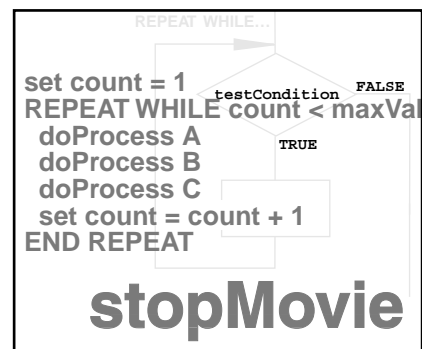
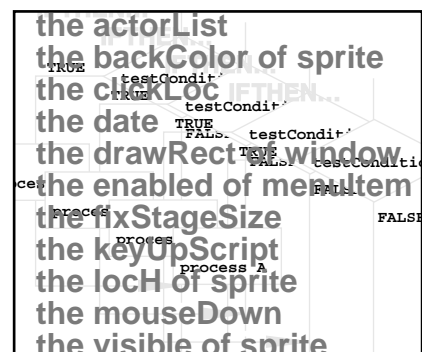
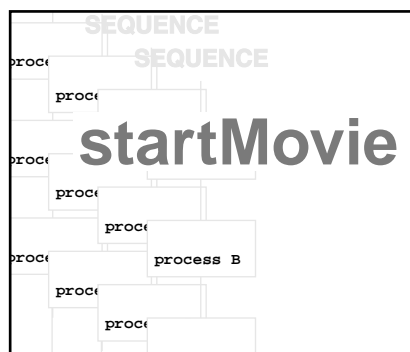


---

# Learning To Program in Lingo

---

This handbook and the accompanying digital material provide an introduction to using the Lingo programming language embedded in Macromedia™ Director™.



Author Ian Phillips DipAD, PGCE, MSc  
Limestone Cottage  
Pooles Lane  
Charlbury  
Oxfordshire OX7 3RT

Date April 1997

Edition 2.0

### **About this handbook**

This handbook provides an introduction to using the Lingo programming language embedded in Macromedia Director (all trademarks acknowledged). It should be used in conjunction with the accompanying Director movies to support classes and workshops in Lingo programming. This revision covers those changes in Lingo and the Lingo programming environment introduced with Director 5 which are of particular interest to the novice programmer.

### **About the author**

Ian Phillips was until recently Principal Lecturer in Computer-Aided Art and Design (60% FT) at Coventry University. He now works freelance as a consultant designer, developer and trainer specialising in learning materials.

He started teaching postgraduate students to use Director in 1993 and in 1994 wrote an AGOCC case-study comparing Director and HyperCard. He presented a paper and chaired a conference workshop on teaching programming to art and design students at CADE 95 and presented a paper on computer program visualisation in art and design computing at Eurographics UK in 1996.

He recently completed a prototype, which integrated a CD-ROM utilising Director movies with an Internet site, for a major publisher.

### **Acknowledgements**

Thanks to: all participants in the workshop on programming in art and design education at CADE 95; John Jostins for advice on Director project planning; Mark Peden for lots of help with Lingo; Simon Turner for occasional but valuable help; students on the postgraduate programme at Coventry School of Art and design for trying out the movies; Rosey Bennett for editorial input, and teachers such as John Vince, Simon Ritchie, Garry Bulmer and Alan Chantler who showed me that programming could be explained.

### **Feedback**

The author welcomes constructive criticism of these materials, which should be emailed to:

`iphillips@patrol.i-way.co.uk`

Fees for support and further training can be negotiated.

### **Copyright**

Copyright © 1996 and 1997 by Ian Phillips.

All rights reserved. This work may be copied in its entirety, without modification and with this statement attached, for non-commercial use in education. Redistribution in part or with modifications is not permitted without advance agreement from the copyright holder.

### **Print record**

This document was printed on 25 April 1997.

## Contents

---

<b>Introduction</b>	<b>1</b>
Why learn to program in Lingo? 1	
Objectives and intended outcomes 1	
Approach and methods 2	
Target audience 2	
How the project started 3	
Summary of changes from Edition 1.0 3	
Looking forward to Director 6.0 4	
Limitations of the materials 5	
<b>How to use the materials</b>	<b>6</b>
Assumed knowledge and equipment 6	
Production note 7	
Organisation of the materials 7	
Accessing the movies 7	
<b>Getting started with Lingo</b>	<b>9</b>
Determine your objectives 9	
Assess your knowledge and experience 9	
Set up your working environment 9	
Using the Message window 10	
Coping with the unknown 11	
Learning more about Lingo elements 12	
Using the Message window with cast and score 12	
Summary 13	
<b>Algorithm development</b>	<b>15</b>
What is an algorithm? 15	
Where do I start? 15	
Stepwise refinement of the algorithm 16	
Variables 19	
Controlling execution flow 24	
Summary 26	

<b>Basic scripting</b>	<b>27</b>
First things first	27
Learning the jargon	27
What's in this section?	28
Useful things to know	28
Controlling execution flow in scripts	29
Other sources of information	31
<b>Program visualisation</b>	<b>32</b>
What is program visualisation?	32
What methods are used in these materials?	32
Strengths and weaknesses of visualisation methods	33
Visualising data	39
Summary	39
<b>Fixing broken programs</b>	<b>40</b>
Why do programs break?	40
Tools for fixing programs	40
Summary	44
<b>Example movies</b>	<b>45</b>
What's in the movies?	45
Further work	45
<b>Annotated Bibliography</b>	<b>46</b>
On Lingo	46
On-line help and information	47
On programming in general	48
On object-oriented programming	49
<b>Appendix One - Update</b>	<b>50</b>
Summary of Lingo-related changes in Director 5	50
Understanding internal and external casts	60
Xtras	61

## Introduction

---

### Why learn to program in Lingo?

Director is a powerful, flexible and full-featured piece of software and its method of organising and integrating media components over time by direct manipulation in “cast” and “score” windows is fairly easy to understand. However, the control offered by the score is insufficient for many purposes. That is why the “scripting language” called Lingo, which offers more control, is included in the Director package.

Providing interactive capabilities and controlling the behaviour or properties of sprites on the stage often entails going behind the scenes and writing Lingo scripts. These are really small computer programs and although you can write simple scripts without knowing much about programming, you will soon get into difficulties if you don't acquire some basic understanding.

Learning to program computers is not easy. I explored this topic in more detail in *Computer Program Visualisation in Art and Design Computing* [PHILLIPS, 1996] but for the purposes of this handbook it is only necessary to note that, unless you can employ a professional programmer, there is currently no alternative to learning to program in Lingo if you wish to make full use of Director.

### Objectives and intended outcomes

This handbook and the associated Director movies were produced as an AGOCC-funded project with the following objectives:

- Establish the need to program in Lingo by showing examples of generic problems in user interface prototyping, control of media components, animation *etc.* that cannot be solved through use of the score alone.
- Illustrate the algorithmic approach to tasks.
- Explain the basic principles and practice of programming and provide examples of good programming style.

- Provide pointers to resources for the further exploration of computer programming in general and Lingo programming in particular.

The outcomes of using the materials are expected to include:

- An appreciation of the role of programming in developing multimedia software products.
- An understanding of how to approach the design of computer programs.
- The ability to develop modest algorithms and programs.
- Some ability to diagnose and correct faulty programs.
- The ability to work with programming specialists when entering employment, where teamwork will be the norm.

### **Approach and methods**

The approach has been to provide a background text, movies in various stages of development, and pointers to other resources with the aim of encouraging the learner to take off in pursuit of his/her own interests. Any one of the supplied movies could be taken in a number of directions: when combined with the sample movies and code fragments available from the other sources mentioned in this handbook, there should be more than enough material for an in-depth exploration of Lingo lasting several months.

The approach taken in this project differs from many programming tutorials in three key respects:

1. By providing a lengthy algorithm development example.
2. By exploring the visualisation of code and data.
3. By including extensive comments in movie scripts.

It is assumed that users of the materials will pull the movies apart and modify them, so they should serve both as learning vehicles and as sources for solutions to generic problems. Some examples of Lingo-based solutions to such problems are included in the movies in the *Example Movies* folder.

The underlying assumption is that it will be necessary to program for some time yet if we wish to utilise multimedia authoring tools to the full. Products and languages may change, so students now in the system should acquire something more generally useful than skills in using one product. These materials aim to ease the process of learning to program and, in so doing, help to develop the transferable skill of understanding the programming process irrespective of the language used.

### **Target audience**

The materials are aimed primarily at the UK art and design education community. However, they may also be suitable for use in any introductory course in Lingo programming.

### How the project started

As a postgraduate course tutor between 1991 and 1997, I saw plenty of novice programmers struggling with Lingo and with the whole concept of programming. I tried various ways of making their task easier and it was always obvious while doing this that one of the major problems facing novices was the lack of good tutorial material.

The supply of third-party Lingo materials has improved since publication of Edition 1.0 of *Learning to Program in Lingo* (see the Annotated Bibliography) but there is still a need for the approach taken here, hence the publication of Edition 2.0.

Edition 1.0 was approved in July 1995 and work started in September. The materials were delivered at the end of March 1996. Edition 2.0 was proposed in September 1996, approved in January 1997 and delivered in April.

With a colleague at Coventry, I began producing Lingo programming support material in 1993. We waited in vain for third-party books or CD-ROMs to appear to supplement the rather meagre Macromedia documentation. In the meantime, we used the excellent Internet mailing list “direct-1”. Postings suggested publications on Lingo were imminent but nothing had appeared in the UK by the spring of 1995. At the time, I worked on a fractional contract for Coventry University and therefore approached AGOCC as a freelance in June 1995 for funds to develop some Lingo materials, based on the work already undertaken since 1993.

Ideally I would have spent more time researching the potential audience before starting work on the materials but this would have been a costly and time-consuming business. My idea was to get something based on my experience and intuition out to users within a very limited budget and then solicit feedback. In the event, there was little response direct from users. Edition 2.0 is therefore based once again on my instinct for which of the many new or changed aspects of Lingo should be addressed in a limited time and space. By the time this edition appears, Macromedia may have released Director 6: further revision or extension of the materials can be undertaken if it is warranted by the reaction to this version.

### Summary of changes from Edition 1.0

Director 5 was a major upgrade, with many changes and new features. Reviewing them all is outside the scope of this project, which continues to focus on those items of particular interest to the novice programmer. These are explored in more detail in *How To Use The Materials* and *Fixing Broken Programs*. With this in mind, here is a summary of the changes made in Edition 2.0:

- All movies from Edition 1.0 have been updated to the Director 5 format using the built-in Xtra “Update Movies”
- A folder has been added (*D5\_MOVIES*), containing more than twenty new movies exploring new or significantly changed aspects of Director relevant to learning to program.
- A new method of formatting Lingo scripts has been introduced with these new movies, using a colour-coding convention and more extensive comments for improved readability.

- A new section of the handbook has been created (*Fixing Broken Programs*), exploring the single issue which probably gives novices most difficulty.
- New references to printed and online material about Lingo and programming have been added to the *Annotated Bibliography*.
- Ten sample sound files from a commercial demonstration sampler have been added to the "*~RESOURCES*" folder, with purchase information in a 'ReadMe' file.
- Folders have been re-named and the folder hierarchy modified: **users of Edition 1.0 please note.**
- A new run-time Projector (*LingoProject\_2.0*) has been created, fixing some minor problems with the original and linking to a file-listing movie which can be modified for use elsewhere.



These changes should be useful both to those familiar with Director 5 and those who have yet to change from Director 4. This Edition also takes a brief look forward at likely features of interest in the forthcoming release of Director 6.

### Looking forward to Director 6

As of 21 April 1997, Macromedia's website was listing the following features of Director 6 which seem likely to be relevant to the novice Lingo programmer:

#### *Authoring and interface features*

- New score
- 120 sprite channels
- Multiple Score windows
- Score generation at run time
- Unlimited castmembers
- Switch casts dynamically

#### *Scripting-related features*

- Drag and drop behaviors
- Construct interactivity using menus and dialog boxes
- Object-oriented behaviors
- Behavior Inspector
- Includes over 30 packaged behaviors
- Auto-puppeting
- Support for rollover in Lingo

#### *Help and support features*

- Interactive online tutorial movies
- Macromedia Information Exchange
- Connected 'Help' system
- 350,000 developers worldwide, resulting in extensive support via training, user groups, conferences, online forums

For more up-to-date information, visit:

<http://www.macromedia.com/director/>

### **Limitations of the materials**

The materials do not constitute an open learning package. They have not been produced by a professional programmer and are not intended as instruction in advanced programming. Neither do they claim to be an authoritative guide to the Lingo language, which has over 500 elements.

This is not a commercial product and there may well be errors, omissions or undiscovered faults in the movies or the text. Macromedia Inc. have not seen or approved the materials. The movies have been used at various times in classes and workshops at Coventry School of Art & Design but no warranty is implied. Use them at your own risk!

All new movies and scripts were created on a PPC604 PowerMac with Director version 5.0.1. Although most of the Lingo should translate to the Windows environment, no attempt has been made to create dual-platform materials in Edition 2.0. A Windows version may follow at a later date.

## **How to use the materials**

---

### **Assumed knowledge and equipment**

It is assumed that users of these materials are familiar with the use of Director's cast and score to produce animation or interactivity and with the creation or import of graphics, sound and digital video to the cast. If you do not have this level of knowledge, stop reading now and return to the materials when you have explored these aspects of Director for a while.

Furthermore, you should know enough about setting up a Macintosh to allocate application memory, install extensions and control panels and sort out common operational problems.

Programming in Lingo involves a lot of typing and text editing, so you should be comfortable with a keyboard and with Macintosh text editing conventions and procedures.

In terms of equipment, you will need a 68040 Macintosh or better with at least 12Mb of RAM free for Director. Note that more application RAM is required when running Director on a PowerMacintosh. A complete installation, including the online Help system but excluding Macromedia's tutorial movies, occupies around 25Mb of disk space. You should not omit the Help files: they are an invaluable source of Lingo definitions and example fragments. The tutorial movies are of little use in my opinion and they occupy a great deal more space than in Director 4: you can safely omit them for day-to-day work. Director will require plenty of memory and disk space if you are working with movies with lots of sound or bitmap castmembers: consider 32Mb RAM and a 1Gb hard disk the desirable minimum. The movies used in this project are mostly less than 1Mb in size and many are less than 100Kb, so disk space and memory requirements are not as great.

