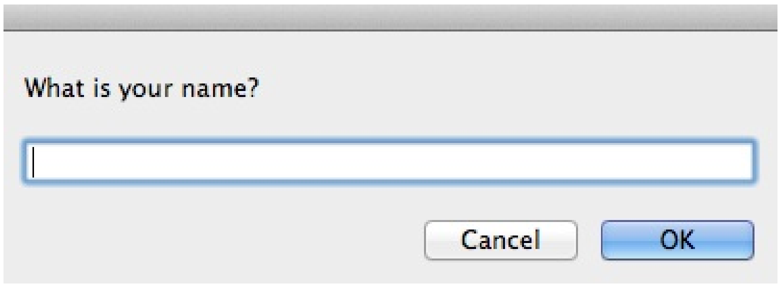


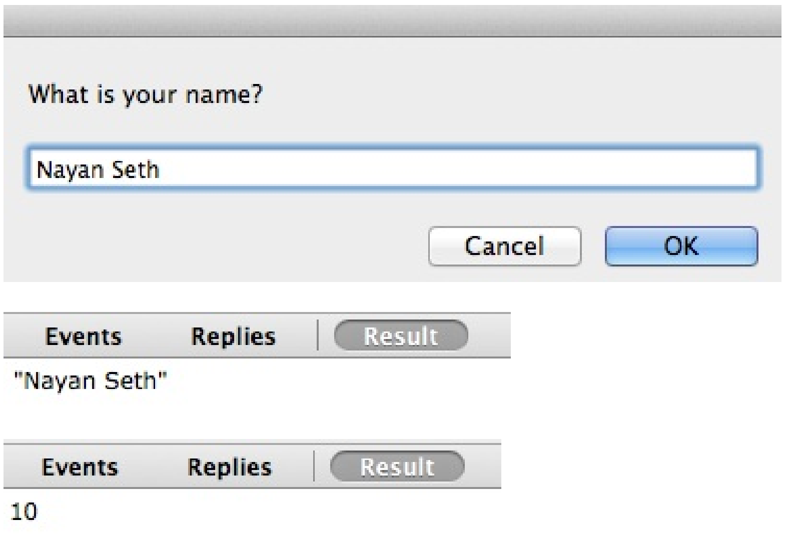
Figure 7.2.1 User Input Using Dialogs

Script [7.2.2]:

```
set myName to display dialog "What is your name?" default answer ""
set studentData to text returned of myName
set lengthSize to length of text returned of myName
```

Explanation: Variable myName will store details of dialog. The dialog will ask the user for his/her name.

Once user, enters the name as input, studentData (type String) will store the actual text entered by user whereas lengthSize (type number) will store the length of text entered by user.



Nayan Seth

```
"Nayan Seth"
```

```
10
```

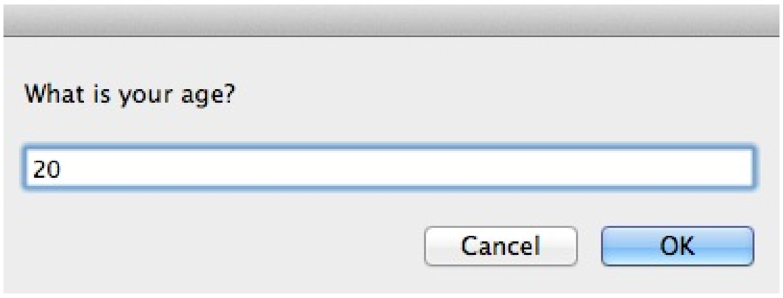
Figure 7.2.2 User Input Values Returned

Coercion

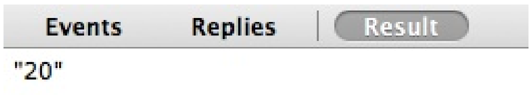
Script [7.3.1]:

```
set myAge to display dialog "What is your age?" default answer ""
set studentAge to text returned of myAge as number
```

Explanation: Variable *myAge* will store details of dialog. The dialog will ask the user for his/her age. *studentAge* will contain the value entered by user. But here’s the catch. I have asked AppleScript to convert the text to number. As I don’t want age to be a String.

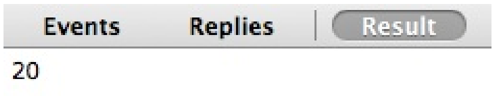

What is your age?

20



```
"20"
```

Figure 7.3.1 Without Coercion



```
20
```

Figure 7.3.1-2 With Coercion

Records

A record is a list of properties. You can retrieve items from a record by name, but not by index.

Declaring Records

Script [8.1.1]: `set propertyName to {myAge:20}`

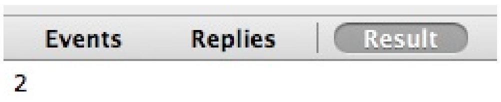
Explanation: *propertyName* is the name of record. And it contains the property *myAge*. Make sure the properties inside record cannot have space that is, we cannot write *myAge* as my Age.

Count

Script [8.2.1]:

```
set studentData to {myName:"Nayan Seth", myAge:20}
set recordSize to count of studentData
```

Explanation: The above Script creates a record named *studentData*. *recordSize* contains total number of properties present in studentData. This is done using the keyword count.



```
2
```

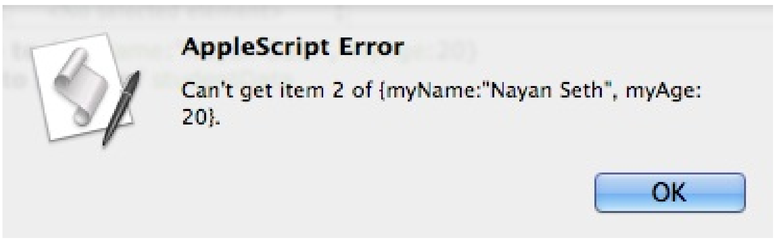
Figure 8.2.1 Count of Properties

Extracting Information From Record

Script [8.3.1]:

```
set studentData to {myName:"Nayan Seth", myAge:20}
set studentAge to item 2 of studentData
```

Explanation: The first line of chapter says that we can retrieve items from record by name and not by index. This is important because the 2nd statement will give an error. As we cannot retrieve items by index.


AppleScript Error

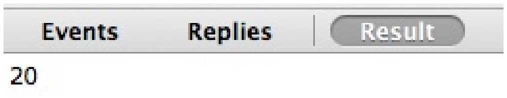
Can't get item 2 of {myName:"Nayan Seth", myAge: 20}.

Figure 8.3.1 Cannot Retrieve By Index

Script [8.3.2]:

```
set studentData to {myName:"Nayan Seth", myAge:20}
set studentAge to myAge of studentData
```

Explanation: Here a record named *studentData* is created. And variable *studentAge* retrieves age from *studentData*.



```
20
```

Figure 8.3.2 Retrieve Items from Record

Import Records

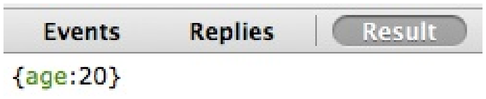
Script [8.4.1]:

```
set studentData to {myName:"Nayan Seth", myAge:20}
set studentAge to myAge of studentData
set newStudentData to {age:studentAge}
```

Explanation: The above Script will import data from one record to another via an intermediate variable. So how does it work?

Well I have first created a record named *studentData*. I then went on to create a variable named *studentAge* which contains value of *myAge* used in *studentData*.

Finally I created a new record named *newStudentData* which contains a property named age. This property imports value from the variable *studentAge*.



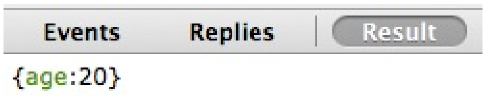
```
{age:20}
```

Figure 8.4.1 Import Value from Record

Script [8.4.2]:

```
set studentData to {myName:"Nayan Seth", myAge:20}
set newStudentData to {age:myAge of studentData}
```

Explanation: In this case we don’t use an intermediate variable. We directly import value of *myAge* in *studentData* to *newStudentData*.



```
{age:20}
```

Figure 8.4.2 Import Value from Record Directly

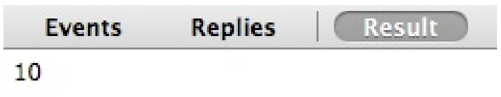
Copy Command

The copy command works in a different way for Records and Lists. Let’s take a simple example first.

Script [8.5.1]:

```
set x to 10
set copyX to x -- stores copy of x i.e. 10
set x to 100
get copyX -- returns copy of x
```

Explanation: Here I created a variable *x* which stores 10. I created a copy of x i.e. *copyX* which stores value of x that is 10. Now I assign x with a new value of 100. However when I use get command to print value of copyX, 10 is printed in result tab. This is because I stored a COPY of x to copyX.



```
10
```

Figure 8.5.1 Copy of a Variable

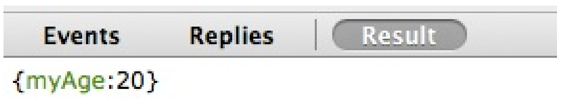
Let’s take a look at how this works in Records (and Lists).

Script [8.5.2]:

```
set studentData to {myAge:18}
set copyStudentData to studentData
set myAge of studentData to 20
get copyStudentData
```

Explanation: In this case, I have created a record named *studentData* with a property *myAge*. I also created a copy of record *studentData* and stored it in *copyStudentData*.

Then I changed the value of *myAge* in *studentData* to 20. Finally when I used the get command to print *copyStudentData*, the new value 20 is printed. This is because, when I use get command, the *copyStudentData* checks for the new data in *studentData*.



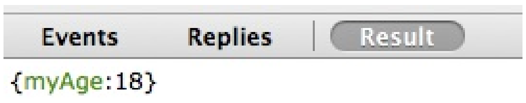
```
{myAge:20}
```

Figure 8.5.2 Trying to Copy Records

If 3rd statement of Script 8.5.2 was,

`set studentData to {myAge:20}`

then output would be 18 and not 20. This is because when we use the get command, the new changes in the properties are returned and not new properties.



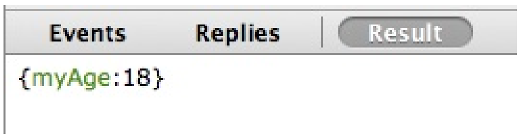
```
{myAge:18}
```

Figure 8.5.2-2 Copying Records

Script [8.5.3]:

```
set studentData to {myAge:18}
copy studentData to copyStudentData -- now a copy is stored
set myAge of studentData to 20
get copyStudentData
```

Explanation: Here I created a record named *studentData* and copied it to *copyStudentData* using copy command. On using copy command, a copy is created. So now if I change property values in original record, *copyStudentData* will still retain the values which were copied when copy command was used.



```
{myAge:18}
```

Figure 8.5.3 Copy Command

Copy command plays a vital role in script, when you try to debug the script. Because if we use the *set* command in lists and records, it will not store the copy. Rather it will store the latest changes.

Easier Dialogs

In this chapter we will learn how to create dialogs with a few clicks and tweaks.

