

Hands On AGK BASIC

**A Beginner's Guide to Multi-Platform
Games Programming**

Alistair Stewart

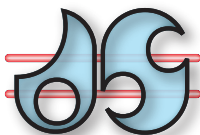


Digital Skills

Hands On AGK BASIC

A Beginner's Guide to Multi-Platform Games Programming

Alistair Stewart



www.digital-skills.co.uk
tel: +44(0)1465 861 638

Digital Skills

Milton

Barr

Girvan

Ayrshire

KA26 9TY

United Kingdom

Copyright © 2012-2013 Alistair Stewart

All rights reserved.

No part of this work may be reproduced or used in any form without the written permission of the author.

Although every effort has been made to ensure accuracy, the author and publisher accept neither liability nor responsibility for any loss or damage arising from the information in this book.

AGK BASIC is produced by The Game Creators Ltd.

Cover Design: Sébastien Leroux

Printed June 2012
Updated February 2013
ebook Updated April 2013

Title: Hands On AGK BASIC

ISBN: 978-1-874107-14-9

Other Titles Available:

Hands On DarkBASIC Pro Vols 1 & 2
Hands On Milkshape

Table of Contents

Foreword	i
Preface	iii
Acknowledgements	iii
How to Get the Most Out of this Book	iv

Chapter 1 - Algorithms

Designing Algorithms	2
Following Instructions	2
Control Structures	3
Sequence	3
Selection	4
Complex Conditions	10
Iteration	14
Data	20
Levels of Detail	22
Checking for Errors	26
Summary	29
Solutions	32

Chapter 2 - Starting AGK

Programming a Computer	36
Introduction	36
The Compilation Process	36
Summary	38
Starting AGK	39
Introduction	39
Starting Up AGK	39
The Program Code	42
Transferring Your App to a Tablet or Smartphone	43
Summary	44
First Statements in AGK BASIC	45
Introduction	45
Print()	45
Adding Comments	47
PrintC()	47
Other Statements which Modify Output	48
Summary	52
The Second Source File	54
A Splash Screen	55
Starting a New Project	56
App Window Properties	57
Measurements	57
Summary	59
Solutions	61

Chapter 3 - Data

Program Data	64
Introduction	64
Constants	64
Variables	64
Named Constants	68
Summary	69
Allocating Values to Variables	70
Introduction	70
The Assignment Statement	70
The Print() Statement Again	77
Acquiring Data	79
User Input	87
Summary	90
Testing Sequential Code	91
Solutions	93

Chapter 4 - Selection

Binary Selection	98
Introduction	98
if	98
The Other if Statement	107
Summary	108
Multi-Way Selection	109
Introduction	109
Nested if Statements	109
The select Statement	112
Testing Selective Code	115
Summary	117
Solutions	118

Chapter 5 - Iteration

Iteration	124
Introduction	124
The while .. endwhile Construct	124
The repeat .. until Construct	126
The for..next Construct	128
Finding the Smallest Value in a List of Values	133
The exit Statement	134
The do .. loop Construct	135
Nested Loops	135
Nested for Loops	136
Testing Iterative Code	137
Summary	139
Solutions	140

Chapter 6 - A First Look at Resources

Resources - A First Look	146
Introduction	146
Images	146
Images in AGK	149
Sound	156
Music	159
Detecting User Interaction	163
Text Resources	165
Later	170
Summary	170
Solutions	172

Chapter 7 - Spot the Difference Game

Game - Spot the Difference	176
Introduction	176
Game Design	176
Game Code	182
Solutions	188

Chapter 8 - User-Defined Functions

Functions	192
Introduction	192
Functions	192
Parameters	196
Summary	206
BASIC Subroutines	207
Introduction	207
Creating a Subroutine	207
A Library of Functions	209
Introduction	209
Creating a Library	209
Creating Modular Software	211
Introduction	211
Top-Down Programming	212
Bottom-Up Programming	219
Structure Diagrams	221
Summary	222
Solutions	224

Chapter 9 - String and Math Functions

String Functions	232
Introduction	232
String-Handling Functions	232
Creating Your Own String Functions	242
Summary	248

Math Functions	250
Introduction	250
Coordinates	250
Trigonometric Functions	251
Other Math Functions	259
Summary	262
Solutions	265