The maximum colour depth used in the project is 8-bit. You will need at least one fourteen-inch display. Two monitors (one of which could be monochrome), will be extremely useful because of the extensive use Director makes of windows. Even seventeen-inch monitors do not provide enough screen space on their own. Access to a printer, preferably a PostScript printer, is highly desirable and will be essential from time to time.

You need MacOs System 7 (preferably the latest release) and, of course, a licensed copy of Director 5.0.1 or later. Director and the on-line help system should be installed. You will not need any additional fonts as system fonts have been used throughout the project. A word processor or text editor will be useful for formatting and printing scripts as Director's own facilities in this area are rather limited. Director 5 introduced Xtras as a replacement for XObjects and included Xtras for printing and database management. Explore use of these with the aid of the Macromedia manuals.

### **Production note**

Materials for Edition 1.0 were produced on a Macintosh IIvx running System 7.1 with 20Mb of RAM and a 230Mb disk. Two fourteen inch 8-bit displays were used. A Wacom Artpad was used for drawing and painting. Printed output was from a Personal LaserWriter 320 and the backup storage device was a d2 230Mb magneto-optical drive.

Edition 2.0 materials production was carried out with Director 5.0.1 on a PowerMacintosh 7600/132 with 32Mb RAM and 1Gb hard drive, running MacOs 7.5.3.

### **Organisation of the materials**

The handbook provides background information and a guide to the topics and movies. The movies are organised in five folders (*names in parenthesis*), in the following categories:

Algorithm Development	(AD_MOVIES)
Basic Scripting	(BS_MOVIES)
Director 5 Features	(D5_MOVIES)
Example Movies	(EG_MOVIES)
Program Visualisation	(PV_MOVIES)

which of course overlap to some extent. There is no implied order in which to study the movies.

Sample sounds, pictures and digital video are provided in a resources folder (~RESOURCES). The guiding design principle behind the organisation has been to avoid the approach taken by many programming texts: that is, defining the language elements and syntax in detail before looking at problems and their solution. The movies, especially those in *Program Visualisation*, incorporate text handling, sprite control, use of variables and so on. Commenting and layout of the code should help the learner understand what is going on.

### **Accessing the movies**

When accessing the movies for the first time, launch the Projector *LingoProject\_2.0*. This gives some introductory information. In future, you can launch any of the movies in the topic folders. You will find that some movies are linked together. Others may reside

in a sub-folder in order to make use of cast in a *SHARED.DIR* movie, which was the Director 4 mechanism for sharing castmembers between movies. A *SHARED.DIR* movie is no longer used and the name will have been changed to *SHARED.CST* in the process of updating. Movies in the *D5\_MOVIES* folder may reside in a sub-folder in order to make use of multiple casts (named *WHATEVER.CST*), which is the Director 5 mechanism for sharing castmembers.

You are expected to pull the movies apart and re-work them to suit your own requirements: this is a good way to learn about Lingo. If anything breaks it is probably NOT YOUR FAULT but you should nevertheless work only on copies.

## **Getting started with Lingo**

---

### **Determine your objectives**

The first thing to do is sort out your objectives. Do you intend to do a lot of programming in the future or are you just trying to acquire an appreciation of Lingo's capabilities and limitations? In the author's experience, post-graduate students (for example) seem to fall between these two extremes in needing to become sufficiently competent to produce their degree project. After that, they may need some programming skills in their first few jobs or they may never program again. In either event, they are unlikely to become professional programmers.

A frequent contributor to the `direct -1` mailing list, Simon Biggs, responded to a request from one of my students for advice on learning Lingo. He suggested allocating about three hours a day to the task for nine months: the student did this and became a fairly competent Lingo programmer. If you require less proficiency, then allocate less time: taught workshops at Coventry occupy three hours a week for four weeks, with a week for follow-up work between workshops. This is sufficient for many students' needs.

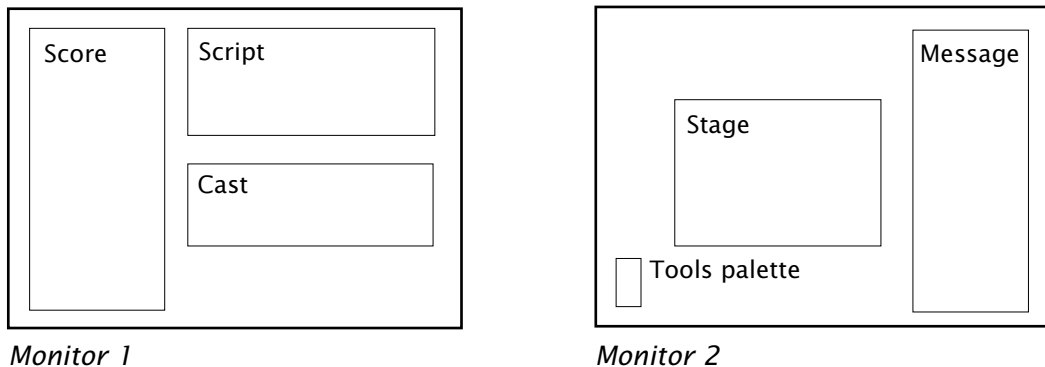
### **Assess your knowledge and experience**

The second thing to do is assess your existing knowledge and previous experience. Have you ever used any other programming language? If not, have you at least used spreadsheet or database software? Do you have a good grounding in mathematics (eg GCSE pass or better)? What about languages? Do you understand the roles of grammar, syntax and vocabulary in producing concise and precise text? Lack of experience or ability in these areas will *not* prevent you learning to use Lingo but any understanding you *do* have will help a great deal.

### **Set up your working environment**

Learning Lingo will be easier and more pleasant if you set up your working environment properly. This means arranging the hardware and configuring the software. Director uses a large number of

windows and two monitors will make it much easier to arrange these for maximum efficiency. An arrangement with two fourteen inch monitors I have found effective is:



If you don't have two monitors you will have to adapt this to your own equipment.

#### Setup suggestions

1. Have a small stage (e.g. 9 inch monitor or 320x240 pixels) on one monitor, leaving room for a tall thin Message window which doesn't obscure the stage.
2. Have **Score**, **Cast** and **Script** windows tiled on the second monitor.
3. Set **Movie Info** and **Preferences** up as has been done in the project movies.  
It is generally best to "load cast when needed" as this saves RAM and to specify "black and white user interface" as using non-system or custom palettes can make the coloured interface illegible.
4. Have the Lingo documentation to hand, that is *Using Lingo* and the *Lingo Dictionary*.

#### Using the Message window

Note that the Message window has been substantially enhanced since Director 4.

A good way to become acquainted with Lingo is to explore some elements through the Message window. Lingo is an interpreted language, which means that any valid Lingo statement will be processed as soon as it is entered. Feedback can thus be almost instantaneous. One-line Lingo statements can be entered in the Message window and processed. Pressing the RETURN key signals the end of the statement. Output from certain statements will be sent back to the window.

Open the Message window and position it where you can see the stage. Initially, you will be working without cast members or sprites but as soon as you introduce these you need a clear view of what is happening. This is where a small stage and/or multiple monitors become useful.

With the Message window active, check that the "Trace" option is not selected. Without starting the movie, type:

```
put the date
```

Terminate the line with a RETURN. From now on, this step will be assumed. The Message window should respond by displaying:

```
<today'sDate>
```

If you enter:

```
put the long date
```

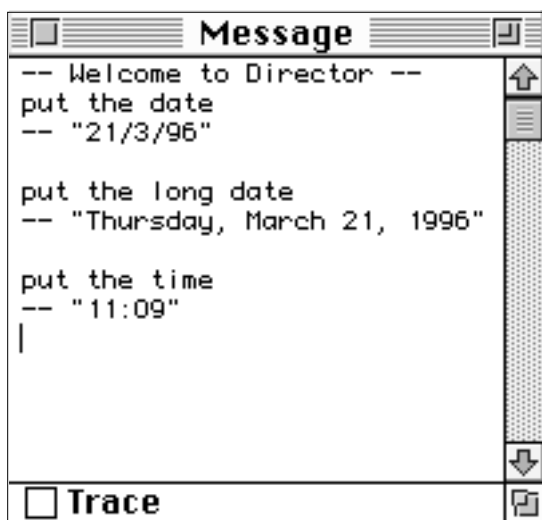
the Message window should display:

```
<day month year>
```

The Message window toolbar was introduced in Director 5. See Appendix One for further details.

The "Trace" function is now controlled by this toolbar button.

This demonstrates that Lingo is working even when a movie is stopped and that there are inbuilt Lingo elements which perform useful tasks. Later, the role of the Message window in diagnosing and correcting faults in programs will be explored.



Director 4 Message window



Director 5 Message window

### Coping with the unknown

Enter the following in the Message window:

```
set myName = "Ian"
```

replacing Ian with your own first name. There will be no response but this is a good sign! Next, enter:

```
put myName
```

and the window should display:

```
"Ian"
```

or whatever name you typed. This is an example of a programming construct called a *variable*.

Variables are just containers for whatever we want to put in them. In this case, we created a variable called <myName> and

used it to store the characters making up a first name. This could be useful wherever we wanted to retrieve a name that we did not know in advance, *e.g.* a name typed in by a user of our movie in response to a prompt. Try entering:

```
set myNumber = 25
```

and again there should be no response. Now enter:

```
put myNumber
```

and the response should be:

```
25
```

Note the absence of inverted commas. This means the Lingo interpreter is treating the digits 2 and 5 as the integer (whole) number 25 and can perform arithmetic on it. Enter:

```
set myNumber = (myNumber * 4)
```

followed by:

```
put myNumber
```

The result should be:

```
100
```

The parentheses are not necessary here but help when handling longer arithmetic expressions (see any maths primer on the rules of precedence in evaluating expressions). Again, the variable would be useful wherever we wanted to perform arithmetic on a number that we did not know in advance.

Skilful use of variables can make programs much easier to understand. You can find examples of variable usage in the project movies and in movies from other sources. In general, variables are used whenever we want to store results temporarily and process them later without knowing the actual values in advance.

### Learning more about Lingo elements

Lingo has over 500 elements, so trying them all in the Message window would take a very long time! In fact, many elements cannot be explored in this way because they do not in themselves form complete statements or do not by themselves produce any output in the window.

You can access elements through the Lingo menus in the Message and Script window toolbars. Choosing an item from the menu will place a skeleton Lingo statement at the text cursor position in the active window.

Research has shown that examining the syntax and vocabulary is not a good way of learning a programming language anyway [PHILLIPS, 1996]. This project therefore concentrates on the tasks to be performed and the techniques for developing solutions, rather than the details of the Lingo language.

### Using the Message window with cast and score

The Message window is not divorced from cast and score. Sprites can be manipulated from the Message window and Lingo scripts can cause sprites to feed output back to the window. In fact,

You can also look up element definitions in the Lingo Dictionary or the on-line help system. Either method will give you an example or examples of the element usage but these may not be complete in themselves. They may also be confusing to a novice and some are actually wrong!

entering simple Lingo statements into the Message window is an invaluable technique for testing out scripting ideas. Get used to working with castmembers and sprites in this way before trying more complex scripts.

Create a one-frame movie with this enterFrame script in the Script channel of the score :

```
on enterFrame
  go the frame
end enterFrame
```

This will cause the playhead to loop on the frame when the movie is played. Next, create a simple QuickDraw or bitmap shape. Note the registration point: for a QuickDraw shape it is top left and cannot be reset; for a bitmap it is at the centre by default and can be reset. If you are using a bitmap shape, transform the colour depth to 1-bit. Place the castmember in channel 1 of the score. This will position it centre stage.

Play the movie. Open the Message window and position it where you can see the stage and the sprite. Enter:

```
put the foreColor of sprite 1
```

and the response should be:

```
255
```

that is, black. Now enter:

```
puppetSprite 1, TRUE
```

followed by:

```
set the foreColor of sprite 1 = N
```

where N is an integer number of your choice between 0 and 255.

The sprite should change colour. If things go wrong, check that you are entering one complete line of Lingo at a time into the Message window, terminated by a RETURN.

You can alter the sprite's location in a similar way, using commands such as:

```
put the locH of sprite 1
```

to find out what the current x-axis coordinate is, followed by:

```
set the locH of sprite 1 = N
```

where N is an integer number of your choice. This will set an absolute position. Alternatively, you can specify a position relative to the current position like this:

```
set the locH of sprite 1 = the locH of sprite 1 + N
```

You could of course set up a variable to store the current and new locations: this would make each statement shorter.

The Director 5 Lingo property "loc of sprite" can be used instead of "locH" and "locV", especially where you wish to change both horizontal and vertical location at the same time. The property value is given as a point in the form point(x,y). Try typing:

```
put the loc of sprite 1
```

and you should receive a response such as:

```
point(160,120)
```

Note that a sprite must be a puppet to be controlled in this way.

An `updateStage` command is necessary if you want the stage redrawn immediately, as you will if you are animating a sprite within one frame.

Once Lingo statements have been tested in the Message window, you can begin combining them into longer scripts. These scripts can then be placed in the Script channel, attached to a frame or frames, attached to a castmember or placed in the movie script, according to the guidelines in *Using Lingo*.

### **Summary**

You should now have a clear idea of your objectives and of how much time you are willing and able to devote to learning Lingo. Your eventual level of proficiency will depend far more upon these factors than upon any innate ability you may have. You have been introduced to variables and single-line Lingo statements and should be able to make use of the Message window to develop and debug your scripts.

## **Algorithm Development**

---

### **What is an algorithm?**

Lingo is an example of a programming language for a computer that works in sequential steps unless told to branch off or repeat something. Tasks must be analysed with this in mind and the process of doing this and developing solutions is called algorithmic thinking. Developing algorithms, which can be done in human language, should precede all but the most trivial coding in Lingo.

The purpose of this section is to encourage you to develop your algorithms thoroughly, since that will help ensure your programs work properly. It is not to teach you how to code in Lingo but Lingo conventions have been used for certain things where appropriate in order to make the transition to code as straightforward as possible.