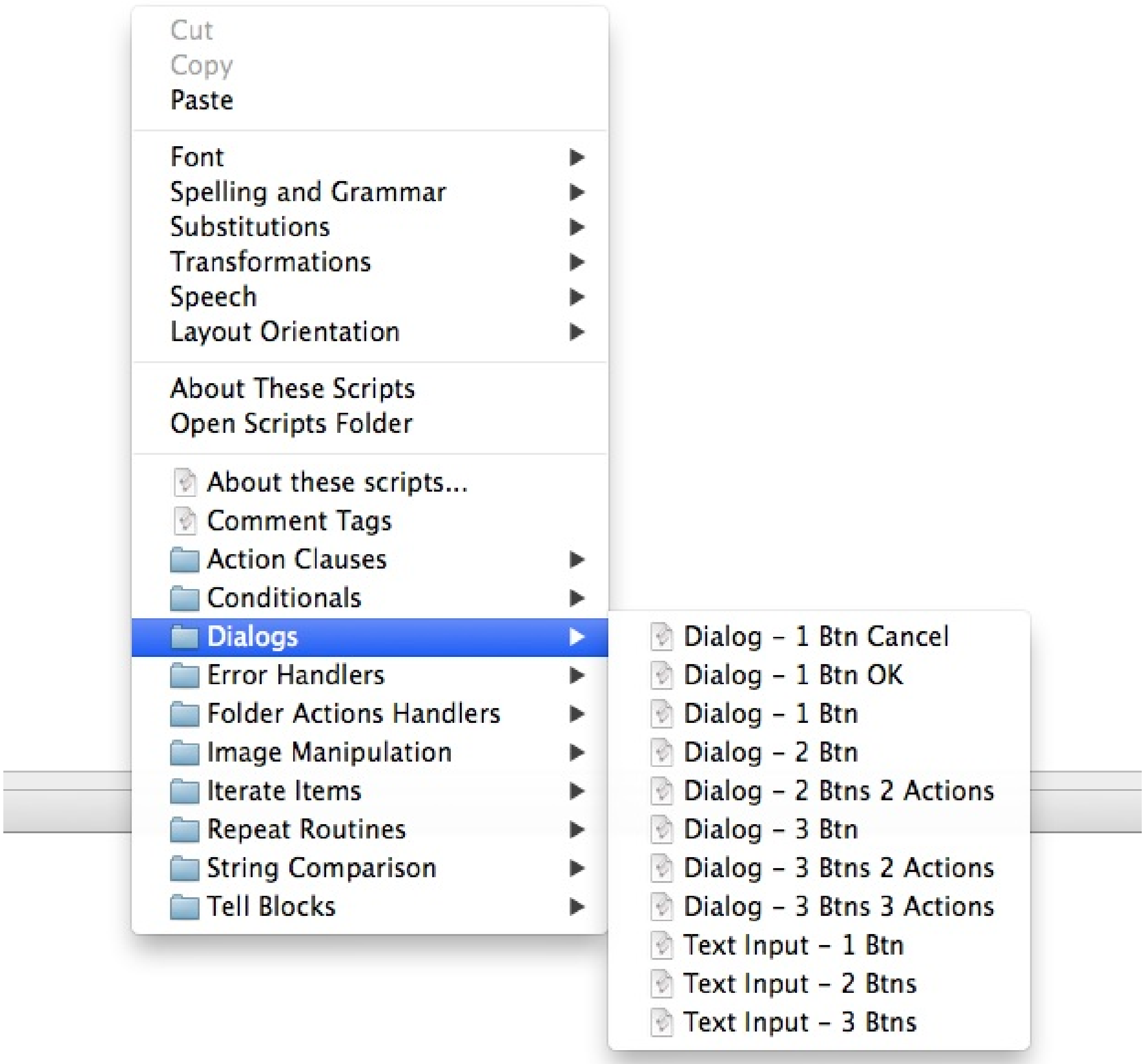


Figure 9.1 Creating Dialogs

Right Click in your AppleScript window, select Dialogs. A number of options are provided. You can have a Dialog with Buttons, Actions or even Text Input.

Conditional Statements

Life is full of if and else. If this then that. Programming and Scripting allow us to add in our own if-else blocks. But here we decide the conditions. Before proceeding with the if...else blocks, let's go through the relation operators which are at our disposal.

Comparison

Relation Operators for Comparisons

Serial No	Programming Language	AppleScript
1	=	is / is equal to
2	>	greater than
3	<	less than
4	>=	greater than or equal to
5	<=	less than or equal to

In AppleScript, we can compare numbers and strings directly. If they are equal the result tab will return true else it will return false.

Script [10.1.1]:

```
30 = 30
"nayan" = "nayan"
```

Explanation: As both the comparisons in the Script section are equal, the result tab will print true. Remember that AppleScript prints result of last executed statement only.

Based on whether the output for particular operator is true or false, we can use the if...else statements.

if...else

Relation Operators for Strings

Serial No	AppleScript
1	begins with / starts with
2	ends with
3	is equal to
4	comes before
5	is in
6	contains

These operators can be negated too. Like for example, *does not contain*, *does not begin with*, is not in, etc.

Script [10.2.1]:

```
if true then
    -- insert actions here
end if
if false then
    -- insert actions here
end if
```

Explanation: If the operation performed returns true then it will execute certain set commands.

If the operation performed returns false then it will execute certain set commands. The if command ends with *end if*.

Script [10.2.2]:

```
set myAge to 20
if myAge is greater than or equal to 18 then
    beep
    display dialog "You are eligible to Vote"
    say "You are eligible to Vote" using "Zarvox"
end if
```

Explanation: This program is intended to check whether you are eligible to vote or not. Here I created a variable named *myAge*. It stores my age.

Then to check my eligibility, I created a if condition. But in order to execute the commands inside the if condition, my comparison should be true that is *myAge*>=18. If comparison is true, the a beep sound will be made and a dialog will be displayed saying you are eligible to vote. Finally Zarvox will say that you eligible to vote.

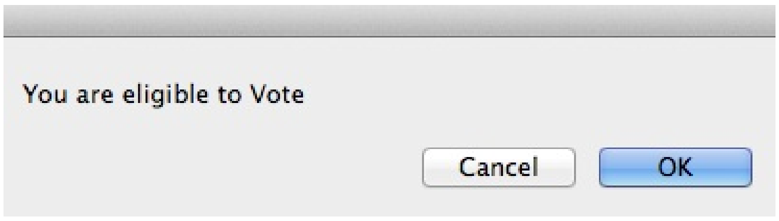

You are eligible to Vote

Figure 10.2.2 if Condition

Script [10.2.3]:

```
set myAge to 17
if myAge is greater than or equal to 18 then
    beep
    display dialog "You are eligible to Vote"
    say "You are eligible to Vote" using "Zarvox"
else
    display dialog "You are not eligible to Vote"
end if
```

Explanation: This program is intended to check whether you are eligible to vote or not. However I have added an else condition too. If the age comparison is false then script will execute the else section.

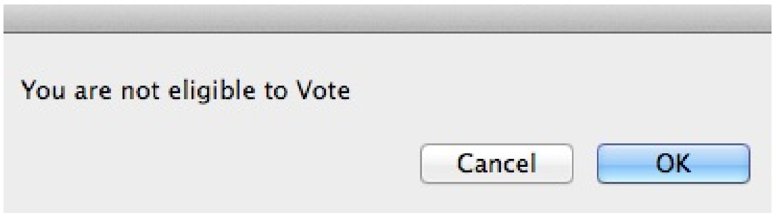

You are not eligible to Vote

Figure 10.2.3 if...else Condition

Script [10.2.4]:

```
set myName to "Nayan Seth"
if myName is equal to "Nayan Seth" then
    display dialog "True"
end if
--    Starts With
if myName starts with "Nayan" then
    beep
end if
```

Explanation: In the above Script we are comparing strings and then using if conditions. So I have created a variable named *myName*. It stores my name.

In the first if condition I am checking if *myName* is equal to “Nayan Seth”. Since the comparison is true, a dialog will display True.

In second if condition, I am checking if *myName* begins with “Nayan”. As this comparison holds true too. So a beep sound will be made.

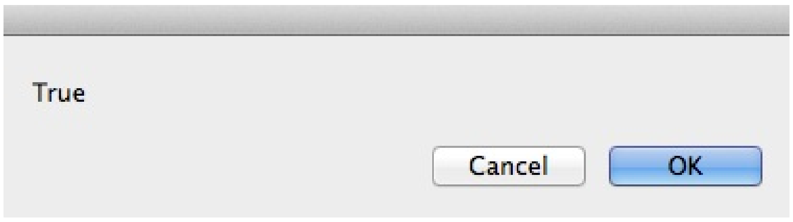

True

Figure 10.2.4 if Condition for Strings

Script [10.2.5]:

```
if "Nayan" comes before "Seth" then
    beep
end if
```

Explanation: comes before or comes after works alphabetically. Here I am checking if Nayan comes before Seth. As “N” (first letter of Nayan) comes before “S” (first letter of Seth), the condition holds true and a beep sound will be made.

Script [10.2.6]:

```
set singleCharacter to "s"
set myName to "Nayan Seth"
considering case
    if myName contains singleCharacter then
        beep
    else
        display dialog "Does not contain " & singleCharacter
    end if
end considering
```

Explanation: It may so happen that you want to make case sensitive comparisons. It's simple, use considering case command.

In the above example I am checking if singleCharacter is present in *myName*. Do note singleCharacter is "s" and *myName* contains "S". So condition holds false and so the else section will be executed that is a dialog will be displayed.

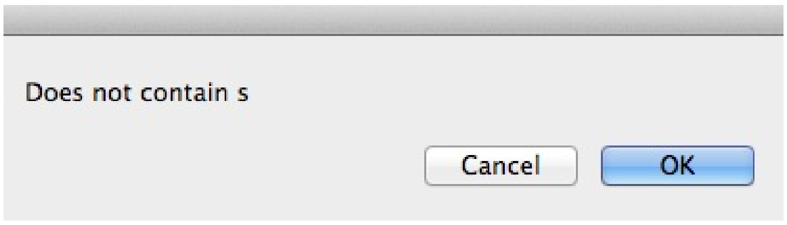

Does not contain s

Figure 10.2.6 Case Sensitive

Script [10.2.7]:

```
set fullName to "Naya n Seth"
set myName to "Nayan Seth"
ignoring white space
    if myName is equal to fullName then
        display dialog "Success"
    else
        beep
    end if
end ignoring
```

Explanation: What if you want to compare strings but ignore white space (it means space between two characters)? Then you can make use of ignoring white space command.

Here I have two strings fullName & myName that contain "Naya n Seth" and "Nayan Seth" respectively. I have ignored white space and I am checking if they are equal. So the comparison will be done on "NayanSeth" and "NayanSeth". Since both are equal, if condition will be executed that is displaying a dialog.

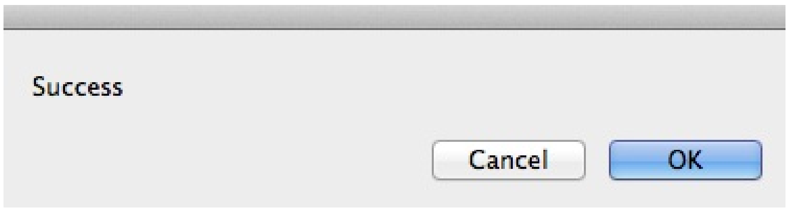

Success

Figure 10.2.7 Ignoring White Space

Conditions In Lists

Relation Operators for Lists

Serial No	AppleScript
1	begins with
2	ends with
3	contains
4	is equal to
5	is in

Script [10.3.1]:

```
display dialog "My Name is Nayan Seth" buttons {"Cancel", "No", "Yes"} default button 3
if the button returned of the result is "Yes" then
    say "That is true"
else if button returned of the result is "No" then
    say "You clicked the wrong button"
end if
```

Explanation: In this example I am displaying a dialog with 3 buttons. Depending on the button you click, the if...else condition will be checked.