Chapter 10 - Arrays

Arrays	270
Problems with Simple Variables	270
One Dimensional Arrays	271
Using Arrays	276
Dynamic Arrays	293
The undim Statement	294
Multi-dimensional Arrays	294
3-Dimensional Arrays and Higher	295
Arrays and Functions	296
Summary	296
Solutions	297

Chapter 11 - Data Types and Operators

Data Storage	304
Introduction	304
Declaring Variables	304
Type Definitions	305
Summary	310
Data Manipulation	311
Introduction	311
Other Number Systems	311
Shift Operators	312
Bitwise Boolean Operators	314
A Practical Use For Bitwise Operations	317
Summary	318
Solutions	320

Chapter 12 - File Handling

Files	324
Introduction	324
Accessing Files	324
File Management	330
Folder Management	331
Zip Files	335
Summary	336
Solutions	338

Chapter 13 - Particles

Particles	342
Introduction	342
Creating Particles	342
Retrieving Particles Data	355
Summary	359
Solutions	361

Chapter 14 - Text

Text	366
Introduction	366
Review	366
Further Text Statements	367
Text Character Statements	375
Summary	385
Solutions	389

Chapter 15 - User Input

Virtual Buttons	392
Introduction	392
Virtual Button Statements	392
Using Multiple Virtual Buttons	397
Summary	399
Keyboard Input	400
Introduction	400
Text-Input Statements	400
Summary	404
Edit Box Statements	405
Introduction	405
Edit Box Statements	405
Summary	418
Joystick Input	421
Introduction	421
Virtual Joystick Statements	421
Physical Joysticks	427
Summary	430
Device Dependent Input	432
Introduction	432
Accelerometer Statements	432
Mouse Statements	435
Joystick Statements	437
Keyboard Statements	440
Device Identity	442
Summary	442
Solutions	444

Chapter 16 - Images

Images	450
Introduction	450
Review	450
Further Image Statements	450
The ImageJoiner Utility	455
Atlas Texture Files and Proportional Fonts	456
Manipulating Images	457
Image Selection from Storage	460
Using a Device's Camera	461
Mapping Images to Sprites	463
Summary	466
Solutions	468

Chapter 17 - Sprites

Sprites	470
Introduction	470
Review	470
Other Sprite Statements	471
The Sprite Offset Feature	494
Sprite Bounding Areas	499
Sprite Groups	505
Moving Sprites	511
Controlling Speed	523
Ray Casting	524
Summary	532
A Jigsaw Puzzle Game	535
Introduction	535
The Game	535
The Data Files	535
Game Layout	536
The Game Code	537
Solutions	541

Chapter 18 - Animated Sprites

Introduction	550
Using an Animated Sprite	550
A Card Trick	556
Summary	558
An Asteroid Game	560
Introduction	560
Game Layout	560
Game Logic	561
Game Resources	561
Game Code	561
Solutions	573

Chapter 19 - Screen Handling

Screen Handling	580
Introduction	580
Screen-Related Statements	580
Zooming and Scrolling	583
Touch Statements	595
Summary	602
Secrets of Sync()	604
Summary	608
Solutions	609

Chapter 20 - Physics

Sprite Physics - 1	614
Introduction	614
Basic Physic Statements	614
Physics Collisions	628
Physics Sprite Shapes	630
Summary	634
World Physics	636
Introduction	636
General Statements	636
Forces	638
Summary	641
Sprite Physics - 2	643
Contacts	643
Physics Groups and Categories	648
Physics Ray Casting	653
Summary	656
Joints	658
Introduction	658
Joint Statements	658
Summary	683
Solutions	685

Chapter 21 - Accessing a Network

Multiplayer Games	692
Introduction	692
Hardware Requirements	692
The Host and its Clients	692
Multiplayer Statements	693
Summary	716
Multi-Player Tic Tak Toe	718
Introduction	718
Game Logic	718
Program Code	719
HTTP	727
Introduction	727