### **Where do I start?**

The most common question is programming is “Where do I start?”. The answer is:

1. State your assumptions.
2. Break your task down into smaller tasks.
3. Formulate step by step instructions to carry out each of the smaller tasks.

What follows is an example of algorithm development that will not actually be turned into Lingo, although some Lingo terms and conventions will be used to make the relationship between algorithm and code as obvious as possible. The example is modified from one given in [PATTERSON *et al*, 1989], a useful book for novice programmers. When you have studied the example, take something you are working on in Director and apply the same approach. Be careful to avoid coding in Lingo until you are satisfied with your algorithms.

*Task*

Give instructions for making and serving coffee while reading the paper before breakfast.

*Assumptions*

Instructions are for a machine that has the physical abilities of a human but is as literal-minded as a computer. So, it can pick up a coffee pot but will have to be told exactly what to do and when to do it.

*Decomposition*

There are two main parts: preparing the coffee and reading the paper. Initial algorithm:

```
Prepare coffee
Read paper
```

We now proceed in a manner sometimes called “stepwise refinement”, stating any fresh assumptions as we go along. We will refine the algorithm by considering the two parts separately. New material will be in bold type like this:

```
newMaterial
```

while existing material will remain in plain type.

**Stepwise refinement of the algorithm**

Our initial two-step algorithm can first be broken into three steps.

```
Make coffee
Serve coffee
Read paper
```

What is involved in making the coffee?

Assume a filter coffee maker and fresh beans.

```
Put new filter in coffee maker
Grind beans
Put grounds in filter
Boil water
Pour boiling water over grounds
Let drip
Serve coffee
Read paper
```

If you go on like this, the algorithm description soon becomes a long and confusing list of instructions. Like most modern programming languages, Lingo lets you define your own procedures and call them up by name when you need them. This allows you to hide detail, which has been shown to be one of the best ways of improving program quality and increasing programmer productivity.

See the *Basic Scripting* section and the project movies for more information and ideas about how to organise and structure your Lingo code. In following this example, all you need to know is that the Lingo term for a procedure is “handler” and that you should

avoid using any of the Lingo keywords (or “reserved words”) as names for your handlers: check for conflicts by looking through the Lingo menu.

If we assume a procedure called `MakeCoffee` has been defined elsewhere, our main algorithm could be refined to:

```
MakeCoffee
  Serve coffee
  Read paper
```

Note the difference between the word we have invented in line one and lines two and three: the latter are close to plain English while line one is a step closer to Lingo code.

Handlers begin and end with the keywords “on” and “end”, so the handler to make coffee as defined above could look like this:

```
on MakeCoffee
  Put new filter in coffee maker
  Grind beans
  Put grounds in filter
  Boil water
  Pour boiling water over grounds
  Let drip
end MakeCoffee
```

The only thing we have changed is to bracket the steps for making coffee with handler definition lines. The indentation helps us read the procedure as these lines, beginning with “on” and “end”, now stand out. The computer will automatically substitute the full procedure for the term `MakeCoffee` when the program executes. Apart from hiding detail, writing our algorithm out in this form means it is easy to give instructions to make more coffee. We just add one line:

```
MakeCoffee
  Serve coffee
  Read paper
MakeCoffee
```

If you are following this closely, you might realise that the `MakeCoffee` procedure as it stands is not really suitable for use more than once: it does not contain any instructions about removing the used filter! You can rewrite that procedure to take account of this if you wish.

Giving instructions to a “virtual machine” in this form is fine but it is not necessarily easy for humans to understand those instructions or the logic behind them. To improve readability, we add comments: lines in our algorithm that will be ignored by the machine but help make the algorithm intelligible to humans. A lot of programming effort goes into fixing or maintaining existing programs, so anything which will help us understand our programs is a good idea.

All programming languages have “comment conventions”: ways of distinguishing instructions to the machine from text meant for humans. In Lingo, comments are distinguished by two hyphens in front of the text:

```
--This is a comment in Lingo.
```

```
--Lines beginning "--" will be ignored by the machine.
```

Blank lines are also ignored by the machine, so we don’t need any comments for those. We can lay out our algorithms fairly clearly using nothing more than basic visual design sense, comments and white space. The coffee algorithm can now be further refined:

So much of the revised algorithm is new that it has all been set in plain type. From now on, we will revert to using bold for new material. Note also that we have taken care to get our revised algorithm description on one page.

In general, if a description won’t fit on one page, it is time to break it down further. For instance, handler definitions could go on a separate page. The same advice applies to diagrams of algorithms (“flowcharts”) which will be introduced later and explored further in Program Visualisation.

```
--Algorithm for making coffee and reading paper.
--Assumptions:
--Machine has mind as literal as a computer but
--human physical abilities.
--This new version uses three procedures:
--MakeCoffee, ServeCoffee, ReadPaper
-----

--Main program starts
  MakeCoffee
  ServeCoffee
  ReadPaper
--Main program ends---

--handler definitions start---
--handler to make filter coffee from fresh beans
  on MakeCoffee
    Put new filter in coffee maker
    Grind beans
    Put grounds in filter
    Boil water
    Pour boiling water over grounds
    Let drip
  end MakeCoffee

--handler to serve coffee (for one person)
  on ServeCoffee
    --we still have to write this procedure
  end ServeCoffee

--handler to read paper
  on ReadPaper
    --we still have to write this procedure
  end ReadPaper

--handler definitions end---

--TrailerNotes
--We now have a clear idea of what still remains to be
--done, just by looking at the algorithm.
--Last revised 5 Feb 96
-----
```

*Note on the project movies*

The project movies use various layouts for Lingo code. Look at the movie scripts and "script store" text fields for examples. Develop your own standard format to help you distinguish between the various parts of your code. This approach reduces the likelihood of making programming errors and makes it easier to find those errors that do occur.

**Variables**

So far, we have only organised the instructions in our developing program. We usually want to organise information as well. For instance, what can we do about specifying how much mik and sugar to serve with the coffee?

We will use the construct called a "variable" that was introduced in *Getting Started With Lingo*. Remember, a variable is simply a container that can hold whatever we specify. Variables should be given meaningful names (you may encounter programs that use cryptic names like "N" or "x" but this is not usually a good idea), so we will use NumberOfSpoons and TakesMilk for our variable names. These names give some clue to their purpose and likely contents. We can now write the procedure ServeCoffee:

```
--handler to serve coffee (for one person)
--ServeCoffee uses two variables:
--NumberOfSpoons for number of teaspoons
--of sugar per cup
--TakesMilk to indicate whether milk should
--be added to cup

on ServeCoffee
    Pour coffee from pot into cup
    Add NumberOfSpoons spoons of sugar to cup
    IF TakesMilk = yes THEN
        Add milk to cup
    END IF
    Stir
end ServeCoffee
```

The keywords IF, THEN, ELSE and END IF are in upper-case in the example code purely for clarity when reading the example: we would normally use lower-case in working programs.

This introduces a further construct, the conditional action, indicated by the words "IF ... THEN... END IF". The general form of this is:

```
IF <condition> THEN
    Action(s)
END IF
```

If the condition is not met, the action(s) are not performed and we move on to the next instruction in the sequence. This is what is sometimes known as a "branch". Branching, and another construct called "repetition", are explained in more detail later in this section.

The variables will have to be given (“assigned”) values before they can be used. This will be done in the main program, which will therefore look like this:

```
--Main program starts
MakeCoffee
  set NumberOfSpoons = 2      --two sugars please
  set TakesMilk = yes        --and milk
  ServeCoffee
  ReadPaper
--Main program ends
```

This is fine, except that Lingo would not know about these variables in the main program: they were introduced in the handler `ServeCoffee`. Variables which will be used only within a handler are called “local”; variables which can be used by other handlers (or movies) are called “global”.

Local variables can be used “on the fly” but globals have to be declared as such and given a value before being used. Frequently, variables will be “initialised” when first declared; that is, given sensible initial values. In Lingo, this is often done whenever the movie is played but it can be done wherever is most appropriate for the task in hand. The algorithm re-written to use global variables `NumberOfSpoons` and `TakesMilk` is described on the next page.

```

--Algorithm for making coffee and reading paper.
--Assumptions:
--Machine has mind as literal as a computer but human physical abilities.
--This new version uses three procedures:
--MakeCoffee, ServeCoffee, ReadPaper
-----

--Main program starts
global NumberOfSpoons, TakesMilk
MakeCoffee
  set NumberOfSpoons = 2      --two sugars please
  set TakesMilk = yes        --and milk
  ServeCoffee
  ReadPaper
--Main program ends-----

--handler definitions start---
--handler to make filter coffee from fresh beans
on MakeCoffee
  Put new filter in coffee maker
  Grind beans
  Put grounds in filter
  Boil water
  Pour boiling water over grounds
  Let drip
end MakeCoffee

--handler to serve coffee (for one person)
--ServeCoffee uses two variables:
--NumberOfSpoons for number of teaspoons of sugar per cup
--TakesMilk to indicate whether milk should be added to cup
on ServeCoffee
  global NumberOfSpoons, TakesMilk
  Pour coffee from pot into cup
  Add NumberOfSpoons spoons of sugar to cup  --value set in main program
  IF TakesMilk = yes THEN                    --value set in main program
    Add milk to cup
  END IF
  Stir
end ServeCoffee

--handler to read paper
on ReadPaper
  --we still have to write this procedure
end ReadPaper
--handler definitions end-----

--TrailerNotes---
--We now have a clear idea of what still remains to be done,
--just by looking at the algorithm.
--Last revised 5 Feb 96
-----

```

Suppose we wish to serve coffee to more than one person. We can now make further use of the variables by giving them different values for the second serving. The algorithm for the main program now becomes:

```
--Main program starts---
global NumberOfSpoons, TakesMilk
MakeCoffee
set NumberOfSpoons = 2      --two sugars please
set TakesMilk = yes        --and milk
ServeCoffee
set NumberOfSpoons = 1      --only one sugar please
set TakesMilk = no         --no milk
ServeCoffee
ReadPaper
--Main program ends-----
```

We could refine this further by introducing a variable called `RecommendedAmount` which could be set once for the whole program:

```
global RecommendedAmount
set RecommendedAmount = 1
```

We could then assign new values to `NumberOfSpoons` in this way:

```
--Main program starts---
global NumberOfSpoons, TakesMilk
MakeCoffee
set NumberOfSpoons = RecommendedAmount + 1
--one more sugar won't hurt
set TakesMilk = yes      --and milk
ServeCoffee
--second serving
set NumberOfSpoons = RecommendedAmount
--very wise
set TakesMilk = no         --no milk
ServeCoffee
ReadPaper
--Main program ends-----
```

Note that we assign new values to variables using notation like this:

```
set NumberOfSpoons = NumberOfSpoons + 1
```

or this:

```
set NumberOfSpoons = NumberOfSpoons - 1
```

which means “increase (or decrease) `NumberOfSpoons` by one”.

Once we start using variables and procedures more than once, we run the risk of having to keep track of lots of values as the main program grows longer. It would be simpler if our procedures could be supplied with values as they were called. For example, we could

then call `ServeCoffee` like this:

```
ServeCoffee(2, yes)
```

meaning two spoons of sugar with milk. This is a lot more concise and readable than:

```
set NumberOfSpoons = 2      --two sugars please
set TakesMilk = yes        --and milk
ServeCoffee
```

provided we have rewritten `ServeCoffee` to know that it will receive the information in this form. In Lingo, variables passed to a handler are called “parameter variables” or “arguments”. Handlers which accept parameter variables or arguments can be used to do the job of several ordinary handlers.

The parameter variables look like ordinary variables but only exist within the procedure that uses them. The main program looks like this:

```
--Main program starts---
MakeCoffee
ServeCoffee(2, yes)
ReadPaper
--Main program ends-----
```

and `ServeCoffee` now looks like this:

```
--handler to serve coffee (for one person)
--ServeCoffee now uses NumberOfSpoons and TakesMilk
--as “parameter variables” or “arguments” to indicate
--amount of sugar and whether milk should
--be added to cup.
on ServeCoffee
    Pour coffee from pot into cup
    Add NumberOfSpoons spoons of sugar to cup
    --value set in main program
    IF TakesMilk = yes THEN
        --value set in main program
        Add milk to cup
    END IF
    Stir
end ServeCoffee
```

This of course means values are passed from the main program to the `ServeCoffee` handler. The work of maintaining the variables is now localised in the procedure they relate to, rather than in the main program as illustrated on the previous page. At the same time, the main program becomes more readable.

Multiple servings, which would have made for a complicated series of variable assignments, can now be accommodated like this:

```
--Main program starts---
MakeCoffee
ServeCoffee(2, yes)
ServeCoffee(1, no)
ServeCoffee(0, yes)
--and so on..
ReadPaper
--Main program ends-----
```

Over the page is the Lingo from a working example (see the movie *algoDevCoffee\_V1.0* in the *Algorithm Development* folder) that simulates the execution of the *ServeCoffee* procedure by putting text messages into a field.

*Note 1*

The *TakesMilk* argument uses the values 1 for Yes and 0 for No in the working version.

*Note 2*

There is no Main Program in the working example. The *ServeCoffee* handler is called by clicking a button on the stage: different buttons specify different values for sugar and milk. The principle of passing values from the calling code to a procedure is the same.

*Note 3*

Note 2 illustrates one problem with the Lingo/Director environment: bits of the task are carried out by sprites on the stage or the movement of the playhead and bits by Lingo code. This is quite different from “traditional” languages. It means there is great flexibility in this environment but also great potential for confusion.