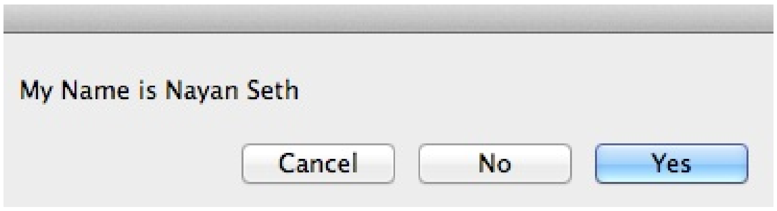

My Name is Nayan Seth

Figure 10.3.1 Dialog Button (if...else)

Script [10.3.2]:

```
set temp to display dialog "My Name is Nayan Seth" buttons {"Cancel", "No", "Yes"} default button 3
set buttonName to button returned of temp
if the buttonName is equal to "Yes" then
    say "That is true"
else if buttonName is equal to "No" then
    say "You clicked the wrong button"
end if
```

Explanation: In this example I am displaying a dialog with 3 buttons. Depending on the button you click, the if...else condition will be checked.

Here instead of directly using AppleScript commands, I am making use of variables. And I have used these variables to check if the button returned was “Yes”, “No” or “Cancel”

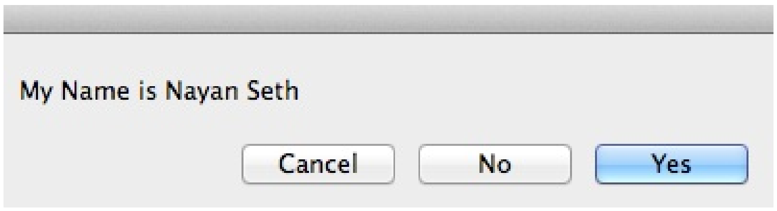

My Name is Nayan Seth

Figure 10.3.2 Dialog Button (if...else)

Dialogs with buttons and actions can be directly created.

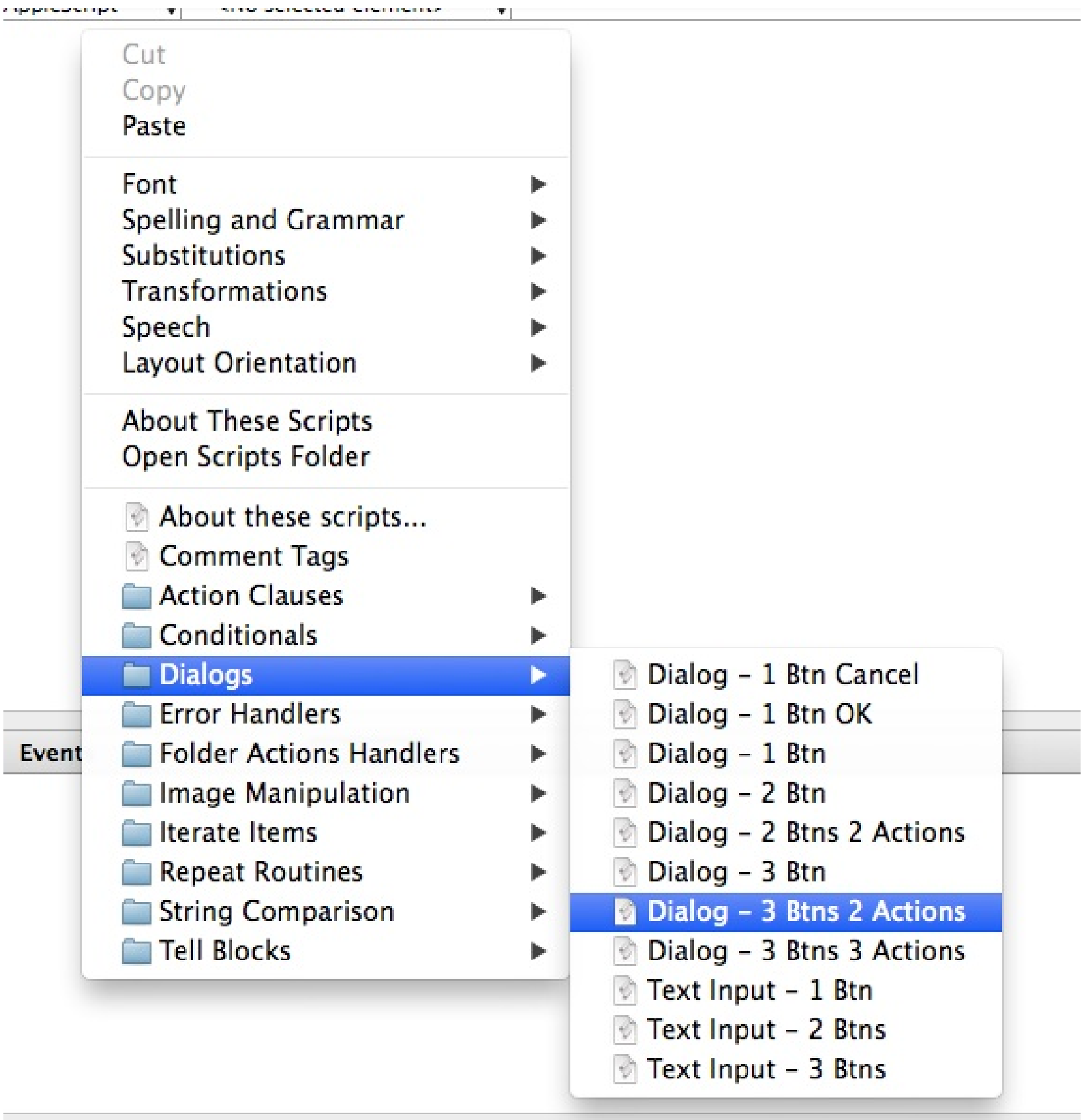


Figure: Simplifying Scripting

Conditions & Records

Relation Operators for Records

Serial No	AppleScript
1	contains
2	is equal to

Script [10.4.1]:

```
set studentData to {myName:"Nayan", myAge:20}
if myName of studentData is equal to "Nayan" then
    display dialog "Success"
else
    beep
end if
```

Explanation: Now its time to use if...else condition on Records. I have created a record named *studentData*. And in if condition I am checking if the property myName is equal to “Nayan”.

This condition holds true and hence if section will get executed.

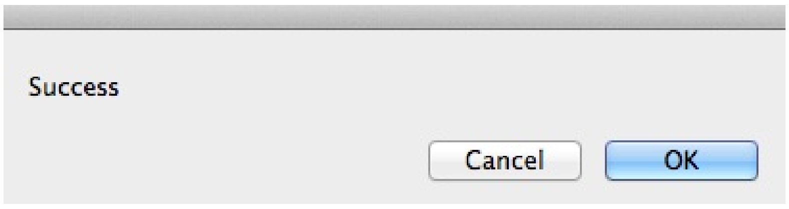

Success

Figure 10.4.1 Records (if...else)

Logical Conditions

Logical conditions that we will look at will be AND & OR. This section reminds me of Electronics and first year of Engineering.

Script [10.5.1]:

```
set x to true
set y to true
if x and y then
    display dialog "True"
else
    display dialog "False"
end if
```

Explanation: For AND to hold true, both inputs should be true. In above Script both x and y are true and hence if section will get executed.

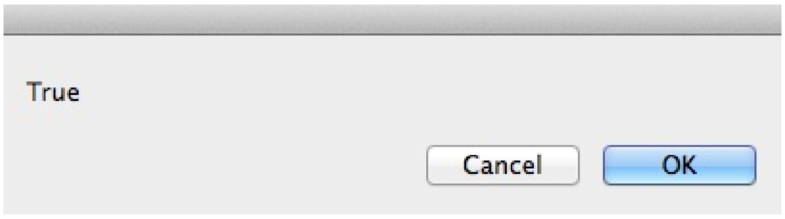

True

Figure 10.5.1 AND Condition

Script [10.5.2]:

```
set x to true
set y to false
if x or y then
    display dialog "True"
else
    display dialog "False"
end if
```

Explanation: For OR to hold true, either one of the inputs should be true. In above Script, x is true whereas y is fals. However the if condition is met, hence if section will get executed.

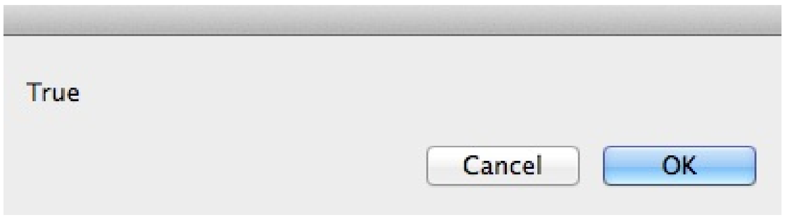

True

Figure 10.5.2 OR Condition

Script [10.5.3]:

```
set x to true
set y to false
set z to (x and y) -- z is false
set p to (x or y) -- p is true
```

Explanation: Here x is true and y is false. I declared variable z and assigned it with value of AND operation performed on x and y. p is assigned with value of OR operation performed on x and y. Hence z is false and p is true.

Script [10.5.4]:

```
set x to true
set y to "xyz"
if x and y = "xyz" then
    beep
else
    say "False Condition"
end if
```

Explanation: Here *x* is a boolean type and *y* is String type. In if condition I have used AND command where one of the variable is true. However we need to check for y variable. So I am checking contents of y variable with *xyz*. If they match then the other variable will return true. And since both return true the if section will get executed.

Try Catch Exceptions

Exceptions are unwanted errors that occur during run time. They can be avoided using try command.

try

Let’s take a simple example to understand use of try first.

Script [11.1.1]:

```
beep
set x to 1 / 0
say "I cannot speak this"
```

Explanation: A beep will be played first. But an error will occur on creating variable x because 1/0 is not defined. Because of this, all the subsequent statements will not be executed.

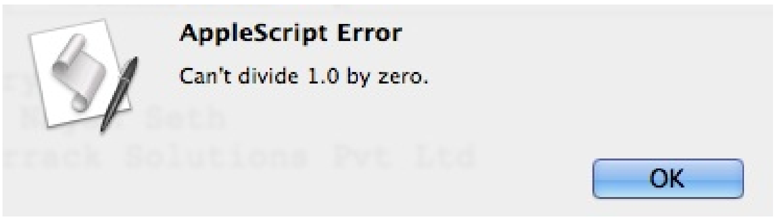

AppleScript Error
Can't divide 1.0 by zero.

Figure 11.1.1 Exception

However this can be avoided by using the try command.

Script [11.1.2]:

```
try
    beep
    set x to 1 / 0
    say "I cannot speak this"
end try
say "Hey There"
```

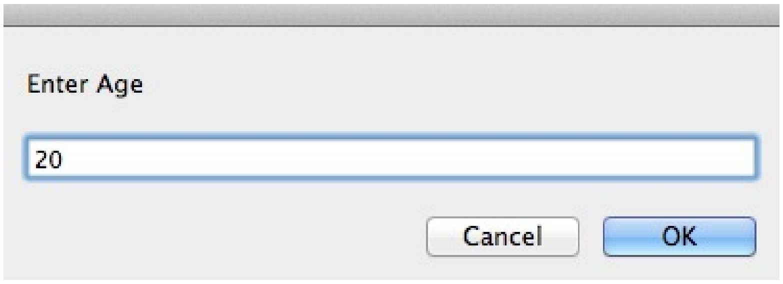
Explanation: Here with the help of *try* command, a beep sound will be made and then variable x will be created. But because 1/0 is not defined an error will occur. And the try block will look for actions to be performed *on error*. But since nothing is mentioned, the execution will proceed from the commands outside the try block.