HTTP Statements	727
Summary	736
Solutions	738

Chapter 22 - Bits and Pieces

Date and Time	748
Introduction	748
Standard Date Statements	748
Unix Date Statements	749
Time Statements	751
Summary	752
QR Coding	753
Introduction	753
QR Code Statements	753
Summary	755
Advertising	756
Introduction	756
Ad Statements	756
Summary	757
Errors	759
Introduction	759
Error Handling Statements	759
Summary	760
Benchmarking	761
Introduction	761
Benchmarking Statements	761
Summary	765
Paused Apps	766
Solutions	769

Chapter 23 - 3D Graphics

Concepts and Terminology	772
Introduction	772
Modelling Ideas and Terminology	776
Summary	783
Creating a First 3D App	786
Introduction	786
Statements	786
User Control of the Camera	790
Summary	792
Object Creation and Modification	793
Creating Primitives	793
Object Appearance	798
Transforming Objects	803
Cameras	816
Introduction	816
Camera-Related Statements	816
Using Camera Commands to Create First Person Perspective	823
Billboarding	828

Summary	829
Lights	831
Introduction	831
Directional Lights	831
Point Lights	833
Object Reflectivity	835
Summary	836
Collisions	
Introduction	837
Ray Cast Statements	837
Summary	855
Other 3D Related Statements	857
Converting Between Screen and 3D Coordinates	857
Sprite and 3D Depth Settings	863
The Depth Buffer	863
Shaders	866
Quaternion Rotation	869
Summary	872
Solutions	873

Chapter 24 - Memory Blocks

Accessing Memory	884
Introduction	884
Memory Block Statements	885
Storing Characters and Strings in a Memory Block	891
Using a Memory Block as an Array	893
Using a Memory Block as a Record Structure	895
Saving Memory Block Data to a File	904
Summary	908
Memory Blocks for Images	910
Introduction	910
Memory Block Image Statements	910
Mapping a Pixel to a Memory Block	912
Modifying an Image's Data	913
Creating Your Own Images from Scratch	914
Summary	916
Creating a Mandelbrot Image	918
Producing the Program	920
Zooming In	927
Shortcomings	930
Solutions	932

Chapter 25 - Drawing

Drawing Statements	940
Drawing a Line	940
Drawing a Dot	940
Drawing a Rectangle	941
Drawing a Circle	943
Drawing an Ellipse	945

Creating a Data Structure for Basic Shapes	947
Summary	952
Drawing a Simple Bezier Curve	953
Introduction	953
Calculating the Curve	953
Creating a Bezier Curve in Real Time	960
Displaying 3D Models in Wireframe	966
Introduction	966
Developing the Program Logic	968
Implementing the Program	969
Solutions	975
Appendix A - ASCII Codes	939
Index	940

Foreword

by Lee Bamber

When I was nine I received my first personal computer, a VIC-20, which was blessed with over 3K of system memory and a maximum palette of 16 colours. From that moment my universe was slightly larger than the amount of memory it takes to store this paragraph of text. In that universe I created lost civilisations, space battles, deep treks into inhospitable lands and dangerous creatures ready leap out from every dark corner. Granted most of it happened in the imagination of the player, but my audience consisted of my parents, my brothers and my uncle who all thought my ‘games’ were amazing.

What was truly amazing was the rate at which the limits of my universe expanded with more memory, more colours, more speed and a bigger audience to play my ‘games’. We went from back-bedroom build-your-own hobby developers to a global industry worth Billions, and it happened so quickly we still have the original founders of this industry working alongside the newest recruits.

Veteran fogies like me can look back and see so much history that when something new comes along, we can almost instantly compare it to five things it strongly resembles from our own fading recollections. We can also identify when something is utterly game-changing, and it usually happens on an epic scale. For me, that moment was when the term ‘apps’ entered the public consciousness. Before then you had software you went out and bought, because you needed software. When the idea of an ‘app’ emerged, it gave ‘software’ a name change and a leviathan marketing budget to spend to the end of time. We are no longer a community of developers who write software, we’re a community that creates solutions to make life better, and its consumers, not developers, who are deciding what those should be.

Here in lies the problem for us poor, overworked developers. We had our plate full just writing software that worked sufficiently for a period of time on one computer. Now we have to create solutions for everyone, where-ever they are, when-ever they want to use it and what-ever they are using as a ‘computer’ at the time. People today want to use their favourite ‘app’ on their home computer, their phone, their TV, in their car and on their fancy new touch tablet, and they want it instantly and constantly up to date. It’s enough to make you cry!