### Controlling execution flow

The normal flow of execution is step by step from start to finish of a program. This is known as a *sequence*. We have already encountered the concept of changing this flow by branching to another set of instructions and then rejoining the main flow. This is known as *selection*. A further alteration in the flow occurs when we want to perform some action or actions several times before proceeding: this is known as *repetition*. Thus we have three constructs for controlling execution flow:

- Sequence
- Selection
- Repetition

These constructs are explored in more detail in the *Program Visualisation* section and movies.

```

--MOVIE to demonstrate use of arguments/parameters in handlers
--as part of algorithm development topic.

--HeaderNotes
--
--Simulates coffee serving procedure by putting messages in text fields.
--Scripts attached to buttons call the handler and pass values
--for NumberOfSpoons and TakesMilk.
-----
--handler definition--

--The ServeCoffee "procedure" written as a handler
--that takes two arguments (or parameters), NumberOfSpoons and TakesMilk.

on ServeCoffee NumberOfSpoons, TakesMilk

    set the text of field "outputFLD" to EMPTY
    --just clearing the field so we know the program executes

    put "Pour coffee from pot into cup" &RETURN into field "outputFLD"

    --sugar?
    IF NumberOfSpoons = 1 THEN
        put "Just enough sugar" &RETURN after field "outputFLD"
    ELSE IF NumberOfSpoons > 1 THEN
        put "Too much sugar" &RETURN after field "outputFLD"
    ELSE IF NumberOfSpoons = 0 THEN
        put "Wise to avoid sugar" &RETURN after field "outputFLD"
    END IF

    --milk?
    IF TakesMilk = 0 THEN
        put "No milk" &RETURN after field "outputFLD"
    ELSE if TakesMilk = 1 then
        put "Add milk" &RETURN after field "outputFLD"
    END IF

    put "Stir" &RETURN after field "outputFLD"

end ServeCoffee

--handler definition ends---

--TrailerNotes---
--
--
--:
--Last modified 21 Feb 96.
-----

```

**Summary**

This is as far as we will go with stepwise refinement of the coffee-making algorithm. The section has introduced the following important concepts:

- Algorithms
  - stating assumptions
  - decomposition
  - stepwise refinement
  - representing algorithms by diagrams
  - execution flow
  - comments
- Procedures
  - handlers (defining a procedure)
  - parameters and arguments
- Variables
  - declaring a variable
  - initialising a variable
  - assigning a value to a variable

and they are the basis of successful programming. The more you use them, the easier you will find them to understand.

You can get more information from the *Annotated Bibliography*. Books I have found particularly useful are [CHANTLER, 1981; MEEK *et al*, 1983; PATTERSON *et al*, 1989]. See the *Basic Scripting* and *Program Visualisation* movies for examples of structuring and organising actual Lingo code.

## Basic Scripting

---

### First things first

Lingo statements look something like English but the rules governing their construction and use are stricter. As with any programming language, you need to become familiar with certain concepts and bits of jargon. The most important concepts are:

1. The idea of issuing instructions in the form of *statements* in a special language called Lingo.
2. The use of Lingo *elements* such as keywords, operators and functions, together with variables and text strings, to construct *statements*.
3. The packaging of a number of statements into a *script*.

Thus, elements can be combined into statements and statements can be combined to form scripts. Complicated scripts can be repackaged into one or more *handlers* that can be called with one word statements. Try to get these distinctions clear for the sake of swift and unambiguous communication with all concerned.

### Learning the jargon

Some terms have special meanings in the context of programming. Familiarity with these will be essential later on. It would be helpful to read Chapter 3 of *Using Lingo*, entitled “Concepts” before proceeding much further and particularly helpful to read “The Elements of Lingo” on pp 119-120 first. Lingo elements are classified there as:

commands	instructions that cause something to happen while a movie is playing
functions	elements that return some value
keywords	reserved words with a special meaning, <i>e.g.</i> ‘the’, which indicates that the word following is a property
properties	attributes of an object, such as locH or locV (horizontal and vertical location of an object in screen pixels)

operators	symbols or words that change the value of other things, <i>e.g.</i> the '+' operator adds two values together to produce a new value
constants	elements that don't change, <i>e.g.</i> FALSE and EMPTY, which always have the same meaning

### What's in this section?

Many programming texts start by examining the syntax and vocabulary of a language in detail. This is not all that helpful to a novice programmer and this project does not take that approach. Rather, it provides example movies that you can study, take apart and modify so that you learn by doing. In this section, there are movies on:

If you are familiar with the WorldWideWeb, compare the links between frames and movies in Director with the links in Web documents and the HTML implementation of links.

- changing frames with scripts
- controlling sprites with scripts
- linking movies with scripts that
  - a. pass control to another movie
  - b. play all or part of another movie and return
- various other simple button or frame-triggered actions
- getting and responding to user input  
*e.g.* get a name typed in and respond
- the execution flow in a script
  - a. sequence
  - b. selection (or "branching")
  - c. repetition (or "iteration")

These movies also demonstrate ways of organising your castmembers (the program "data") and scripts (the program "source code") that make it easier to understand what your scripts do when they work and what is wrong with them when they don't.

### Useful things to know

Lingo works by passing *events* and *messages* through a *hierarchy* until they encounter handlers that can deal with them. To quote *Using Lingo*:

*When events – such as clicking the mouse or exiting a frame – occur, Director sends a message describing the events to a series of objects... When an object has a script that is set to respond to the particular message, the instructions in the script are carried out.*

Study the Lingo documentation carefully until you understand the messaging process (see *Using Lingo*, p80 ff, "How Director responds to messages"). This manual is not very clear on what happens when a frame is drawn. See the *Lingo Workshop* [THOMPSON & GOTTLIEB, 1995] for a good explanation of "the life of a frame" and for extra information on the messaging hierarchy.

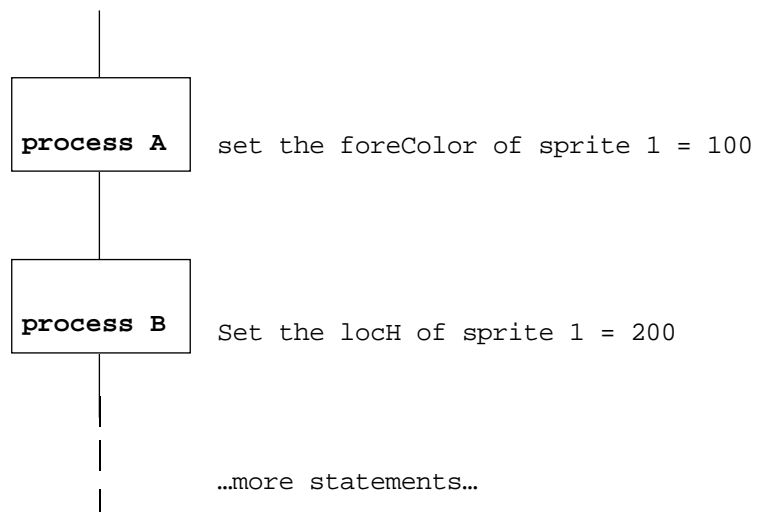
You also need to know that it is either the score or Lingo which is in control of your movie at any moment. Weird things happen if you forget this: for instance, if puppet sprites are subjected to score transitions. Make sure you know which mechanism has control and take the appropriate action. This will involve actions such as applying and removing puppet status when combining scripted animation with score transitions and can lead to a lot of “housekeeping” work. Neglect it at your own risk!

Finally, consult the booklet of tips for developers which comes with Director for advice on ensuring the best performance of your movies. The *Lingo Workshop* is also very good on this topic.

### Controlling execution flow in scripts

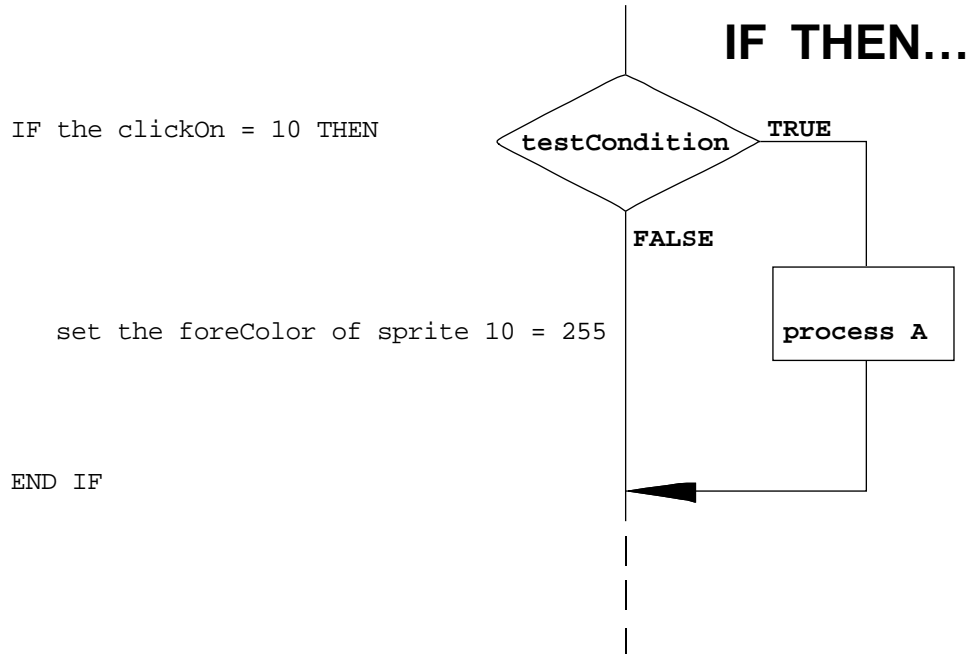
The concept of step by step execution of statements was introduced in *Algorithm Development*, together with the associated concepts of branching to other statements and repeating certain statements. When first creating Lingo scripts, it is best to start with the step by step approach. This is known as a *sequence* and can be represented diagrammatically like this:

## SEQUENCE

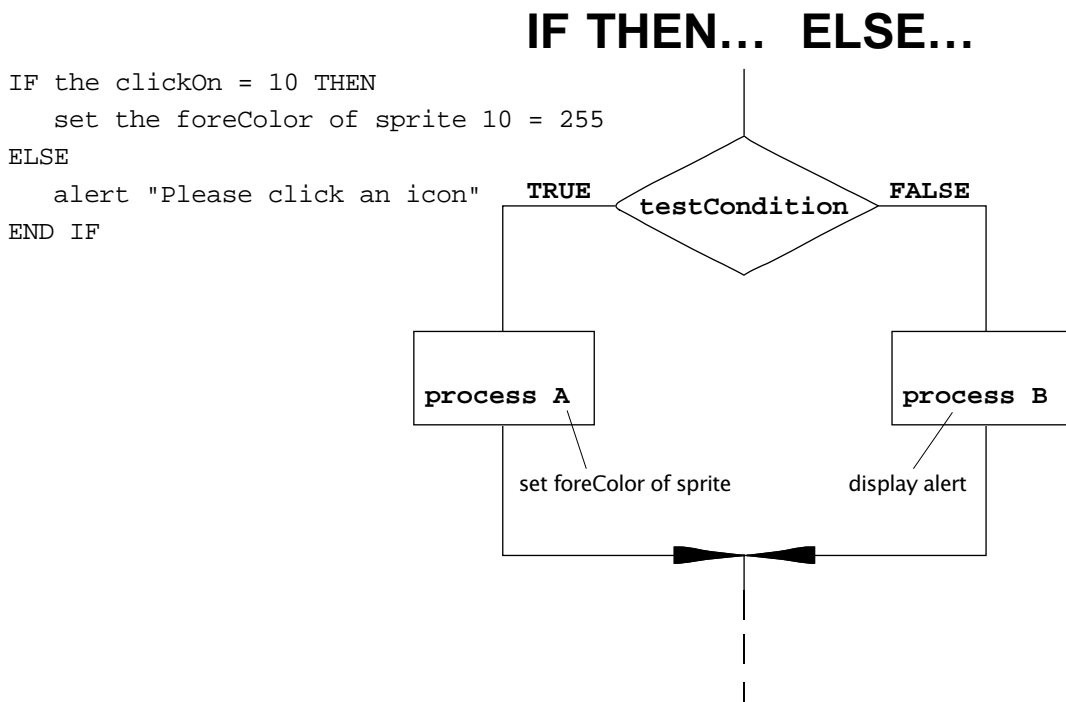


Quite a lot can be accomplished in this manner (see the movie *sequentialScript1* for an example) but long sequences become unwieldy. Furthermore, the sequence cannot cope with conditions that may arise. For example, you might want different actions to follow from a user’s choice of icons on a screen.

To cope with this, we use a construct called *selection* (also known as branching or conditional execution). Selection comes in two forms: a simple branch in which actions are performed only IF a condition is met, and a two-way branch, in which alternative actions are performed whether or not the condition is met. The two forms are shown diagrammatically on the next page.



Note that the indenting of the code, which is performed automatically in Director script windows, helps readability. It also performs a check on your code: if the END IF statement is missing, the code is incorrect and will not indent.



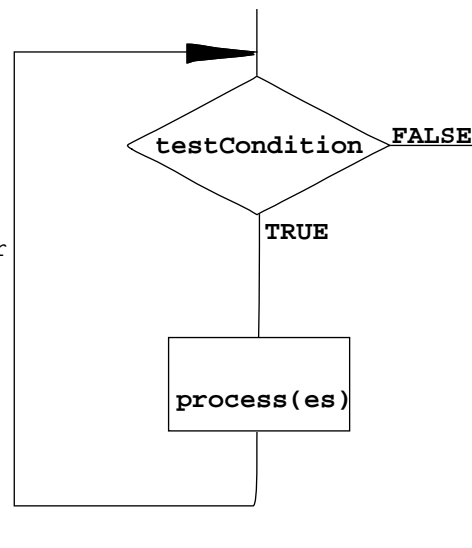
Note that the “process” boxes could actually contain a series of statements: we are showing one for the sake of clarity. The digram for the repeat construct, which is the final building block we need for complex scripts, illustrates this by referring to “process(es)”.

## REPEAT WHILE...

```

set myNumber = 10
REPEAT WHILE myNumber <= 100
  set the stageColor = myNumber
  set the locH of sprite 1 = myNumber
  set myNumber = myNumber + 10
END REPEAT

```



It has been shown mathematically that these three constructs can be combined to create code structures that will solve any problem which *can* be solved algorithmically. The process of combining is known as *nesting*.

Developing programs in the ways advocated (in *Algorithm Development* and this section) became known as “structured programming”. The approach was devised to tackle the enormous problems caused by highly individualistic programming and should help you develop sound solutions in Lingo.