Let’s take another example. Here we will take age as input from the user. However the script will fail if the input is not a number. To handle this we will use the *try* command

Script [11.1.3]:

```
set temp to display dialog "Enter Age" default answer ""
set myAge to the text returned of temp
try
    set myAge to myAge as number
    display dialog myAge
on error
    display dialog "Please enter a number"
end try
```

Explanation: In this example, I have asked user to enter his/her age. This input will be a String. However we need to make sure that the age is in number format. So I use a try block and perform coercion on *myAge*. If successful, then try block will display a dialog with *myAge*. If not successful then the *on error* commands will get executed.


Enter Age

20

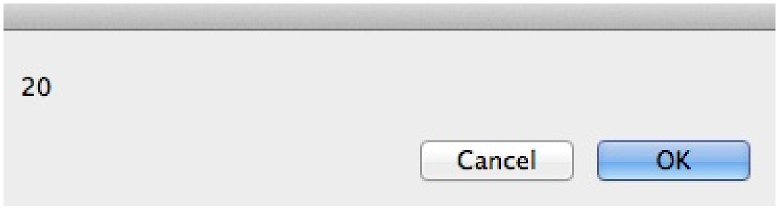

20

Figure 11.1.3 Number Exception Check

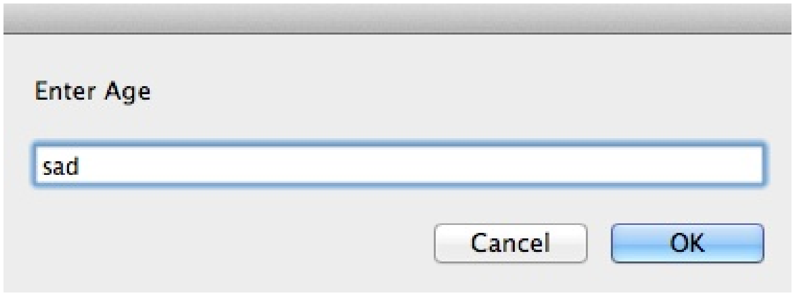
What if when error occurs, I want to print the Error Message and Error Number.

Script [11.1.4]:

```
set temp to display dialog "Enter Age" default answer ""
set myAge to the text returned of temp
try
    set myAge to myAge as number
    display dialog myAge
on error the errorMessage number the errorNumber
    display dialog "Error " & errorNumber & " : " & errorMessage
end try
```

Explanation: This is the same Script as 11.1.3. However there is minor change in the *on error* section. I have created variable *errorMessage* and *errorNumber* which store error message and error number respectively.

Note the declaration of the variables errorMessage and errorNumber.


Enter Age

sad

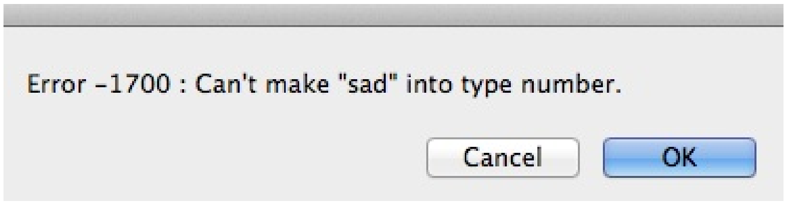

Error -1700 : Can't make "sad" into type number.

Figure 11.1.4 Error Message & Number

Files & Folders

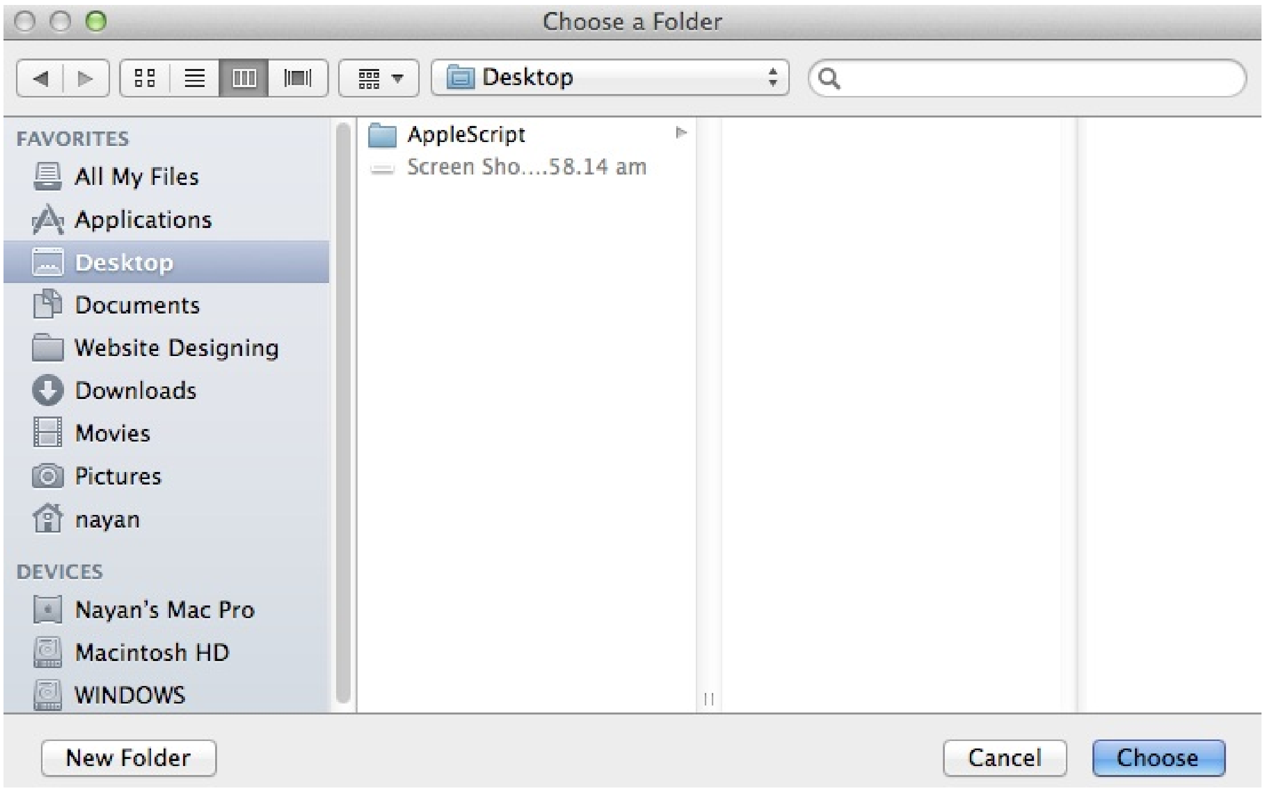
Files and Folders is something which we deal with everyday. A file manager for example, is used to manage your files but what if, you want to access a file or a folder using AppleScript?

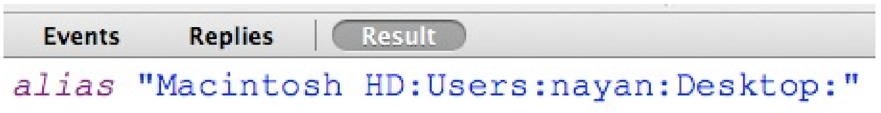
Folders

Let’s take a simple example on how to select a folder.

Script [12.1.1]: `choose folder`

Explanation: A window will pop up allowing user to select the folder he wishes to select. The result tab will show path to the folder.





```
alias "Macintosh HD:Users:nayan:Desktop:"
```

Figure 12.1.1 Choosing a Folder

The folder path in result tab is shown as follows:

hardDiskName:folderName:subFolder:

Opening Folders

Script [12.2.1]:

```
tell application "Finder"
    open "Macintosh HD:Users:nayan:Desktop"
end tell
```

Explanation: The above set of commands ask Finder to open Desktop folder. I have provided the exact location to the Desktop folder on my Mac Pro.

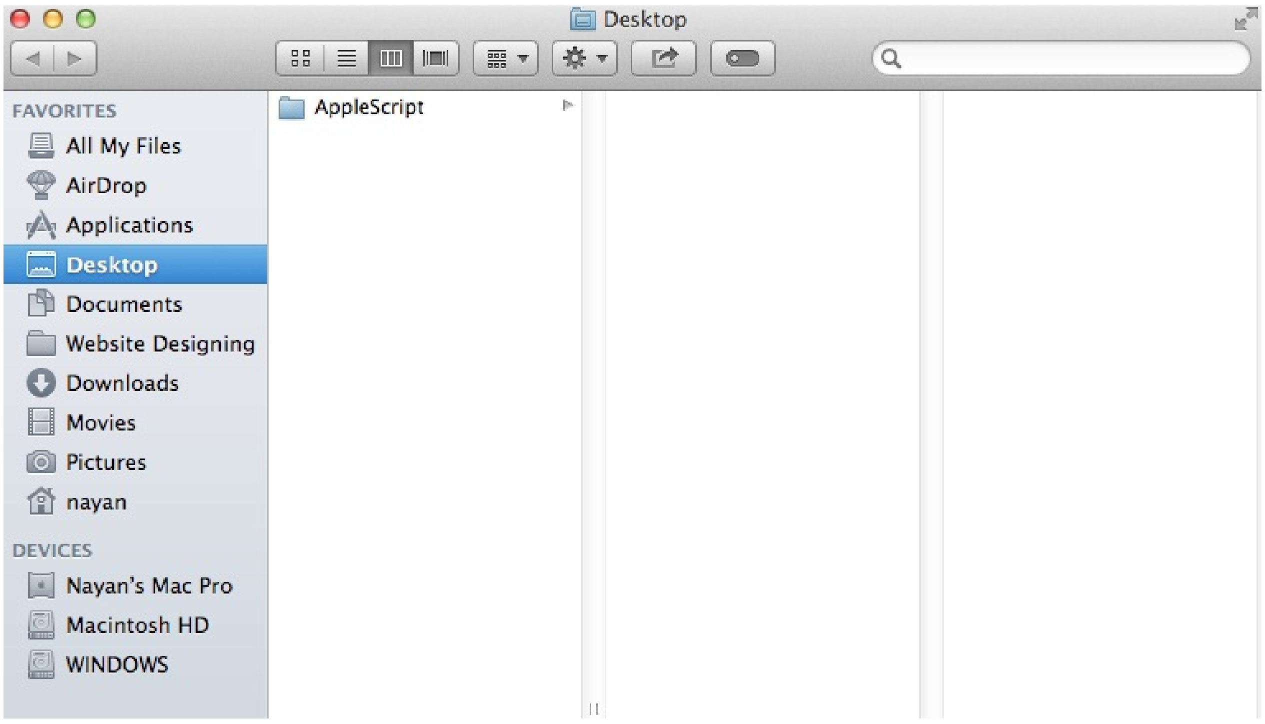
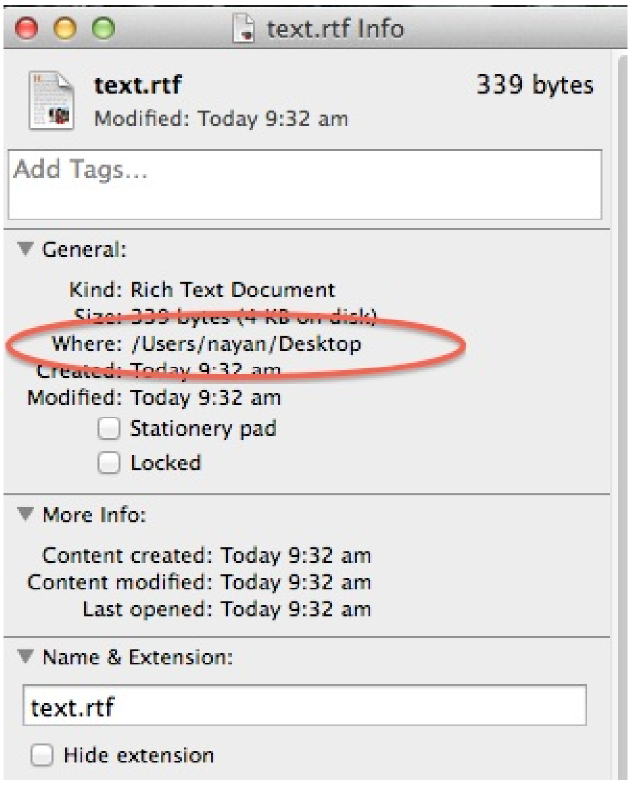


Figure 12.2.1 Open Folder

Opening Files


text.rtf Info

text.rtf 339 bytes
Modified: Today 9:32 am

Add Tags...