In the best tradition of software developers, whenever we face an emergent system that requires an impossible amount of resources, we simply change the system. Why have ten developers working on ten different systems when you can have one developer working on a single system, and then have a cleverer system translate that work to the other nine automatically. Sounds great in theory, but the practical application produces a number of very oddly shaped solutions indeed.

Now what if you could spin the time machine forward a few years and grab one of the nicer solutions to this problem and then zip back to the present day and start using it? Well it just so happens that I do have a time machine and did just that. It seems, The Game Creators Ltd of 2015 ‘will be’ working with a new piece of software called AGK (App Game Kit) and they ‘will make’ me promise that providing I don’t upset causality, I can take an early copy back with me to 2011 to help them omega test it. Call it a moment of weakness, but I might have put this copy of the product on a website at www.appgamekit.com.

Apparently the break-through with AGK is that you can develop an app on one

system, and it will be instantly compatible with every other system on the planet. I've only managed to get it working on Windows, Mac, MeeGo, iOS, Android and Bada at the moment, but with some more tweaking of their strange alien code I 'will be' assured I can get it to produce all the other platforms present on Earth, even the ones that don't exist yet.

AGK uses the concept of universal commands. That is, each command will perform the same functionality no matter which system it happens to be running on. It is also input agnostic, so if your application requires an input source that does not exist, AGK will virtualise that input data from another piece of hardware present on the device or emulate it through virtual controls. The result is that you can write an 'app' just once, and the resulting program will run on any device present today and any device in the future too.

As developers we have a few decades of history under our belt and can swell with pride on what we have achieved to date. My prediction is that we've just created the world's largest rod for our backs, and now have to finish what we started. The only way forward is to evolve ten pairs of hands through a fortuitous genetic mutation, or find a solution that lets us meet the demands of the next few decades with confidence, a sense of fun and above all, ten fingers!

Lee Bamber
CEO The Game Creators Ltd
2012

Preface

Welcome to the amazing world of the App Game Kit. This is an application that will allow you to create a program that you can design on one machine and run on just about any other platform.

Want to write a game that will run on your phone or your tablet? No problem! Write the application on your regular computer and transfer it to your other devices - it's easy!

Graphics, animation, sound, touch screen, mouse, joystick, keyboard - your app will cope with them all.

Write your apps and sell them online. Some game apps have sold over 5 million copies.

And although AGK stands for App Game Kit, there's no reason why your creation has to be a game. You can easily write educational material, utilities or any number of applications.

Who is this book for? It's for you. It doesn't matter if you're a programming guru or have never written a line of code in your life. This book assumes only a basic knowledge of computers. If you can run an application, copy, paste, delete data, access the internet, type (even with just one finger), and know just a little basic arithmetic then that's all that assumed. Everything else is here. And for the guru there are plenty of hints and tips that I'm sure you will find helpful.

Some books can be very hard going: pages and pages of detail - most of which you forget as soon as you turn to the next page, or when you fall asleep. We do things differently here. No getting bored reading page after page - you'll have a series of activities to carry out that are designed to reinforce what you've read on the page. And unlike most other books that seem to forget about any tasks they have set you, you'll find a full set of answers to the activities at the end of each chapter.

Enjoy your journey through this book.

Acknowledgements

I'd like to thank Lee Bamber, Paul Johnston and Mike Johnson from The Game Creators for all their help and guidance, Also, thanks to John McKay for his patience and forbearance in testing every example included in the book. As usual, Virginia Marshall did her best to rid the book of any grammar or spelling problems.

As always, any errors remaining are entirely my own.

I am always happy to receive any helpful suggestions on how to improve the book or - heaven forbid - details of any errors you've found.

Contact me at alistair@digital-skills.co.uk.

Alistair Stewart June 2012

Second Edition

It's an almost impossible task to write an up-to-date book on a language that changes as rapidly as AGK. Until now we've published updates and extra chapters to extend the original Hands On AGK BASIC to deal with the many changes and additions of AGK version 1, but now, with AGK version 2 and another swathe of additional and updated commands, I've taken this opportunity to revamp the whole book.