A step on from this is the method known as “object oriented programming”(OOP). This offers advantages over simple structured programming for many of the projects tackled with Director and Lingo. Interest in the approach began to develop rapidly from the late 1980s and Lingo supports many features of OOP, though the documentation is rather patchy. The *Annotated Bibliography* contains some suggestions for further reading for those interested.

### Other sources of information

As stated in the Introduction, the project came about because Lingo tutorial material was in such short supply. There are now some useful sources of information and advice about Lingo and more are appearing all the time. Those available at the time of writing are listed in the *Annotated Bibliography*.

## **Program Visualisation**

---

### **What is program visualisation?**

This term is used to cover methods of representing program data, source code, or execution that help us visualise what is going on. Academic research such as that cited in *Computer Program Visualisation in Art and Design Computing* [PHILLIPS, 1996] has suggested that these methods, which usually have a strong visual component, can help novices learn to program.

### **What methods are used in these materials?**

The established methods of program visualisation can be summarised as:

- **Commenting**  
Adding lines (which will not be executed) to program source code to clarify structure and function.
- **Pretty printing**  
Formatting program source code (automatically if possible) to clarify structure and function.
- **Flowcharting**  
Using static diagrams to clarify algorithms and program execution flow.
- **Code highlighting**  
Drawing attention to lines of source code as the program executes to clarify execution flow.
- **Watching**  
Using text fields to store the changing values of variables or program states. This method can be used for many purposes, including troubleshooting.
- **Animating**  
Using moving elements (and sound) to clarify the structure or purpose of a program.

All of these methods have been used in one form or another in this project. Some methods have been used extensively, others merely in an experimental manner. Work on program visualisation continues at Coventry School of Art and Design.

### **Strengths and weaknesses of visualisation methods**

Some methods can be employed without any special equipment, software or technical skill. Others require a great deal of programming support to be properly implemented.

#### *Commenting*

Commenting can and should be undertaken by anyone writing Lingo programs. Comments have to be inserted manually. There is no guarantee that comments will be meaningful - comments frequently describe what the author *thinks* the code does, rather than what it *actually* does. Extensive use is made of comments in the project movies.

#### *Pretty printing*

Can be undertaken by anyone writing Lingo programs. Research has concentrated on automatic formatting based on analysis of source code but novice Lingo programmers have to format manually. The method can be used to format code and comments in script windows or to format for print but there is considerable effort involved, even in examples as short as those on pages 31-35.

On-screen we can make use of colour and a rather limited range of typographic effects: italic type rarely displays well, for example. In print, we can insert diagrams and use a wide range of typographic effects but will usually rely on monochrome or grey-scale printers for output. An example of each approach is included in the *Program Visualisation* movies.

#### *Flowcharting*

Producing flowchart components is not difficult for anyone with a visual education: employing them to sort out program logic is less easy and dealing with complex selection or repetition can be difficult. Flowcharts are nevertheless a valuable aid to program development and are akin to animation storyboards in many respects. Simple diagrams have been used to illustrate the basic execution flow constructs in some *Program Visualisation* movies.

#### *Code highlighting*

Stepping a highlighter bar through on-screen source code is a simple way of illustrating execution flow in a Lingo script. The chief difficulty lies in making sure the right lines are highlighted: there is no mechanism in Lingo for automatically picking the line currently being executed. The technique is used in some *Program Visualisation* movies.

#### *Watching*

Some programming environments include "watcher" windows. Lingo is supplied only with the Message window, though sources such as the Gray Matter *MediaBook* CD offer several additional

**Example of pretty printing**

Here is a Lingo script defining a handler, printed without comments or formatting. The only clue to the structure of the code lies in the indentation.

```

on replaceNumberAnim1
  global initH, incrementH, newH, theHotLine, myNum
  puppetSprite 12, TRUE
  set initH = the locH of sprite 12
  set incrementH = 48
  set newH = initH + incrementH
  set theHotLine = 2
  hilite line theHotLine of field "codeFragmentFLD"
  set theHotLine = theHotLine + 2
  hilite line theHotLine of field "codeFragmentFLD"
  set myNum = 2
  put myNum into word 6 of field "varWatchFLD"
  repeat while myNum <= 6
    set theHotLine = 5
    hilite line theHotLine of field "codeFragmentFLD"
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
    set the locH of sprite 12 = newH
    set newH = (newH + incrementH)
    updateStage
    set the castNum of sprite 8 = myNum
    updateStage
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
    set myNum = myNum + 1
    put myNum into word 6 of field "varWatchFLD"
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
  end repeat
  set theHotLine = 5
  hilite line theHotLine of field "codeFragmentFLD"
  beep 2
  hilite line 1 of field "varWatchFLD"
  beep 2
  set the text of field "testConWatchFLD" = "Loop condition failed - end
repeat"
  hilite line 1 of field "testConWatchFLD"
  set theHotLine = 8
  hilite line theHotLine of field "codeFragmentFLD"
  set theHotLine = theHotLine + 1
  hilite line theHotLine of field "codeFragmentFLD"
end replaceNumberAnim1

```

Comments are added on the next page to increase the readability of the source code.

```

--Animating a loop with display of:
--changing loop testCondition variable value,
--changing Cast selection as loop executes,
--and hilited code line in handler fragment.
-----
on replaceNumberAnim1
  global initH, incrementH, newH, theHotLine, myNum
  puppetSprite 12, TRUE --the cast hiliter
  set initH = the locH of sprite 12 --over the bitmap question mark
  set incrementH = 48 --no. of pixels between centres of cast
  thumbnails
  set newH = initH + incrementH --should put us over bitmap number "1"
  set theHotLine = 2
  hilite line theHotLine of field "codeFragmentFLD"
  set theHotLine = theHotLine + 2
  hilite line theHotLine of field "codeFragmentFLD"
  set myNum = 2
  put myNum into word 6 of field "varWatchFLD"
  repeat while myNum <= 6
    set theHotLine = 5
    hilite line theHotLine of field "codeFragmentFLD"
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
    set the locH of sprite 12 = newH
    set newH = (newH + incrementH)
    updateStage
    set the castNum of sprite 8 = myNum
    updateStage
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
    set myNum = myNum + 1
    put myNum into word 6 of field "varWatchFLD"
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
  end repeat
  set theHotLine = 5
  hilite line theHotLine of field "codeFragmentFLD"
  beep 2
  hilite line 1 of field "varWatchFLD"
  beep 2
  set the text of field "testConWatchFLD" = "Loop condition failed -
end repeat"
  hilite line 1 of field "testConWatchFLD"
  set theHotLine = 8
  hilite line theHotLine of field "codeFragmentFLD"
  set theHotLine = theHotLine + 1
  hilite line theHotLine of field "codeFragmentFLD"
end replaceNumberAnim1

```

```

--handler definitions--
--Animating a loop with display of changing loop testCondition variable value,
--changing Cast selection as loop executes, and hilited code line in handler fragment.
-----
on replaceNumberAnim 1
  --preparation
  global initH, incrementH, newH, theHotLine, myNum
  --actually, not used outside this handler at present but...
  puppetSprite 12, TRUE --the cast hiliter
  --record castmember telltale initial position
  --(and set up other variables for later)
  set initH = the locH of sprite 12 --over the bitmap question mark
  set incrementH = 48 --number of pixels between centres of cast thumbnails
  set newH = initH + incrementH --should put us over bitmap number "1" (castNum 2)
  --hilite entering the handler
  set theHotLine = 2
  hilite line theHotLine of field "codeFragmentFLD"
  --hilite initial setting of test variable
  set theHotLine = theHotLine + 2
  hilite line theHotLine of field "codeFragmentFLD"
  set myNum = 2
  put myNum into word 6 of field "varWatchFLD"
  repeat while myNum <= 6
    --hilite start of loop
    set theHotLine = 5
    hilite line theHotLine of field "codeFragmentFLD"
    --hilite setting of castNum to variable
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
    --hilite first cast member in sequence
    set the locH of sprite 12 = newH
    set newH = (newH + incrementH)
    updateStage
    --do Nth frame of the animation
    set the castNum of sprite 8 = myNum
    updateStage
    --hilite incrementing of test variable
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
    set myNum = myNum + 1
    put myNum into word 6 of field "varWatchFLD"
    --hilite end of loop
    set theHotLine = theHotLine + 1
    hilite line theHotLine of field "codeFragmentFLD"
  end repeat
  --when the test is failed...
  --hilite start of loop (test is failed this time)
  set theHotLine = 5
  hilite line theHotLine of field "codeFragmentFLD"
  --show a "test failed/loop terminates" message
  beep 2
  hilite line 1 of field "varWatchFLD"
  beep 2
  set the text of field "testConWatchFLD" = "Loop condition failed - end repeat"
  hilite line 1 of field "testConWatchFLD"
  --hilite end of loop
  set theHotLine = 8
  hilite line theHotLine of field "codeFragmentFLD"
  --hilite end of handler
  set theHotLine = theHotLine + 1
  hilite line theHotLine of field "codeFragmentFLD"
end replaceNumberAnim 1
--NB Core of this handler is still the simple "repeat while..." loop.

```

Reducing the size of the type and using a proportional font allows us to include more comments but the code is still very crowded on the page.

It is also hard to tell comments from statements. Clearly, use of bold and italic type and more white space could make things better.

The example on the next page takes the repeat loop and explores these possibilities.

The final example looks at how the complete movie script from which this handler came might be tackled.

```
--handler definitions---
--Animating a loop with display of changing loop testCondition variable value,
--changing Cast selection as loop executes, and hilited code line in handler fragment.
```

on **replaceNumberAnim 1**

```
--preparation
global initH, incrementH, newH, theHotLine, myNum
--record castmember telltale initial position
--(and set up other variables for later)
```

```
--hilite entering the handler
```

```
--hilite initial setting of test variable
set myNum = 2
put myNum into word 6 of field "varWatchFLD"
repeat while myNum <= 6
  --hilite start of loop
  set theHotLine = 5
  hilite line theHotLine of field "codeFragmentFLD"
  --hilite setting of castNum to variable
  set theHotLine = theHotLine + 1
  hilite line theHotLine of field "codeFragmentFLD"
  --hilite first cast member in sequence
  set the locH of sprite 12 = newH
  set newH = (newH + incrementH)
  updateStage
  --do Nth frame of the animation
  set the castNum of sprite 8 = myNum
  updateStage
  --hilite incrementing of test variable
  set theHotLine = theHotLine + 1
  hilite line theHotLine of field "codeFragmentFLD"
  set myNum = myNum + 1
  put myNum into word 6 of field "varWatchFLD"
  --hilite end of loop
  set theHotLine = theHotLine + 1
  hilite line theHotLine of field "codeFragmentFLD"
```

**end repeat**

```
--when the test is failed...
--hilite start of loop (test is failed this time)
set theHotLine = 5
hilite line theHotLine of field "codeFragmentFLD"
--show a "test failed/loop terminates" message
beep 2
hilite line 1 of field "varWatchFLD"
beep 2
set the text of field "testConWatchFLD" = "Loop condition failed - end repeat"
hilite line 1 of field "testConWatchFLD"
--hilite end of loop
set theHotLine = 8
hilite line theHotLine of field "codeFragmentFLD"
--hilite end of handler
set theHotLine = theHotLine + 1
hilite line theHotLine of field "codeFragmentFLD"
```

end **replaceNumberAnim 1**

```
--NB Core of this handler is still the simple "repeat while..." loop.
--See the movies in Program Visualisation folder for original code.
--Note that new tab stops have been set to give deeper indentation in the code.
```

Use of bold and italic type and more white space has certainly improved the readability of the code.

The comments above the loop indicate that we could define these operations as separate handlers and call them with one word from here. Comments above and below the handler help establish its purpose and derivation: we can make notes to ourselves and others as we go along.

Compare this approach with that taken in *Algorithm Development*.

**MOVIE to explain a REPEAT WHILE... loop (animated explanation)***HeaderNotes*

This one breaks all the rules about using cast names not numbers.  
If the movie script is put in the number one slot (my usual choice),  
the loop breaks!

Apr 95/Mar 96

**--initialisation**on **startMovie**

*--set stageColor to white*

set the stageColor = 0

*--ID the movie*

put the movieName into field "movieNameFLD"

*--set up the animation puppets*

puppetSprite 8, TRUE

—the animated "number" bitmaps

set the castNum of sprite 8 = 1

—the "start button" bitmap

put 0 into word 6 of field "varWatchFLD"

—the "rest" value of "myNum"

set the text of field "testConWatchFLD" = "Loop ready to execute"

end startMovie

on **stopMovie**

*--tidyUp routines to write*

end stopMovie

**--handler definitions**

*--Version 1*

*--Using REPEAT WHILE... loop to switch castmembers of a sprite.*

on **replaceNumber1**

*--NB use of "magic numbers" - move the cast around and the code breaks!*

set myNum = the castnum of sprite 8

**repeat while** myNum <= 6

set the castNum of sprite 8 = myNum

wait 10

updatestage

set myNum = myNum + 1

**end repeat**

wait 10

set the castNum of sprite 8 to 1

end replaceNumber1

*--Call handler from Script channel for endless cycling*

*--and from puppet channel for clickable action.*

*--parameterised handler expects an integer for wait in ticks*

on **wait** ticks

startTimer

**repeat while** the timer < 1 \* ticks

*--waiting for the timer*

**end repeat**

end wait

*--handler definitions end*

*TrailerNotes*

Currently exploring three different ways of animating:

1. "repWithVis" type, using animated text & flowDiagram
2. The old "repWhileVis" method, using static text + watcher fields.
3. This movie type, trying to combine actual animation with animated Cast window and hilited code.

*Last modified 18 Mar 96*

tools. In a standard Director environment, text fields can easily be adapted to hold and display the changing contents of variables *etc.* Lingo has a rich set of text manipulation elements and the technique is illustrated in several *Program Visualisation* movies.

### *Animating*

This is perhaps the most ambitious method, requiring large measures of technical expertise, system resources and creative understanding of animation. It is potentially the most interesting method and work is still proceeding at Coventry on developing visualisations for tutorial purposes. The method is employed in various ways in the *Program Visualisation* movies.

Director is a good tool for producing animation. This animation is not usually produced with the intention of making a Lingo program clear, (though it is of course the *output* of a program) but there is no reason why it should not be used for this purpose.