General:

Kind: Rich Text Document
Size: 339 bytes (4 KB on disk)
Where: /Users/nayan/Desktop
Created: Today 9:32 am
Modified: Today 9:32 am
Stationery pad
Locked

More Info:

Content created: Today 9:32 am
Content modified: Today 9:32 am
Last opened: Today 9:32 am

Name & Extension:

text.rtf

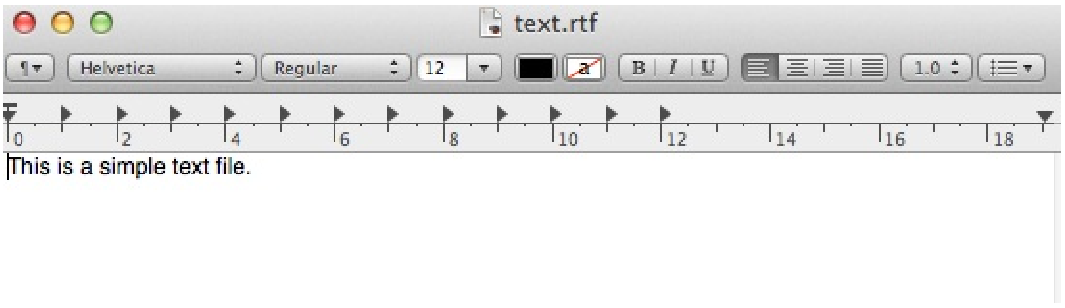
Hide extension

Figure 12.3.1 Path to File

Script [12.3.1]:

```
tell application "Finder"
    open "Macintosh HD:Users:nayan:Desktop:text.rtf"
end tell
```

Explanation: The above set of commands ask Finder to open file which is located in desktop.



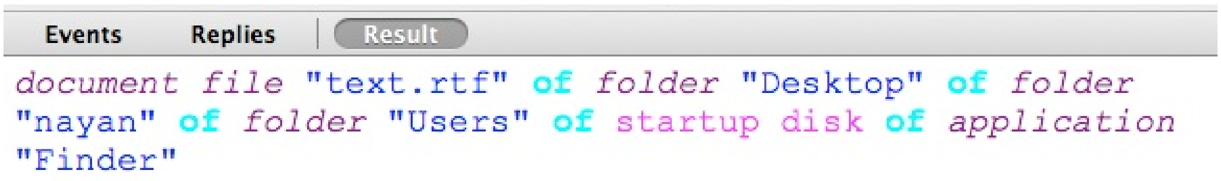
This is a simple text file.

Figure 12.3.1-2 Opening a File

Script [12.3.2]:

```
tell application "Finder"
    set thePath to file "Macintosh HD:Users:nayan:Desktop:text.rtf"
end tell
```

Explanation: Here Finder, sets the path of the file text.rtf to a variable named *thePath*. The output of path is awkward and can be understood by Finder only.



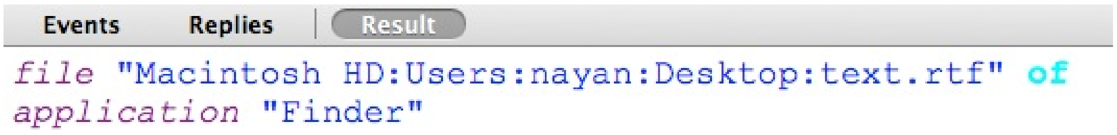
```
document file "text.rtf" of folder "Desktop" of folder
"nayan" of folder "Users" of startup disk of application
"Finder"
```

Figure 12.3.2 Path to File (Variable)

Script [12.3.3]:

```
tell application "Finder"
    set thePath to a reference to file "Macintosh HD:Users:nayan:Desktop:text.rtf"
end tell
```

Explanation: *thePath* will store reference to the file text.rtf which is located on Desktop. The output says file and not alias.



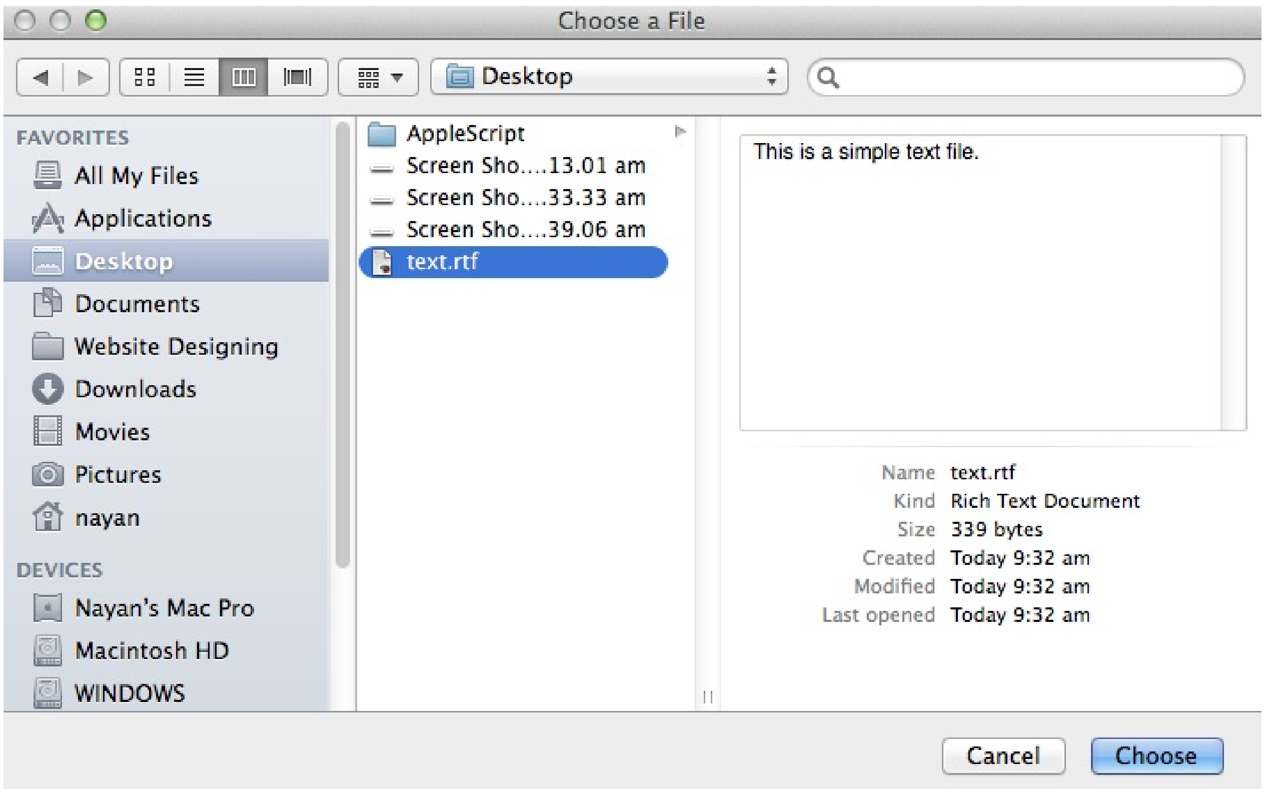
```
file "Macintosh HD:Users:nayan:Desktop:text.rtf" of
application "Finder"
```

Figure 12.3.3 Reference to a File

if the Script was,

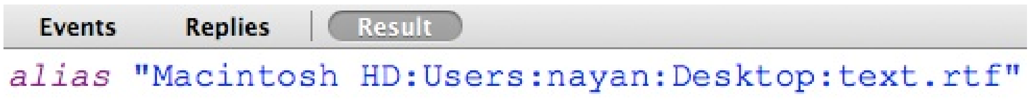
`choose file`

then output would be,



This is a simple text file.

Name	text.rtf
Kind	Rich Text Document
Size	339 bytes
Created	Today 9:32 am
Modified	Today 9:32 am
Last opened	Today 9:32 am



```
alias "Macintosh HD:Users:nayan:Desktop:text.rtf"
```

Figure: Choose a File

Explanation: Important thing to note here is the *alias*. So what's the big deal. Well consider a file in Downloads with a name *text.rtf*. Now you create an *alias* of *text.rtf* on the Desktop.

When you open the *alias*, the file *text.rtf* opens. Let's move *text.rtf* to Documents. Now if you open the *alias* on Desktop, *text.rtf* will still open. How is that possible?

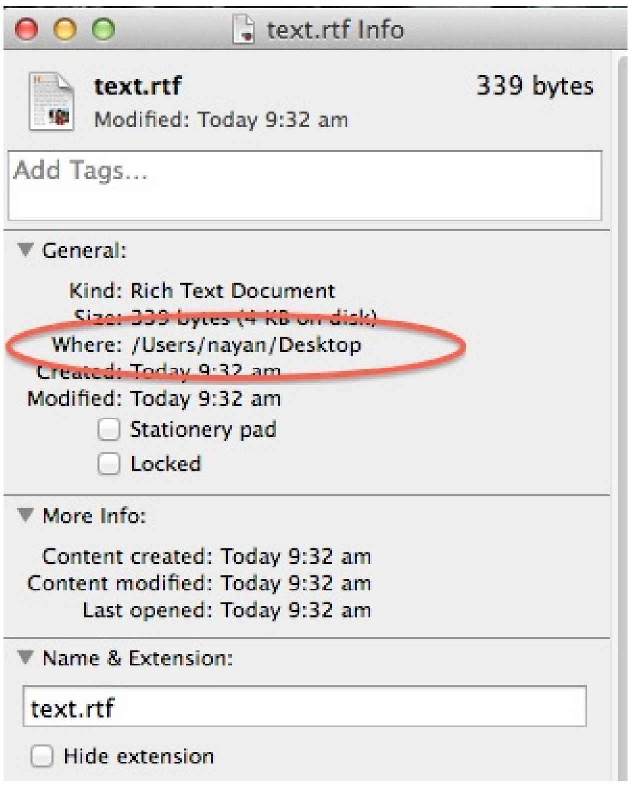
Every file on your Mac has a unique id which Finder maintains. The *alias* does not store the path. It stores the ID. The Finder updates its database with the new path and points it to the File ID.