The main change is that the publication has now been split into two volumes, with the more advanced topics such as 3D and networking commands moved to the second volume. But I've also taken the opportunity to make minor corrections to retained text and to check that the sample programs run correctly on the latest version of AGK.

As always, please feel free to email me with any useful suggestions or corrections.

Alistair Stewart April 2014

How to Get the Most Out of this Book

Is learning the basics of computer programming difficult? No, but you do have to put in the effort. Despite other publications promising to have you expert in a day, or a week, I'm sure you're smart enough to know that's not going to happen. So, let's get real: you'll learn how to program using AGK if you put in the work, take your time to make sure you understand something before moving on, and practice, practice, practice.

We've tried to keep things interesting by giving you plenty of practical work to do as you journey through this book, but feel free to try out your own projects as well.

The first chapter is the only one in which you won't need your computer since it concentrates on the basic concepts behind all computer programming. You can, if you wish, work on the second chapter at the same time as you read through Chapter 1. That way, you'll be able to start programming right away.

Take your time with each chapter. Make sure you do each of the activities: they are there to give you a deeper understanding as well as to keep you actively involved. Since most activities require you to create a program, the computer will let you know if you've got it right, but you should still take the time to look at the activity's solution given at the end of the chapter. The solution given may differ from your own but it's always of use to see how others tackle the same problem.

Don't be afraid to reread a section or a whole chapter - it's the second or third reading of something new that finally gets the information across to most people.

If you are already a seasoned programmer you will be able to skip through much of the early chapters. If you have programmed in DarkBASIC before, many of the core statements in AGK are identical to that earlier language, but look out for a few subtle differences such as the lack of READ and DATA statements and the method used to initialise arrays.

The Files for the Book

Many of the programming activities (particularly in later chapters) make use of other resources such as images, sounds, and 3D models. You can download the necessary files from

www.digital-skills.co.uk/downloads/AGK2Downloads.zip

1

Algorithms

In this Chapter:

- Understanding Algorithms
- Creating Algorithms
- Control Structures
- Boolean Expressions
- Data Types
- Stepwise Refinement
- The Need for Testing

Designing Algorithms

Following Instructions

Activity 1.1

Carry out the following set of instructions in your head.

- Think of a number between 1 and 10
- Multiply that number by 9
- Add up the individual digits of this new number
- Subtract 5 from this total
- Think of the letter at that position in the alphabet
- Think of a country in Europe that starts with that letter
- Think of a mammal that starts with the second letter of the country's name
- Think of the colour of that mammal

Congratulations! You've just become a human computer. You were given a set of instructions which you have carried out (by the way, did you think of the colour grey?).

That's exactly what a computer does. You give it a set of instructions, the machine carries out those instructions, and that is ALL a computer does. If some computers seem to be able to do amazing things, that is only because someone has written an amazingly clever set of instructions. A set of instructions designed to perform some specific task (like that in Activity 1.1) is known as an **algorithm**.

A clear and concise algorithm should have the following characteristics:

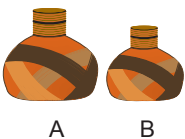
- One instruction per line
- Each instruction is unambiguous
- Each instruction is as brief as possible

Activity 1.2

This time let's see if you can devise your own algorithm.

The task you need to solve is to measure out exactly 4 litres of water. You have two containers. Container A, if filled, will hold exactly 5 litres of water, while container B will hold 3 litres of water. You have an unlimited supply of water and a drain to get rid of any water you no longer need. It is not possible to know how much water is in a container if you only partly fill it from the supply.

If you manage to come up with a solution, see if you can find a second way of measuring out the 4 litres.



As you can see, there are at least two ways to solve the problem given in Activity 1.2. Is one better than the other? Well, if we start by filling container A, the solution needs less instructions, so that might be a good guideline at this point when choosing which algorithm is best.

However, the algorithms that a computer carries out are not written in English like

the instructions shown above, but in a more stylised form using a computer **programming language**. AGK BASIC is one such language. The set of program language instructions which make up each algorithm is then known as a **computer program** or **software**.

Just as we may perform a great diversity of tasks by following different sets of instructions, so the computer can be made to carry out any task for which a program exists.

A traditional disk makes use of a magnetic surface to record information. More recent designs use solid state memory.