### **Visualising data**

Understanding the data that a program is handling is often the key to writing the program in the most effective way. Director has a great aid to visualising data in the form of the cast window: nothing less than a very flexible visual data management system. If anything, it is slightly too flexible: I have seen many supposedly tutorial movies where castmembers are referred to only by number in Lingo scripts and are scattered about all over the cast window.

In these materials, you will find some effort has been put into giving the castmembers meaningful names and organising them into functional groups. These names are then used in scripts. Study the sample movies and devise your own ways of organising your data - it is worth it in the long run!

### **Summary**

Methods of representation that make use of pictures or animation are obviously of particular interest to members of the art and design community. They are likely to respond well to any form of visual communication and should in turn have much to contribute to the development of visualisation methods.

## **Fixing broken programs**

---

### **Why do programs break?**

Programs break because programmers are human! Common faults in programs are:

1. Faulty logic  
Even quite small Director projects can require complicated scripts and even if the scripts themselves are simple, it can be hard to follow the interaction between them. It is therefore quite easy to introduce faulty logic.
2. Faulty coding  
Coding requires a knowledge of language syntax and accurate typing, as well as a grasp of program logic. It is therefore quite easy to introduce faulty coding.
3. Failure to allow for unusual conditions or user error  
This can be hard to spot as the problem condition may only arise occasionally. Like faulty logic, it is a design failure.

All of these faults are common even in professionally-written programs. Expect to make many mistakes: learning from them is all part of the fun of learning to program in Lingo.

### **Tools for fixing programs**

Director 5 introduced several tools to help you fix broken programs. Prior to this version, the only way to troubleshoot was to use the TRACE facility in the Message window, which often produced so much output it was hard to decipher, or insert tracer code in your scripts, which was labour-intensive. The most important new tools are the Debugger and the Watcher but the Memory and Text Inspectors are also useful at times.

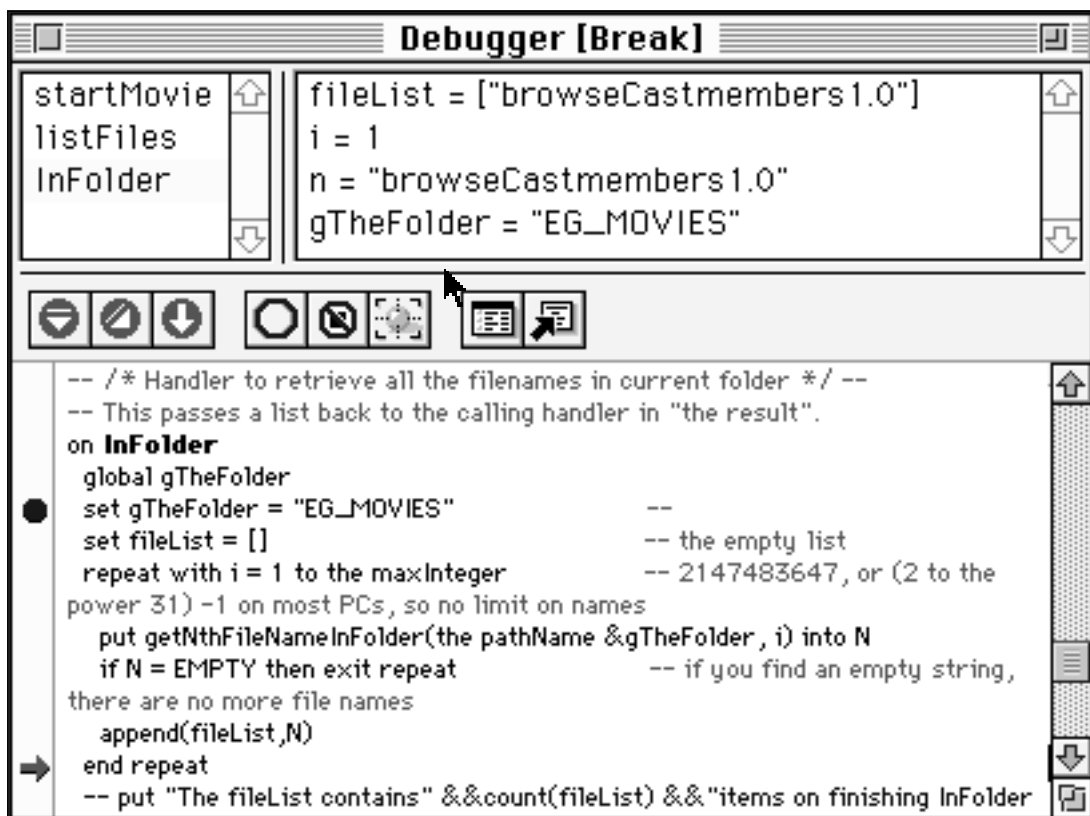
#### *The debugger*

The debugger bears a marked similarity to that incorporated in MetroWerks' CodeWarrior programming environment. Details of appearance and operation are included in *Appendix One - Update* and, of course, the Director documentation and help system.

Essentially, a debugger is simply another computer program which takes control of your own code and allows you to step through it while watching the values of expressions and variables. You decide where to interrupt the execution of your program by setting what are known as 'breakpoints' and then step through the code from there.

It is actually sensible to employ the debugger while developing your program, that is, *before* it breaks. You will then be forced to consider the logic of your program when trying to decide where to set breakpoints. You will also learn a great deal about any unnecessary loops or branches in your code; see expressions or variables go outside their intended range, and so on.

If you don't employ the debugger during development your programs will almost certainly break at some time. You will then be given the option of calling up the debugger or simply examining your code. The latter method often results in many hours spent staring at your monitor and mentally going round in circles.



Debugger window showing breakpoint (spot), current line (arrow), handler chain and value of variables and expressions

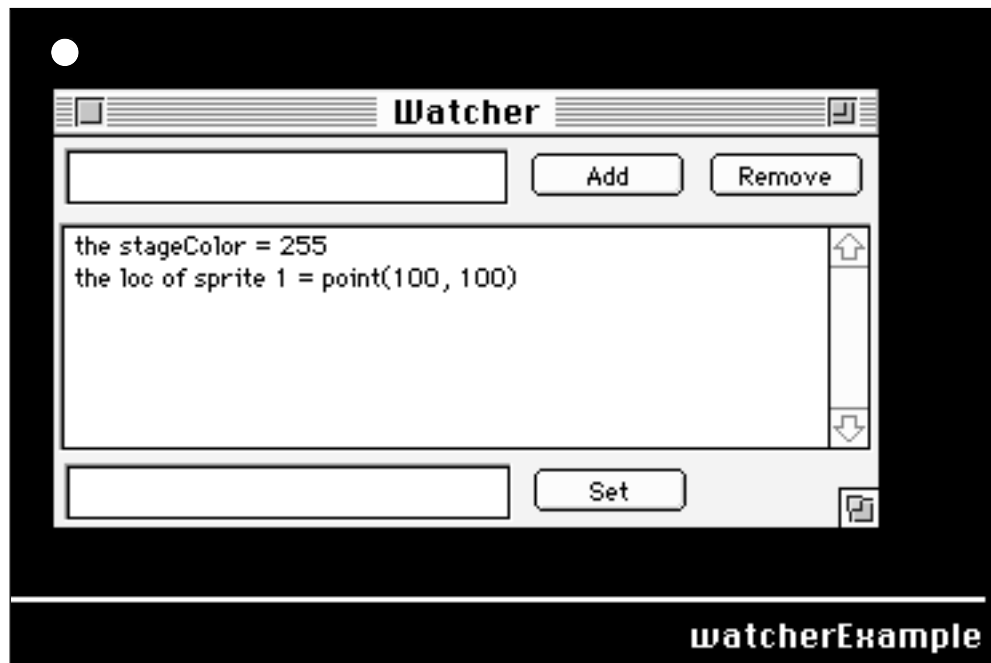
Watching too many variables can decrease Director's response noticeably. For example, the cursor may blink slowly or windows may resize sluggishly. Reducing the amount of data the watcher window must continuously update will immediately improve Director's performance level.

### *The watcher window*

The Watcher window shows the values of simple expressions and variables in your movie's scripts. The variable or expression appears at the left of the window followed by an equal sign (=) and the expression or variable's current value.

If Director can't obtain the value of an expression or variable in the current context, the term "<void>" appears to the right of the equal sign. Director updates values in the watcher window when the user steps through lines of a script while in debug mode or continuously while the movie plays. Some variables, such as the `time` or the `mouseH` are updated even while the movie is not playing.

The Watcher window is often preferable for simpler fault-finding tasks since it occupies much less of the screen.



*Watcher window showing value of stageColor (255 = black) and location of sprite (white spot at 100, 100)*

Examine the following movies:

`D5_MOVIES:debuggerExample`

`D5_MOVIES:watcherExample`

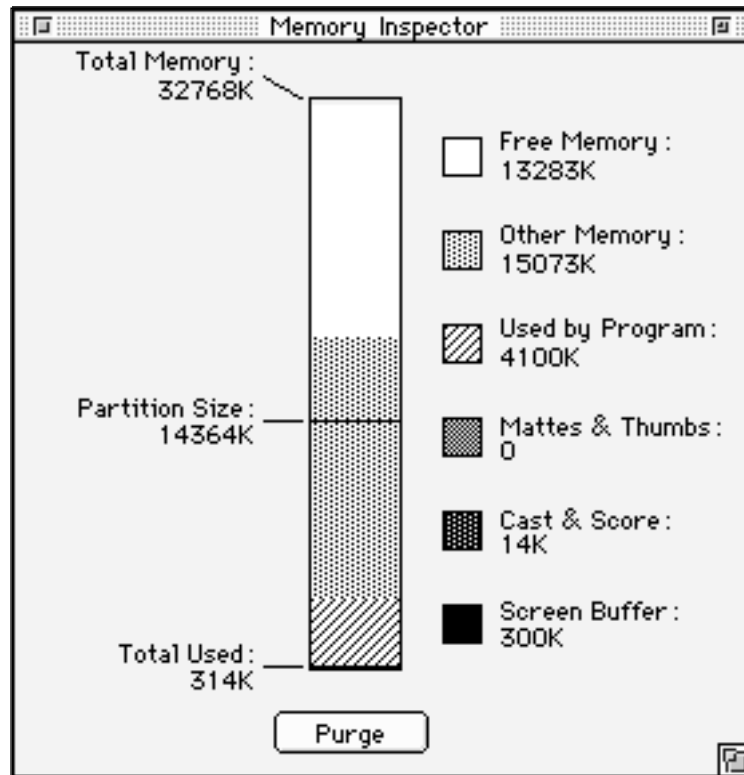
for examples of the respective tools in use.

### *The memory inspector*

The memory inspector displays information about how much memory is available to Director for your movie. It also indicates how much memory different parts of the current movie use and the total disk space the movie occupies.

Not all broken programs are the result of faulty code. Some common faults in programs, for example playing multiple sounds, occur because there is too little memory available. It is therefore

useful to be able to inspect the amount of memory available, if only to eliminate shortage of memory as a cause of failure.



*The Memory Inspector window on machine with 32Mb RAM*

Total Memory	shows you the total memory available in your system. This number depends on the amount of RAM and any System 7 virtual memory.
Physical Memory	indicates the memory the amount of RAM available on your computer. This is available only when virtual memory is on.
Free Memory	indicates how much more memory is currently available in your system.
Other Memory	indicates the amount of memory taken up by the system and by other applications.
Used by Program	indicates the amount of memory used by Director.
Mattes & Thumbs	shows how much memory is used by cast members that use the Matte ink in the score and by thumbnail images in the cast window.

Cast & Score	indicates the amount of memory used by the cast members in the cast window and the notation in the score window. Cast members include all the artwork in the paint window, all the text in the text windows, and any sounds, palettes, buttons, digital video movies, or linked files imported into the cast and currently loaded into memory.
Screen Buffer	shows how much memory Director reserves for a “working area” while animating on the stage.
Partition Size	shows the amount of memory allocated to Director in the Get Info box. Available if Temporary Memory is enabled.
Total Used	indicates how much RAM is being used for a movie.

The ‘Purge’ button removes all purgeable items from RAM, including all thumbnail images in the cast window. All cast members that have Unload (purge priority) set to a priority other than “0-Never” (as specified in the Cast Member Properties dialog box) are removed from memory. This is useful for gaining as much free memory as possible before importing a large file. Edited cast members don’t get purged.

#### *Fields on stage*

The introduction of new tools should not blind you to the value of using fields on stage to record the value of variables and expressions or to track progress through your programs. Sometimes, this is more convenient than using the Debugger or Watcher. The process also tests your understanding of your code. It may also suggest elements of the user interface for your project. Most of the project movies, in particular those in *D5\_MOVIES* demonstrate some aspect of this technique.

#### **Summary**

Fixing broken programs is an unavoidable part of programming. While always something of a chore, new tools in Director 5 (and more are promised in Director 6) make it easier than it used to be for Lingo users. The process should be seen as an invaluable part of learning to program in Lingo because of the increased insight into program logic and grasp of Lingo syntax and conventions which it brings.

Much of the text in this section comes from the Director QuickHelp system. Macromedia copyright gratefully acknowledged.

## **Example Movies**

---

### **What's in the movies?**

You will find examples of Lingo programming applied to controlling sprites, controlling sound, acting conditionally on the basis of user choices, acting conditionally on the basis of the position of the cursor, soliciting and processing user input of text and so on.

The movies come from several different sources: some were originally built to demonstrate solutions and techniques to students, others came from quick fixes to problems in student movies, and some are from students themselves. The movies therefore vary quite a lot in style and they do not all embody the best practice encouraged in this handbook.

### **Further work**

You are encouraged to examine the samples with a critical eye and to make improvements wherever you can. In my view, this is the best way to learn how to program in Lingo, provided you also acquire an understanding of the principles of good programming.

This is essential because it often seems like too much trouble to do things “properly”. While the “quick and dirty” solution is fine for a small and well-defined problem, it will not suffice for any substantial Lingo-based project. Unfortunately, the Director/Lingo environment almost encourages unsound practice. It is a highly interactive and media-rich environment, so novices are tempted to rush into coding without proper planning. As with all design, there is no one right solution to a programming problem but this is almost always the wrong approach.