So when we double click the *alias*, we are asking Finder to open file with this id.

Script [12.3.4]:

```
tell application "Finder"
    move file "Macintosh HD:Users:nayan:Desktop:text.rtf" to "Macintosh HD:Users:nayan:Downloads:"
end tell
```

Explanation: Here Finder, will move text.rtf from Desktop to Downloads folder.



text.rtf Info

text.rtf 339 bytes
Modified: Today 9:32 am

Add Tags...

General:

Kind: Rich Text Document
Size: 339 bytes (4 KB on disk)
Where: /Users/nayan/Desktop
Created: Today 9:32 am
Modified: Today 9:32 am
Stationery pad
Locked

More Info:

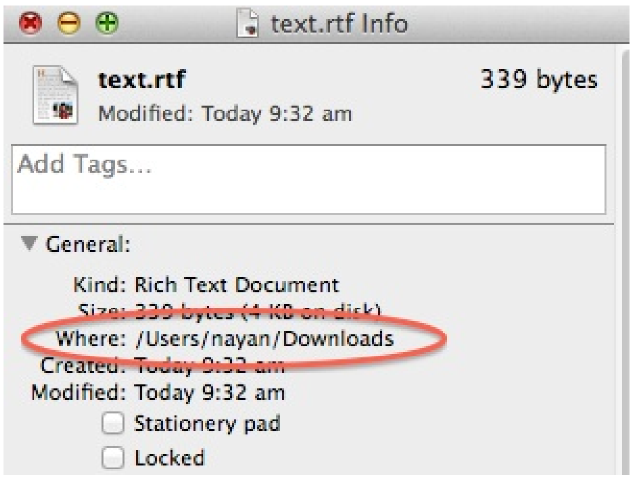
Content created: Today 9:32 am
Content modified: Today 9:32 am
Last opened: Today 9:32 am

Name & Extension:

text.rtf

Hide extension

Figure 12.3.4 Original File Location



text.rtf Info

text.rtf 339 bytes
Modified: Today 9:32 am

Add Tags...

General:

Kind: Rich Text Document
Size: 339 bytes (4 KB on disk)
Where: /Users/nayan/Downloads
Created: Today 9:32 am
Modified: Today 9:32 am
Stationery pad
Locked

Figure 12.3.4-2 Move Files

Script [12.3.5]:

```
tell application "Finder"
    move file "Macintosh HD:Users:nayan:Desktop:text.rtf" to trash
end tell
```

Explanation: Here Finder, will move text.rtf from Desktop to trash.

Script [12.3.6]:

```
set thePath to alias "Macintosh HD:Users:nayan:Desktop:text.rtf"
tell application "Finder"
    move file "Macintosh HD:Users:nayan:Desktop:text.rtf" to "Macintosh HD:Users:nayan:Downloads:"
    open thePath
end tell
```

Explanation: In this case I have created an *alias* of *text.rtf* and stored it in *thePath*. Then I asked Finder to move *text.rtf* from Desktop to Downloads.

Finally I open the file *thePath*. But if you notice I just moved it. However the file will still open as *thePath* contains the ID not the path.

An important thing to note is that the *alias* command can be used without the tell application “Finder” block. Though Finder is responsible for indexing files on Mac, the command can be used without the tell block.

Loops

Loops allow us to repeat certain set of commands, ‘n’ number of times.

Creating A Loop

Script [13.1.1]:

```
repeat 2 times
    say "This is a loop"
end repeat
```

Explanation: Loops can be created using the *repeat* command. Here I have created a loop which will run 2 times. And within the *repeat* block, I have specified the commands to be executed.

As per the above Script, the statement “This is a loop” will be spoken twice.

Script [13.1.2]:

```
set i to 2
repeat i times
    -- commands to be repeated
end repeat
```

Explanation: Loops can be created using variables too. Here I have assigned variable *i* to 2 and I have asked repeat block to repeat *i* times. I chose variable *i*, because in programming languages, we usually use *i* in loops.

Amalgamating Learning

Script [13.2.1]:

```
set temp to display dialog "Enter integer" default answer ""
set i to text returned of temp
try
    set i to i as integer
end try
if class of i is integer then
    repeat i times
        say "Hey this works"
    end repeat
else
    display dialog "Please enter integer"
end if
```

Explanation: This program is meant for understanding exactly how we can use if...else, repeat and try together.

The aim of the script is to ask for an integer input from user and repeat “Hey this works”, ‘n’ number of times, where n is the integer input.

I began with a dialog where I asked user to provide integer input. I copied the user input to variable *i*. It is possible that user may enter a String. So I used try block where I perform coercion on variable *i*. Remember this works on *integer* and not on *number* class.

Then I use if condition to check if variable *i* belongs to class *integer*. If true then “Hey this works” will be repeated i times.

If condition is false then a dialog will be displayed asking user to enter an integer.

Exiting Loops

Script [13.3.1]:

```
set condition to false
repeat while condition is false
    -- commands
    -- make condition true to stop repeat
end repeat
```

Explanation: What if I have a condition which is either true or false? And I want the repeat section run as long as that condition holds true or false.

Then you can do that by using a variable. I have used variable condition and set it to false. Then I ask *repeat* block to execute till the condition is false.

In such *repeat* blocks, it is important to make change the condition, to stop the execution of *repeat* block.

Script [13.3.2]:

```
set condition to true
repeat until condition is false
    -- commands
    -- make condition false to stop repeat
end repeat
```

Explanation: This is another method of using *condition* based *repeat* blocks. The only difference is that, the *repeat* block will execute as long as a particular *condition* is not met. In the above example I have set *condition* to true. So the *repeat* block will execute as long as *condition* is not false.

e.g. **Script [13.3.3]**:

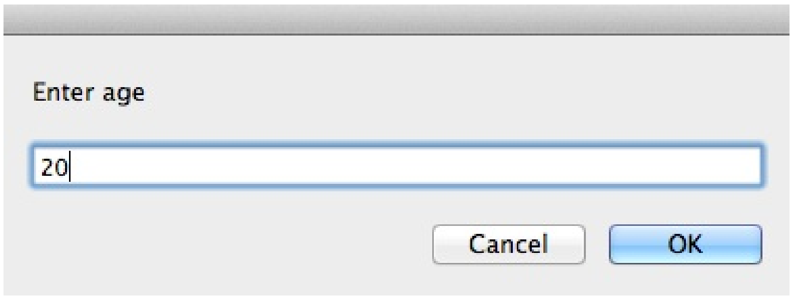
```
set condition to false
repeat until condition is true
    set temp to display dialog "Enter age" default answer ""
    set x to text returned of temp
    try
        set x to x as integer
    on error
        display dialog "Please enter a number"
    end try
    if class of x is integer then
        set condition to true
        if x is greater than or equal to 18 then
            say "Eligible to vote"
        else
            say "Not eligible to vote"
        end if
    end if
end repeat
```

Explanation: I have used *until* condition based loop. In the above example I have set *condition* variable as false. So as long *condition* is false, the *repeat* block will get executed.

I use variable *x* to take user input for age. Then I perform coercion. If coercion fails then the *on error* block will get invoked where a dialog will popup asking user to enter a number.

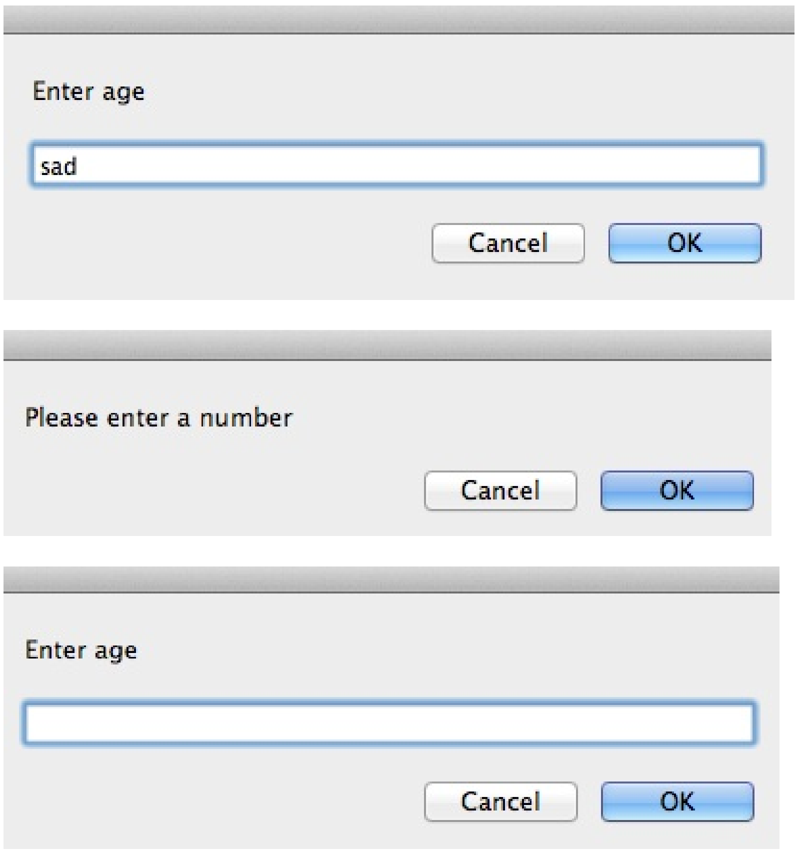
If coercion succeeds, then class of *x* will be checked. If class in integer, then *condition* will be set to true.

Next I checked whether *x>=18*. If it is >=18 then he will be eligible to vote, else he won’t be eligible to vote.


Enter age

20

Figure 13.3.3 Successful Run


Enter age

sad

Please enter a number

Enter age

Figure 13.3.3-2 On Failurer, User Has to Enter Age Again

Counters

It may so happen that you may want to add counter in your loops. So let's have a look at how we can implement counters in AppleScript.

Script [13.4.1]:

```
repeat with counter from 1 to 5
    say "I ran " & counter & " kilometers."
end repeat
```

Explanation: In the above *repeat* command, I have added a counter which will increment from 1 to 5 every time the *repeat* command is executed. The repeat section will execute as long as counter is not >5.

I ran counter kilometers will be spoken 5 times.

Script [13.4.2]:

```
repeat with counter from 1 to 5
    say "I ran " & counter & " kilometers."
    set counter to (counter + 1)
end repeat
```

Explanation: The output remans same. The sentence is spoken 5 times. You must be wondering, how is this possible even after incrementing counter.

Well, in **repeat** block the counter variables value cannot be changed.

Script [13.4.3]:

```
repeat with counter from 1 to 5 by 2
    say "I ran " & counter & " kilometers."
end repeat
```

Explanation: In 13.4.2, we saw that counter’s value cannot be changed inside the *repeat* block. However you can specify by *n* where n is an integer. by n, simply means incrementing by n.

In the above example, I am incrementing counter by 2 excluding the first time when it executes.

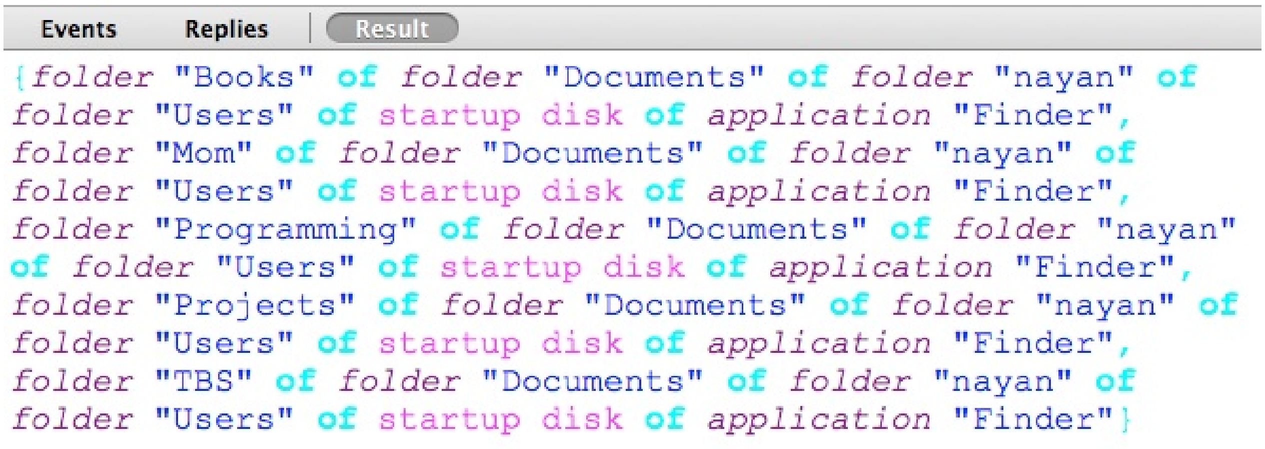
The sentence will be spoken 3 times that is counter = 1, 3 & 5.

Script [13.4.4]:

```
tell application "Finder"
    set folderPath to choose folder "Select Folder"
    set folderList to every folder of folderPath
end tell
```

Explanation: This is a simple script where I am creating a folder list of all folders present in the folder selected.

But if you remember, output was big and it was in a format that Finder understood. Let’s have a look at the output.



```
{folder "Books" of folder "Documents" of folder "nayan" of
folder "Users" of startup disk of application "Finder",
folder "Mom" of folder "Documents" of folder "nayan" of
folder "Users" of startup disk of application "Finder",
folder "Programming" of folder "Documents" of folder "nayan"
of folder "Users" of startup disk of application "Finder",
folder "Projects" of folder "Documents" of folder "nayan" of
folder "Users" of startup disk of application "Finder",
folder "TBS" of folder "Documents" of folder "nayan" of
folder "Users" of startup disk of application "Finder"}
```

Figure 13.4.4 Folder List

This is not something what user wants. He wants simple output with name of folders. How do we do it? It's simple, we use counters...

Script [13.4.5]:

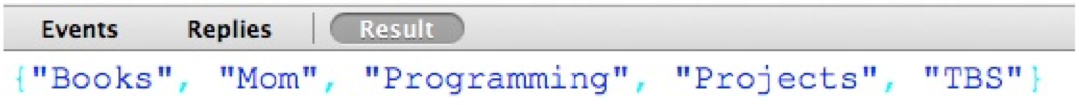
```
tell application "Finder"
    set folderPath to choose folder "Select Folder"
    set folderList to every folder of folderPath
end tell
set actualList to {}
repeat with counter in folderList
    set fName to name of counter
    set actualList to actualList & fName
end repeat
```

Explanation: This is a simple script where I am creating a folder list of all folders present in the folder selected. folderList will store the Finder formatted list of folders.

So I create another list named *actualList* and keep it empty. Then I use *repeat* command along with *counter* in the *folderList*. So every item of folderList can be called using counter.

Then I create a variable *fName (folder name)* which will store the name of item at index *counter (1, 2...)*. Then I append actualList with *actualList* and *fName*.

I append it with actualList also so that previous data is also included or else only the latest folder Name (*fName*) will be added.



```
{"Books", "Mom", "Programming", "Projects", "TBS"}
```

Figure 13.4.5 Folders in a Folder

Handlers

A handler is a routine/function/method which is specialized in a certain type of data or focused on certain special tasks.

Defining A Handler

Script [14.1.1]:

```
on method1()
    display dialog "This handler was executed"
end method1
```

Explanation: Just like in try block when we say *on error*, we are actually declaring a handler which will get executed when error occurs. Similarly to declare a handler we type *on handlerName()*. To end the handler we type *end handlerName*.

Calling A Handler

Script [14.2.1]:

```
on method1()
    display dialog "This handler was executed"
end method1
method1()
```

Explanation: In order to call a handler we just type the *handlerName*. And all the commands in the handler block will get executed.

Parametrized Handler

Script [14.3.1]:

```
on method1(input) -- parametrized handler
    display dialog input
end method1
method1("Hey There. I am Nayan Seth.") -- passing parameters value
```

Explanation: Parametrized Handlers are functions in which a variable is passed.

In the above example I have created a handler named *method1* and I have also passed variable input to the handler.

In the *method1* block, I have displayed a dialog with the text of variable input.

Now when I call *method1*, I have to pass the information of parameters inside the round brackets. When we run the script a dialog will pop up with text “Hey There. I am Nayan Seth.”

Script [14.3.2]:

```
on area(radius) -- parametrized handler
    set circleArea to pi * (radius ^ 2)
    display dialog "Area of Circle is " & circleArea
end area
set condition to false
repeat until condition is true
    set temp to display dialog "Enter radius of Circle" default answer ""
    set r to text returned of temp
    try
        set r to r as integer
    on error
        display dialog "Enter a valid number"
    end try
    if class of r is integer then
        set condition to true
        area(r)
    end if
end repeat
```

Explanation: Huge program! Don’t worry... It’s super simple.

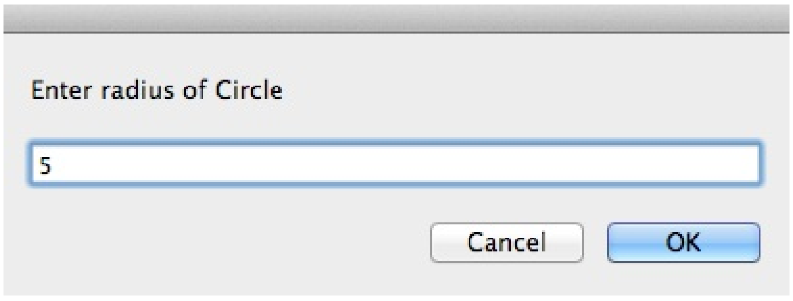
This program calculates area of circle by taking radius as input from user.

I have defined a handler named *area(radius)*. It computes area of circle and displays it in a dialog.

But since the handler is parametrized, it needs value for the radius. [Refer to 13.3.3] To get the value of the radius, I asked for user input. But we know that user can type a string too.

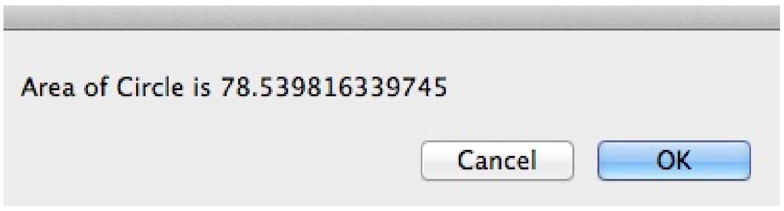
So I used repeat command so that if error occurs, user can re-enter the value of radius.

The value of radius that I get from user is stored in variable *r*. This value is passed to handler named *area(r)*.



Enter radius of Circle

5



```
Area of Circle is 78.539816339745
```

Figure 14.3.2 Area of Circle

Return Statement

What if I want a handler to compute some data and return it, rather than printing it in a dialog? It's possible.

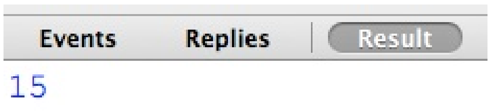
Script [14.4.1]:

```
on large(a, b) -- parametrized handler
    if a > b then
        return a -- returns a value
    else
        return b -- returns a value
    end if
end large
set largest to large(15, 10) -- largest stores return value of handler large
```

Explanation: I have a handler named *large(a,b)*. Its main objective is to find whether *a* is greater than *b* or b is greater than a.

However instead of displaying a dialog, I return a value. So whenever the handler is called, it gives a value which can be stored in a variable.

In above example I created a variable named *largest* which stores the output of *large(15,10)*.



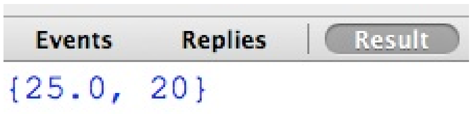
```
15
```

Figure 14.4.1 Return Value From Handler

Script [14.4.2]:

```
on square(s) -- parametrized handler
    set perimeter to 4 * s
    set area to s ^ 2
    return {area, perimeter} -- returns a list
end square
set squareList to square(5)
```

Explanation: This script is similar to 14.4.1. The only difference is that, the handler returns a list. And this list is stored in a variable named *squareList*.



```
{25.0, 20}
```

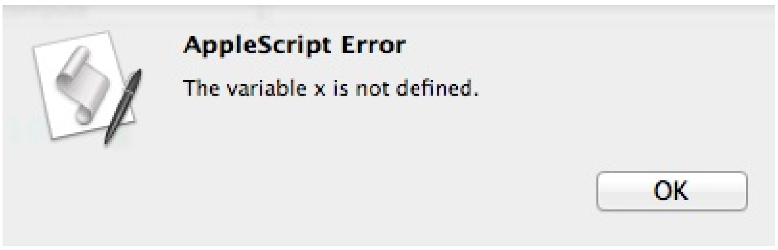
Figure 14.4.2 Return a List From Handler

Local Variables

Script [14.5.1]:

```
set x to 5
on compute()
    set y to 10 * x
end compute
compute()
```

Explanation: The above script will generate error on calling the handler named *compute()*. This is because variable *x* has been declared outside the handler *compute()* but we are using it inside. *compute()* does not know anything about variable *x*. Hence error gets generated.



AppleScript Error

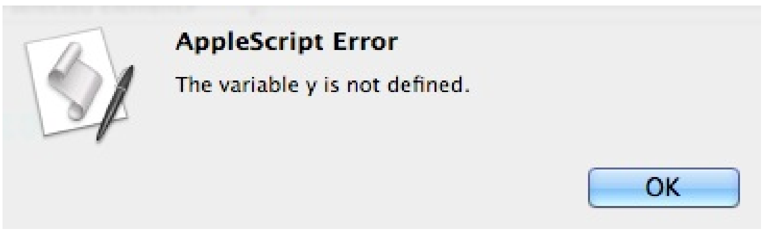
The variable x is not defined.

Figure 14.5.1 Local Variable

So anything declared inside the handler is local to the handler only i.e. if variable *x* is declared in the handler than the value of *x* will be local to the handler only. It cannot be called outside the handler.

e.g.

```
on compute()
    set y to 10
end compute
compute()
get y
```



AppleScript Error

The variable y is not defined.

Figure 14.5.1-2 Local Variables in Handler

Importing AppleScript File

In all programming languages we can import our own packages. What if you want to import your own script in another AppleScript file? Its' possible.

Script [14.6.1] (Syntax):

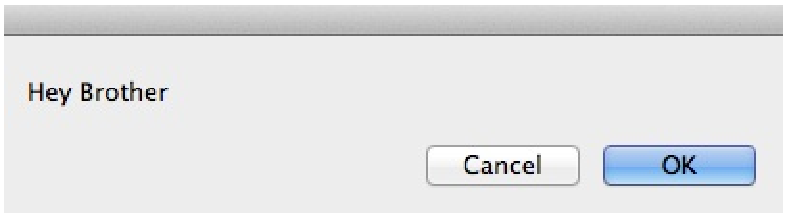
```
set scriptPath to (load script file "file Location")
tell scriptPath
    methodName()
end tell
```

Explanation: First we create a variable pointing to the file location of the AppleScript file that we want to load.