Resist the temptation. Check out some of the sources in the *Annotated Bibliography* before getting too deeply into Lingo itself. Avoid turning your programming project into a house of cards, waiting to collapse as soon as you add one more element.

At the time of writing, Macromedia are reportedly extending Lingo to cover their whole product range (Common Lingo). A sound understanding of Lingo programming should therefore lead to good employment prospects, to say nothing of increased creative control of Director and other Macromedia products.

## Annotated Bibliography

---

### On Lingo:

My local branch of Blackwells listed eight books dealing with Lingo on 24 April 1997: a very different situation from that obtaining at the time Edition 1.0 was released. Note that the supplied Director documentation is essential for reference, though less useful in a tutorial context.

MACROMEDIA, 1996?  
*Lingo for Director 5*  
Macromedia Inc., San Francisco.  
£42.98.

CALLERY, M., 1996.  
*Learning Lingo*  
Addison-Wesley, Reading.  
£29.50.

MACROMEDIA, 1997.  
*Lingo Authorized*  
Macromedia Inc., San Francisco.  
£31.95.

JULIUS T., date unknown  
*Lingo!*  
New Riders.  
£41.99.

SMALL, P., 1996.  
*Lingo Sorcery*  
John Wiley & Sons, Chichester.  
£24.95. *Excellent but idiosyncratic treatise on object-oriented programming in Lingo. Especially good on working with lists.*

THOMPSON, J.T. & GOTTLIEB, S., 1996.  
*Macromedia Director Lingo Workshop (2nd ed.)*  
Hayden Books, Indianapolis.  
Mid-price at around £50.00.  
*Very useful tutorial and reference material - I still use the first edition. CD packed with example movies (arranged in Chapter order).*

G/MATTER, 1995. (formerly Gray Matter Design)  
*The MediaBook CD for Director*  
Gray Matter Design, San Francisco.  
Expensive at around £250.00.  
*Tutorial and interactive dictionary, plus development tools.*

**On-line help and information:**

There are WWW sites at:

<http://www.macromedia.co/director/>

<http://www.mcli.dist.maricopa.edu/director/>

which, for those with web browsers, have taken over from:

`direct-1`

All are useful and others appear from time to time. Be prepared for congestion and slow response from the WWW sites and a flood of incoming postings from the mailing list.

A Web document titled "Director Scripting for Authors and Artists" by Gary Rosenzweig appeared at:

<http://www.csn.net/~rosenz/lingo.html>

This is organised as thirty-two short chapters, which can be downloaded and formatted for print to make a document of sixty pages. There are no sample movies but plenty of good code fragments. Check it out - I believe it is also available in print.

A very active developer and distributor of Director Xtras is g/matter Inc. Their site is:

<http://www.gmatter.com/>

and they maintain a mailing list called `xtras -1`. Details and subscription instructions are available from the g/matter website.

A technician at Coventry School of Art & Design, Mark Peden, has developed a number of interesting 'artificial life' animations and a colour conversion palette for Director users. See his work at:

<http://www.csad.coventry.ac.uk/~mep/amoebic.lifeforms/start.html>

<http://www.csad.coventry.ac.uk/~mep/new.forms/colorwall.html>

<http://www.csad.coventry.ac.uk/~mep/tools/palette.html>

Addresses of other sites of interest may be distributed from time to time if there is enough demand. Email me:

[iphillips@patrol.i-way.co.uk](mailto:iphillips@patrol.i-way.co.uk)

about your favourite site(s), books, or other support materials.

**On programming in general:**

APPLEMAN, D., 1994.

*How Computer Programming Works*

Ziff-Davis, Emeryville, Ca.

*Highly visual approach but sometimes rather confusing.*

CHANTLER, 1981.

*Programming Techniques and Practice*

National Computing Centre, Manchester.

*Old and with terrible graphics but very clear summary of methods of representing algorithms and programs.*

HAIGH, J., 1995.

*Designing Computer Programs*

Edward Arnold, London.

*Business-oriented but up to date, fairly visual, and English.*

KNUTH, D., 1984 [2].

“Literate Programming” in

*Computer Journal*, 27 No. 2, May 1984, p109.

*Classic stuff on making programs readable and intelligible.*

KERNIGHAN, B. & PLAUGER, P., 1978.

*The Elements of Programming Style*

McGraw-Hill, New York.

*More good stuff on making programs readable and intelligible.*

*Like KNUTH, it does not date.*

MEEK, N., HEATH, P. & RUSHBY, N., 1983.

*Guide to Good Programming Practice*

Ellis Horwood, Chichester.

*A good companion to CHANTLER. Not visual.*

PATTERSON, D., KISER, D. & SMITH, N., 1989.

*Computing Unbound: Using Computers in the Arts and Sciences*

W.W. Norton, New York

*Despite title does not contain much of direct reference to the visual arts but very good on algorithm development.*

PHILLIPS, I., 1996.

“Computer program visualisation in art and design computing” in *Proceedings of Eurographics UK 96, London.*

*A review of various visualisation methods. Expands on themes contained in the “Program Visualisation” section of the project handbook.*

**On object-oriented programming:**

BLAIR, G., GALLAGHER, J., HUTCHISON, D., and SHEPHERD, D., 1991.

*Object-Oriented Languages, Systems and Applications*  
Pitman, London.

*Very clear and thorough primer in OOP.*

BOOCH, G., 1991.

*Object Oriented Design, With Applications*  
Benjamin/Cummings, Redwood City, CA.

*Authoritative and well-written. Pretty much everything you need to know but dense and therefore hard going in some places. One of the best parts is the section on Applications, with lengthy worked examples. These have been updated for the new edition now out: at first glance, the originals look better in many respects.*

For Lingo OOP, there is quite a lot of stuff available from the on-line sources on parent and child scripts, ancestry, birthing etc.

## **Appendix One - Update**

---

### **Summary of Lingo-related changes in Director 5**

Director 5 introduced many changes and new features. The most relevant features from the point of view of Lingo programming are:

- New, changed, and outdated Lingo
- The introduction of Debugger and Watcher windows
- Enhancements to the Message and Script windows
- Multiple casts
- XObjects superseded by Xtras
- Improved Help system
- Lingo FAQs (Frequently Asked Questions) in the Help

These features are summarised below and dealt with in more detail in the printed and online Director documentation. Illustrations and the reference material in this Appendix are taken from the Director QuickHelp screens: copyright is acknowledged where necessary.

#### *New, changed, and outdated Lingo*

There are over 100 additions or changes to Lingo in Director 5, including a new logic structure (CASE, a multi-way branch) and new ways of controlling QuickTime sprites. It would have been a near-impossible task to predict the way all these changes might have to be confronted in developing Director projects, so I have chosen to highlight those changes I consider of most relevance to the novice Lingo programmer.

There have also been improvements in the Lingo programming environment, listed in this section and dealt with in more detail in *Fixing Broken Programs*, which speed the process of coding and testing.

*The following elements are new in Director 5.0 or have had functionality added since Director 4.0:*

activeWindow	lineCount of member
autoTab of member	linePosToLocV
	lineSize of member
beginRecording	loc of sprite
border of member	locToCharPos
boxDropShadow	locVToLinePos
boxType of member	loop of member
buttonType	
	margin of member
cancelIdleLoad	media of member
case	member
the castLibNum of sprite	memberNum of sprite
center of member	
changeArea of member	name of CastLib
channelCoun of member	new
charPosToLoc	number of CastLib
chunkSize of member	number of castLibs
clearFrame	number of members of castLib
crop of member	
	on activateWindow
deleteFrame	on closeWindow
desktopRectList	on moveWindow
digitalVideoTimeScale	on resizeWindow
digitalVideoType of member	on rightMouseDown
dropShadow of member	on rightMouseUp
duplicate(list)	on zoomWindow
duplicateFrame	openWindow
duration of member	otherwise
editable of member	pageHeight of member
emulateMultiButtonMouse	paletteMapping
end case	pattern
endRecording	paletteRef
	the platform
fileName of castLib	preLoadMode of CastLib
filled of member	preLoadMovie
finishIdleLoad	
frameLabel	rect of member
framePalette	rightMouseDown, the
frameScript	rightMouseUp, the
frameSound1	
frameSound2	sampleRate
frameTempo	sampleSize
frameTransition	save castLib
frontWindow	score
	scoreSelection
height of member	scriptsEnabled
	scriptType
idleHandlerPeriod	scrollByLine
idleLoadDone	scrollByPage
idleLoadMode	scrollTop of member
idleLoadPeriod	selection of castLib
idleLoadTag	setCallBack
idleReadChunkSize	shapeType
insertFrame	sound of member
keyPressed	timeScale of member
	trackCount(member)

trackCount(sprite)	trackType (member)
trackEnabled	trackType (sprite)
trackNextKeyTime	transitionType of member
trackNextSampleTime	type of member
trackPreviousKeyTime	
trackPreviousSampleTime	unloadMovie
trackStartTime(member)	updateFrame
trackStartTime(sprite)	updateLock
trackStopTime(member)	
trackStopTime(sprite)	windowPresent
trackText	wordWrap of member

### *Lingo that has changed in version 5.0*

The following terms have been revised to keep terminology clear now that Director has multiple casts. The older terms are still supported, but they will become obsolete and should be avoided:

#### **Director 4.0 Term**

backColor of cast  
 cast  
 castmembers  
 castNum of sprite  
 castType of cast  
 center of cast  
 controller of cast  
 crop of cast  
 depth of cast  
 duplicate cast  
 duration of cast  
 erase cast  
 fileName of cast  
 foreColor of cast  
 frameRate of cast  
 height of cast  
 hilite of cast  
 loaded of cast  
 loop of cast  
 modified of cast  
 move cast  
 name of cast  
 number of cast  
 number of castmembers  
 palette of cast  
 picture of cast  
 preLoad of cast  
 preLoadCast  
 purgePriority of cast  
 scriptText of cast  
 size of cast  
 sound of cast  
 text of cast  
 textAlign of field  
 textFont of field  
 textHeight of field  
 textSize of field  
 textStyle of field  
 video of cast  
 width of cast

#### **Director 5.0 Term**

backColor of member  
 member  
 number of members  
 memberNum of sprite  
 type of member  
 center of member  
 controller of member  
 crop of member  
 depth of member  
 duplicate member  
 duration of member  
 erase member  
 fileName of member  
 foreColor of member  
 frameRate of member  
 height of member  
 hilite of member  
 loaded of member  
 loop of member  
 modified of member  
 move member  
 name of member  
 number of member  
 number of members  
 palette of member  
 picture of member  
 preLoad of member  
 preLoadMember  
 purgePriority of member  
 scriptText of member  
 size of member  
 sound of member  
 text of member  
 alignment of member  
 font of member  
 lineHeight of member  
 fontSize of member  
 fontStyle of member  
 video of member  
 width of member

*Lingo that is outdated in version 5.0*

The following elements are obsolete and no longer supported:

```
birth
instance
factory
closeDA
openDA
when...then constructs
```

*The introduction of Debugger and Watcher windows*

Developing and debugging Lingo programs was always hampered by the need to insert code to signal entry to or exit from parts of the program and/or output the values of variables and expressions.

A reasonable debugger has been added in Director 5 which allows the setting of breakpoints and line by line execution of scripts: both extremely helpful when trying to fix broken programs. A separate Watcher window can be set up to track the values of variables and expressions. See the illustrations taken from QuickHelp on Pages 55 and 56.

*Enhancements to the Message and Script windows*

The Message and Script windows now include:

- Alphabetical listing of Lingo commands
- Categorised listing of Lingo commands
- Trace button (Message window only)
- Go To Handler button (Message window only)
- Breakpoint toggle button (Script window only)
- Button to call up Watcher window
- Button to recompile all scripts (Script window only)

**Message window (Window menu)**

Command-M



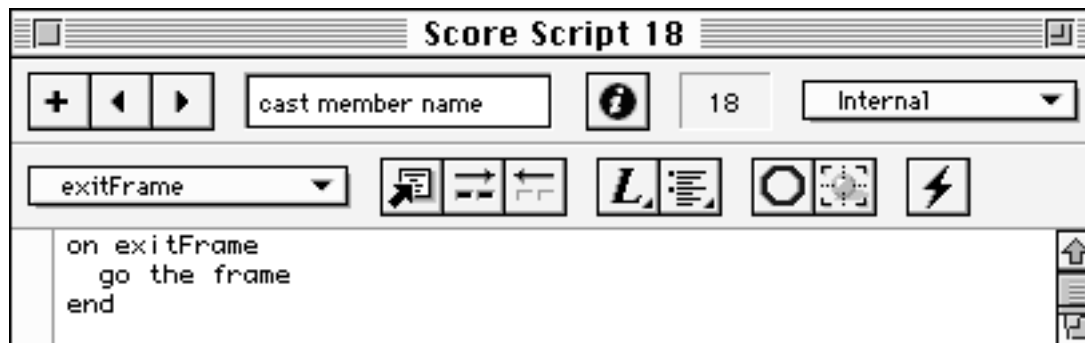
The message window is a convenient place to experiment with and test Lingo scripts. Actions occur immediately when you press the Return key, so you can see the results before you insert your scripts into a movie. This allows you to see the results of any script, including whether it is a valid script.

For descriptions of the tools on the message toolbar, see the [Message toolbar](#) topic.



### Script window (Window menu) Command-0

Use the script window to enter and edit Lingo scripts. A script can contain up to 32K of text.



For a description of the buttons at the top of the script window, see the [cast window](#) topic.

For descriptions of the tools on the script toolbar, see the [Script toolbar](#) topic.

**Tip** The help topics for [Lingo](#) commands include examples that you can paste into your scripts and use.

Director saves changes you make in the script window when you click anywhere outside of the window, close it, click the Previous or Next buttons to go to a different script, or if you choose Recompile All Scripts in the Control menu.