Then we can tell this variable to call the handlers from the AppleScript file we imported.

Script [14.6.2]:

```
set scriptPath to (load script file "Macintosh HD:Users:nayan:Desktop:AppleScript:14 - handlers.scpt")
tell scriptPath
    method2("Hey Brother")
end tell
```



Hey Brother

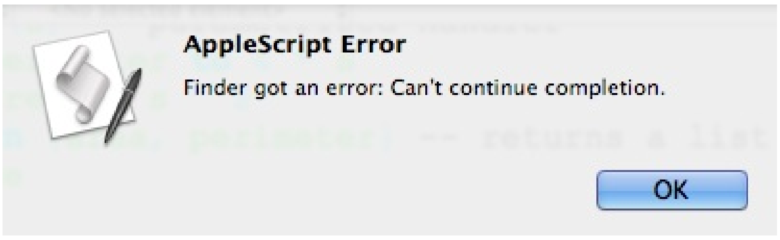
Figure 14.6.2 Calling Handlers from AppleScript File We Imported

Call Methods In Tell

Script [14.7.1]:

```
on completion()
    display dialog "Success"
end completion
tell application "Finder"
    empty trash
    completion()
end tell
```

Explanation: Here I have created a handler named *completion()*. But when I call it in the *tell* block, I get error.



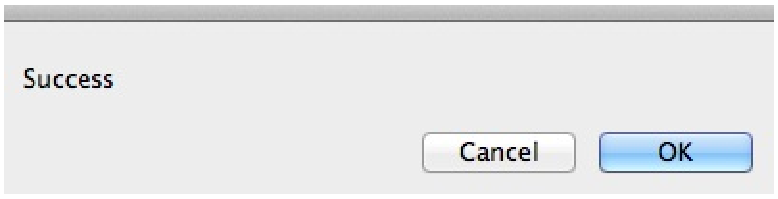
AppleScript Error

Finder got an error: Can't continue completion.

Figure 14.7.1 Error on Calling Handler in Tell

To solve this issue use command *of me* after calling handler

```
on completion()
    display dialog "Success"
end completion
tell application "Finder"
    empty trash
    completion() of me
end tell
```



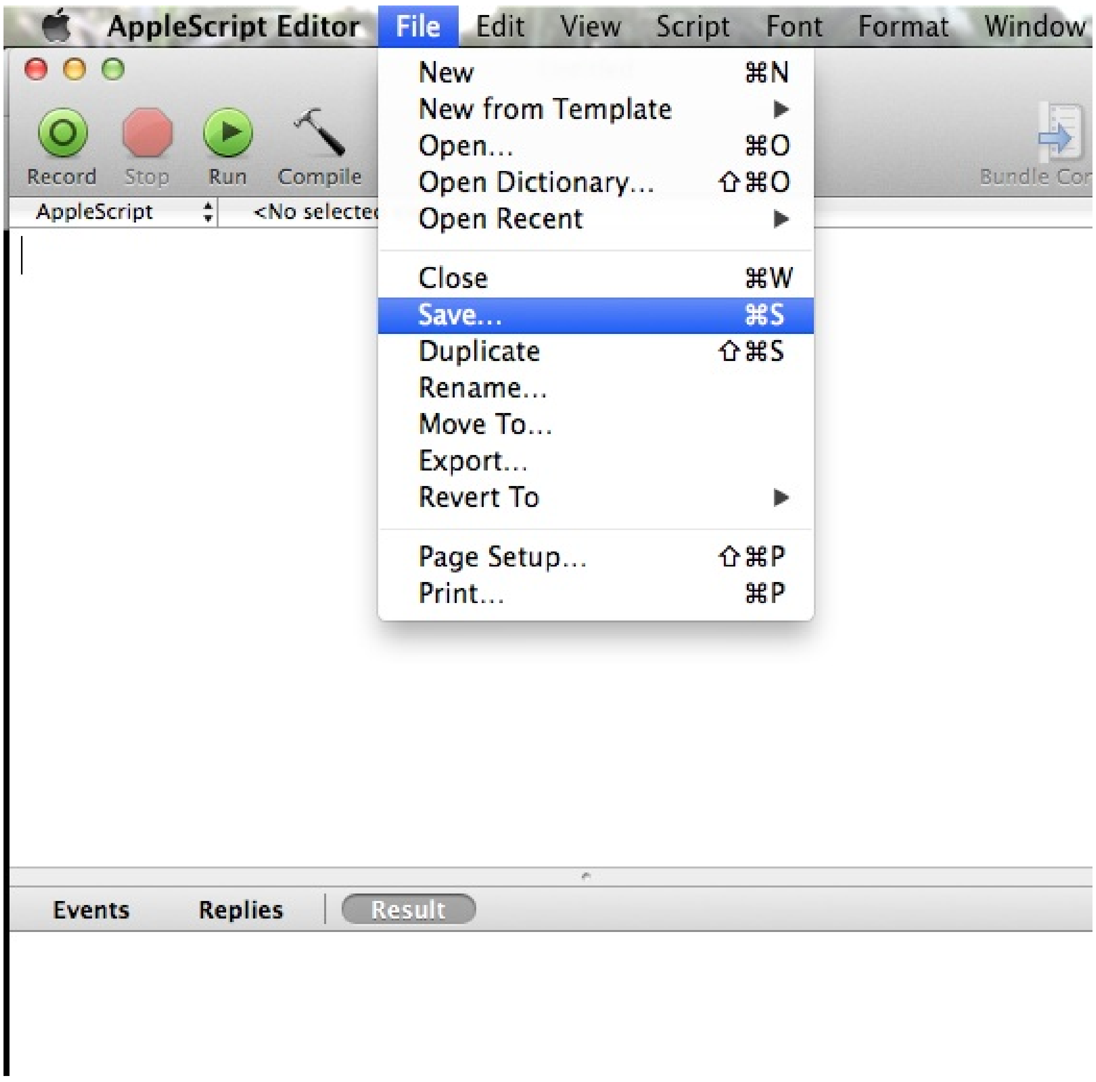
Success

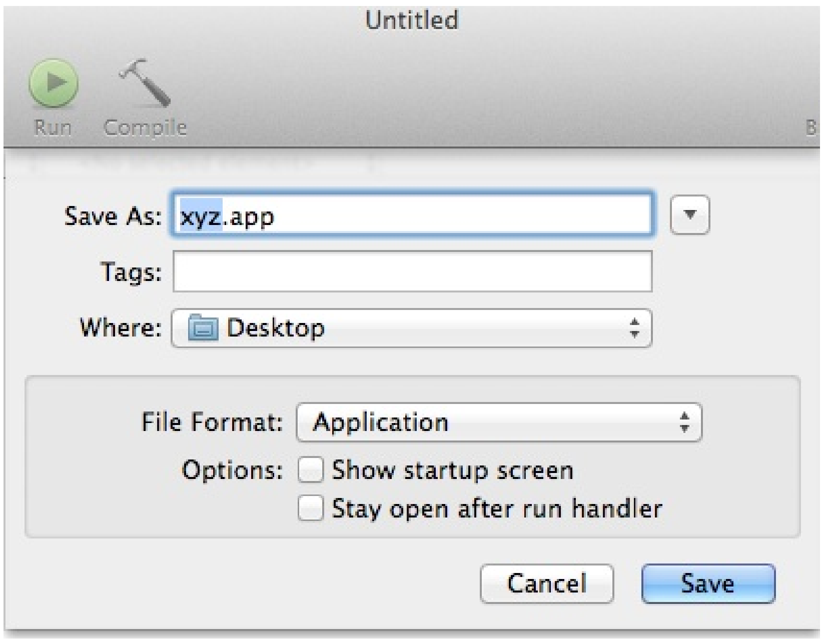
Figure 14.7.1-2 Success on Calling Handler In Tell

This issue of calling handler in tell blocks is not applicable to variables

Saving As Application

The best part of AppleScript is that you can save your script as an application.





Save As: xyz.app

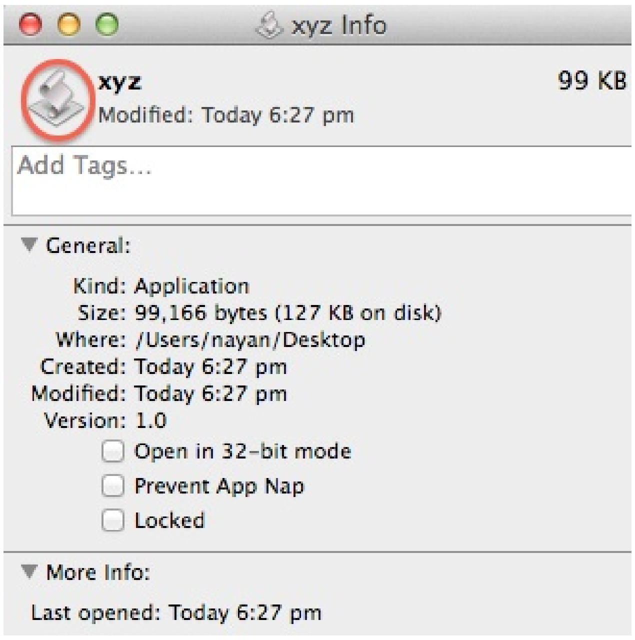
Tags:

Where: Desktop

File Format: Application

Options: ☐ Show startup screen

☐ Stay open after run handler


xyz Info

xyz 99 KB
Modified: Today 6:27 pm

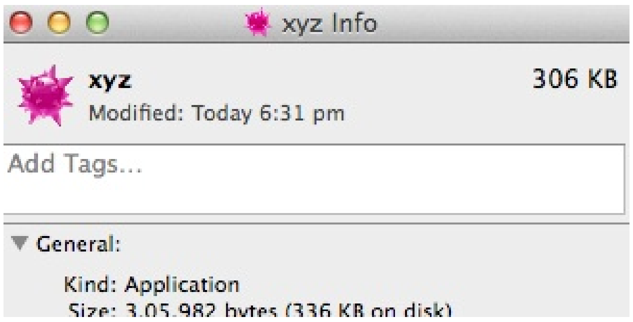
Add Tags…

General:

Kind: Application
Size: 99,166 bytes (127 KB on disk)
Where: /Users/nayan/Desktop
Created: Today 6:27 pm
Modified: Today 6:27 pm
Version: 1.0
☐ Open in 32-bit mode
☐ Prevent App Nap
☐ Locked

More Info:

Last opened: Today 6:27 pm


xyz Info

xyz 306 KB
Modified: Today 6:31 pm

Add Tags…

General:

Kind: Application
Size: 3,05,982 bytes (336 KB on disk)

Figure 15.1 Saving as an Application

To add logo to the application just drag the .icns file to the AppleScript logo in Get Info of the Application

I have made 2 apps on AppleScript, one is fake virus and other is quit.

Check out my apps at http://www.techbarrack.com/#downloads

Additional Guides

Before making any script, you can google about the script, as most of the scripts have been made by someone. So this gives you an heads up. These are some websites worth checking out:

1. http://www.macscripter.com
2. http://iworkautomation.com/index.html
3. http://macosxautomation.com
4. http://goo.gl/4QroIX (Official Apple Documentation)

I hope you enjoyed reading and learning!