**Tip** Double-clicking a cell in the script channel opens the script window.



[Script Cast Member Properties](#)  
[Debugger](#)  
[Message](#)  
[Watcher](#)



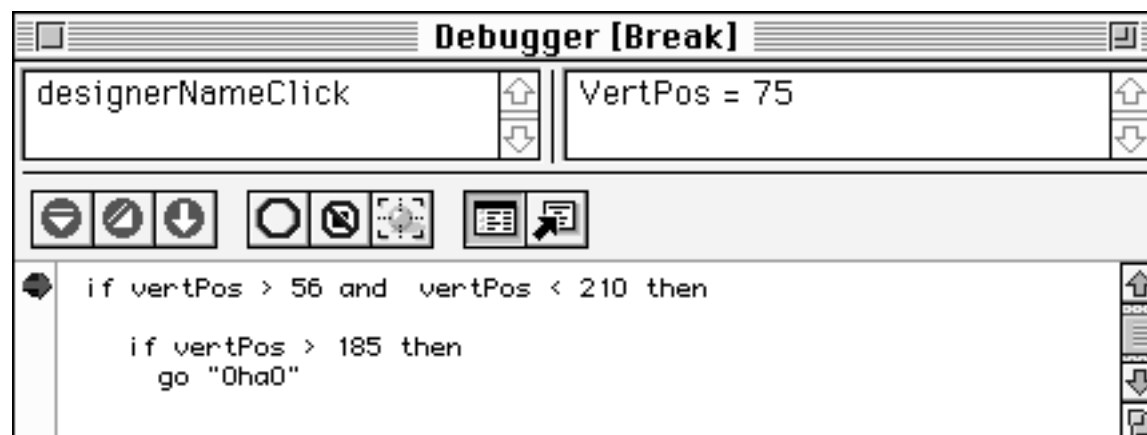
## Debugger window (Window menu)

Command-`

The debugger helps with troubleshooting. The window helps to locate and correct bugs in lingo scripts. It includes several tools that let you:

- See the current line of Lingo
- Run the current handler line by line
- Track the sequence of handlers that were called as part of getting to the current handler
- Display the value of any local variable, global variable, or property related to the Lingo that you're investigating
- Open related windows such as the watcher window and script window.

Click part of the illustration for more information:



Any of the following actions opens the debugger window:

- Choosing Debugger from the Window menu
- Encountering a breakpoint in a script
- Clicking Debugger in an error message that appears when Director encounters a syntax error in a script.

You can't edit the script directly in the debugger; you must return to the script window.



[Script](#)  
[Message](#)  
[Watcher](#)

**Watcher window (Window menu) Command-Shift-`**

The Watcher window shows the values of simple expressions and variables in the movie's scripts.

The variable or expression appears at the left of the window followed by an equal sign (=) and the expression or variable's current value. If Director can't obtain the value of an expression or variable in the current context, the term "<void>" appears to the right of the equal sign. Director updates values in the watcher window when the user steps through lines of a script while in debug mode or continuously while the movie plays. Some variables, such as the time or the mouseH are updated even while the movie is not playing.

**Note:** Watching too many variables can decrease Director's response noticeably. For example, the cursor may blink slowly or windows may resize sluggishly. Reducing the amount of data the watcher window must continuously update will immediately improve Director's performance level.

**To add variables or expressions to the list in the watcher window by selecting them in the script window:**

1. Open a script window in which the variable or expression appears.
2. Select the variable or expression.
3. Click the Watch Expression button at the top of the script window.



Director adds the selected variables or expressions to the list in the watcher window and also displays those changes in the variable pane.

**To change the value of a variable or expression:**

1. Select the value or expression in the watcher window.
2. Type the new value in the field next to the Set button.
3. Click Set.

**To add variables or expressions to the list directly from the watcher window:**

1. Type the variable or expression in the field to the left of the Add button.
2. Click Add.

The variable or expression appears in the list.

**To remove a variable or expression:**

1. Select the variable or expression in the watcher window.
2. Click Remove.

### *Multiple casts*

You can create multiple casts for your Director 5.0 movies. There are two types of casts, internal and External. Working with cast members within the cast window is the same regardless of whether the cast is internal or external.

Prior to version 5.0 of Director, all casts were internal and only one cast was allowed per movie. The only way to share castmembers was to create a movie called `SHARED.DIR`, which was empty apart from the Cast. Members of this cast could be shared with other movies in the same folder. This was a clumsy mechanism and multiple casts are a great improvement.

See the movies:

```
D5_MOVIES:multipleCasts_01
```

```
D5_MOVIES:multipleCasts_02
```

for example usage. See also Page 60 for more detailed information on multiple casts.

### *XObjects superseded by Xtras*

Xtras are software modules that extend Director's functionality. Previously, such extensions were known as XObjects. Xtras are stored in one of two places and appear under the "Xtras" item on the menubar.

They are one of the types of XLibrary (external code library) supported by Director. Six Xtras and three Xlibs are included as standard with Director 5, providing facilities for reading/writing text files; printing; and managing databases as well as some other tasks.

### *Improved Help system*

The QuickHelp system in Director 5 is faster and more comprehensive than the system in Director 4. Help screens can be copied as text or pictures and used elsewhere, as has been done in this handbook. See the next page for an illustration of the main Lingo help screen.

### *Lingo FAQs (Frequently Asked Questions) in the Help*

There are five rather arbitrary questions about Lingo in the FAQ section of the Help system, as well as some Lingo-related questions scattered about the other sections. See Page 59 for an illustration.



Click a letter (above) to view Lingo elements alphabetically.

Or select one of the categories below to see a list of relevant Lingo elements:

<a href="#">Cast members</a>	<a href="#">Movie control</a>
<a href="#">Casts</a>	<a href="#">Movie in a window</a>
<a href="#">Code structures &amp; syntax</a>	<a href="#">Navigation</a>
<a href="#">Computer &amp; monitor</a>	<a href="#">New Lingo elements</a>
<a href="#">Digital video</a>	<a href="#">Operators</a>
<a href="#">External files</a>	<a href="#">Parent scripts</a>
<a href="#">Fields</a>	<a href="#">Puppets</a>
<a href="#">Frames</a>	<a href="#">Score generation</a>
<a href="#">Interface elements</a>	<a href="#">Sprites</a>
<a href="#">Lingo that has changed in 5.0</a>	<a href="#">Sound</a>
<a href="#">Lingo that is outdated</a>	<a href="#">Strings</a>
<a href="#">Lists</a>	<a href="#">Time</a>
<a href="#">Math &amp; logical operators</a>	<a href="#">User interaction</a>
<a href="#">Memory management</a>	<a href="#">Variables</a>

For general information on using [Lingo](#), see the [Lingo Basics](#) topic.

The Lingo menu appears when you click and hold the Lingo button in the script window. This menu displays the complete set of Lingo commands that you can use to create scripts for your movie.

Choosing an element from the Lingo menu enters it into a script at the insertion point. This saves you from typing the command and also eliminates typing.

**Note:** You can use the Copy Topic Text command from the Help file's Edit menu to copy Lingo examples from the Help topics; then you can paste the example into the Director script window and modify it for your use.



## FAQs – Lingo



How do I find the movie script?



How do I use a custom cursor in Director?



What is a mask cast member, and how do I make one?



I have a `rollOver` test in a frame that works properly, but when I jump to a new frame, that `rollOver` area is still being evaluated even though the sprite is no longer there. This also happens with the `cursor of sprite` property.



Where is a list of `keyCodes`?

### Understanding internal and external casts

The multiple cast mechanism is a great improvement over the `SHARED.DIR` mechanism used prior to Director 5.

#### *Internal casts*

When you create a new movie, Director automatically creates an internal cast. Internal casts are stored inside the movie file. When you save a movie, all internal casts are saved. When you create a projector, they are stored inside the projector file. Internal casts cannot be shared by other movies.

#### *External casts*

External casts are stored outside of the movie file. They can be shared between movies or serve as libraries for commonly used movie elements. They are also useful for distributing work in a project team.

When you create an external cast by choosing New Cast from the File menu, you have the choice of linking or not linking the cast to the current movie.

- If you link an external cast to the current movie, Director opens the cast every time you open the movie. If it can't find the cast in the original location, it prompts you to locate the cast file. When you save a movie, all linked external casts are saved as well. The first time you save a movie with a linked external cast file, Director prompts you to enter a file name and choose a location for the file.
- If you don't link an external cast to a movie, you must open it separately with the Open command on the File menu, and save it by activating the window and choosing Save from the File menu. It is not saved along with the movie when you choose Save. If you move a cast member to the stage or score from an unlinked external cast, Director prompts you to link the cast to the current movie.

You can link and unlink existing casts to the current movie with the Movie Casts command on the Modify menu.

#### *Moving members between casts*

Multiple casts provide a convenient way of moving members around between otherwise unrelated movies or projects. Members can be 'OPTION-dragged' from one cast to another to copy them, or normal COPY and PASTE procedures can be used. Members can also be deleted in the usual way. **Remember** to save any unlinked external cast(s) as soon as operations are completed.

Remember to see the following in the D5\_MOVIES folder:

multipleCasts\_01  
multipleCasts\_02

for example usage of multiple casts.

**Xtras**

Xtras are software modules that extend Director's functionality. An Xtra file can contain one or more such modules. There are four types of Xtras:

- Transition Xtra cast members, which supply transitions in addition to predefined transitions available in the Frame Properties:Transition dialog box. After they are used in the score's transition channel, they appear in the cast window the same as any cast member.

An Xtra transition cast member can have its own custom properties, properties dialog box, animated thumbnail, cast window icon, and About box. Open the dialog box that sets properties by opening the Frame Properties:Transition dialog box and then clicking Options.

- Xtra cast members, which can be a wide range of objects such as databases, text managers, or special graphics. They appear in the Insert menu after the Xtra is loaded. An Xtra can create more than one menu item if it is designed to do so.

Add an Xtra cast member to a cast by choosing the Xtra from the Insert menu. Cast member Xtras are sometimes called sprite Xtras because they can be assigned to the score after they are in the cast window.

An Xtra cast member can have its own custom properties, properties dialog box, media editor, animated thumbnail, cast window icon, and About box. Open the dialog box that sets the Xtra's properties by opening the cast member's Properties dialog box and then clicking Options. Open the Xtra's media editor by double-clicking the cast member's thumbnail in the cast window.

- Lingo Xtras, which add Lingo elements to Director's built-in Lingo.
- Tool Xtras, which you can use during authoring. To open a tool Xtra, choose it from the Xtras menu.

*Providing Xtras*

When Director launches, it automatically registers Xtras that are in either of two places:

- The Xtras folder in the same folder that contains the Director application or projector
- One of the following folders, depending on which platform Director is running on:

For Windows 95 and Windows NT, the Xtras folder:

`Program Files\Common Files\Macromedia\Xtras`

For Windows 3.1, the folder:

`Windows\Macromed\Xtras`

For Macintosh and Power Macintosh, the folder:

`System Folder:Macromedia:Xtras`

To make an Xtra available, place its file in one of these folders before you launch Director. (The Xtra can be in a folder within the Xtras folder up to five layers deep.) Director also automatically closes these Xtras when the application quits.

You can also open Lingo Xtras after Director is running by using the `openXlib` command. The Lingo Xtra can be in any folder if you open it this way. However, you must use the `closeXlib` command to close the Xtra after Director is finished with it.

Xtras aren't packaged in projectors. The Xtras must be in an Xtras folder in the same folder as the projector or an Xtras folder that is valid for the current operating system.

If an Xtra that the movie uses is missing, an alert appears when the movie or external cast file opens. For missing Xtra transition cast members, the movie performs a simple cut transition instead. For other missing Xtra cast members, Director displays a red "X" on the stage as a placeholder for the missing Xtra.

Copies of the same Xtra can have different filenames or have the same filename but reside in different folders. If they are used in the same movie, Director detects that such Xtras are duplicates and displays an alert. You can avoid this situation by just deleting any duplicate Xtras if this occurs.

#### *Creating Xtras by using the new function*

You can create a new instance of an Xtra by using the new function. The specific way you do this depends on the Xtra's type.

You can create new transition Xtras and cast member Xtras just as you can built-in cast members. For transition cast members, use `new` and the symbol `#transition`. Other cast member Xtras have their own symbols specified by the developer.

For example, a QuickDraw 3D cast member could be given the symbol `#quickDraw3D`. In this case, to create a new cast member, you'd use the statement `new(#quickDraw3D)`.

This statement creates a new instance of the Xtra cast member which has the symbol `#math`:

```
new(#math)
```

After the cast member is created, you can assign it content the same way as you do for other castmembers.

You create new instances of Lingo Xtras by using the new function and the term `xtra` as the first parameter. For example, this statement creates a new instance of the Lingo Xtra `stringReader`:

```
set string1 = new(xtra "stringReader")
```

For instances of Lingo Xtras created by using the new function, you must set the variable that contains the Xtra to 0 before you use the `closeXlib` command to delete the Lingo Xtra.

See the Director 5 documentation on "Authoring from Lingo" for more information about creating cast members from Lingo.

*Checking which Xtras are available*

Lingo can tell you how many Xtras are available, the name of each, and what each Xtra contains.

The number of xtras property indicates how many Xtras are available in the current movie.

The name of xtra property determines the name of a specific Xtra. The name of xtra property can be tested and set.

For example, the following repeat loop displays the name of each Xtra in the message window:

```
repeat with counter = 1 to (the number of xtras)
  put the name of xtra counter
end repeat
```

The showXLib command displays each Xtra file and its contents. For example, suppose that a Lingo Xtra Friends is in the folder c:\Xtra Reserve. If the Xtra file Friends contains the modules Fred and Joe, the showXLib command would give the following results:

```
Director:Xtras:Friends
xtra Fred
xtra Joe
```

Use mMessageList to display message with information about the Xtra. For example, the statement put mMessageList(xtra "Fred") displays information about the Xtra Fred.

*For more information on Xtras*

For a listing of Xtras available for Director, see the Macromedia Web site at:

```
http://www.macromedia.com
```

You will also find information on the Xtra Developer's Kit (XDK). The g/matter website:

```
http://www.gmatter.com
```

is also worth visiting (see the *Annotated Bibliography*). The XtraNet™ Xtra will be of particular interest to those wishing to connect a Director project to the Internet. Novice programmers will however usually need help installing Xtras and writing the appropriate scripts and handlers.