Computer programs are normally **copied** (or **loaded**) from a disk into the computer's memory and then **executed** (or **run**). Execution of a program involves the computer performing each instruction in the program one after the other. This it does at impressively high rates, possibly exceeding 160,000 million (or 160 billion) instructions per second (160,000 **mips**).

Depending on the program being run, the computer may act as a word processor, a database, a spreadsheet, a game, a musical instrument or one of many other possibilities. Of course, as a programmer, you are required to design and write computer programs rather than use them. And, more specifically, our programs in this text will be mainly multimedia and game oriented, an area of programming for which AGK has been specifically designed.

Activity 1.3

- a) A set of instructions that performs a specific task is known as what?
- b) What term is used to describe a set of instructions used by a computer?
- c) The speed of a computer is measured in what units?

Control Structures

Although writing algorithms and programming computers can be complicated tasks, there are only a few basic concepts and statements which you need to master before you are ready to start producing software. Luckily, many of these concepts are already familiar to you in everyday situations. If you examine any algorithm, no matter how complex, you will find it consists of only three basic structures:

- **Sequence** where one instruction follows on from another.
- **Selection** where a choice is made between two or more alternative actions.
- **Iteration** where one or more instructions are carried out over and over again.

These structures are explained in detail over the next few pages. All that is needed is to formalise how they are used within an algorithm. This formalisation better matches the structure of a computer program.

Sequence

A set of instructions designed to be carried out one after another, beginning at the first and continuing, without omitting any, until the final instruction is completed, is known as a **sequence**. For example, instructions on how to perform an everyday task such as plant a bush in the garden would be:

- Choose spot for planting
- Dig hole
- Add fertiliser
- Place shrub in hole
- Refill hole

The set of instructions given earlier in Activity 1.1 is also an example of a sequence.

Activity 1.4

Re-arrange the following instructions to describe how to play a single shot during a golf game:

- Swing club forwards, attempting to hit ball
- Take up correct stance beside ball
- Grip club correctly
- Swing club backwards
- Choose club

Selection

Binary Selection

Often a group of instructions in an algorithm should be carried out only when certain circumstances arise. For example, if we were playing a simple game with a young child in which we hide a sweet in one hand and allow the child to have the sweet only if she can guess which hand the sweet is in, then we might explain the core idea with an instruction such as

Give the sweet to the child if the child guesses which hand the sweet is in

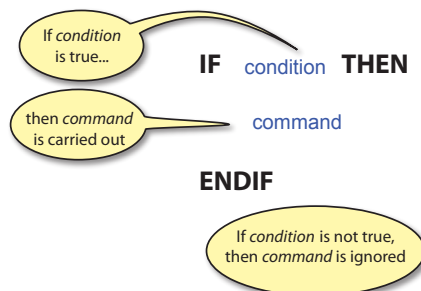
Notice that when we write a sentence containing the word IF, it consists of two main components:

- a condition : the child guesses which hand the sweet is in
- and
- a command : give the sweet to the child

A **condition** (also known as a **Boolean expression**) is a statement that is either true or false in a given situation. The command given in the statement is only carried out if the condition is true at that particular moment and hence this type of instruction is known as an **IF** statement and the command as a **conditional instruction**. Although English would allow us to rewrite the above instruction in many different ways, when we produce a set of formal instructions, as we are required to do when writing algorithms, then we use a specific layout as shown in FIG-1.1, always beginning with the word IF.

FIG-1.1

The IF Statement



Note that there are two alternative actions in this structure: to carry out the command or to ignore it.

Notice that the layout of this instruction makes use of three terms that are always included. These are the words IF, which marks the beginning of the instruction; THEN, which separates the condition from the command; and finally, ENDIF which marks the end of the instruction.

The indentation of the command is important since it helps our eye grasp the structure of our instructions. Appropriate indentation is particularly valuable in aiding readability once an algorithm becomes long and complex. Using this layout, the instruction for our game with the child would be written as:

```
    IF the child guesses which hand the sweet is in THEN
        Give the sweet to the child
    ENDIF
```

Sometimes, there will be several commands to be carried out when the condition specified is met. For example, in the game of Scrabble we might describe a turn as:

```
    IF you can make a word THEN
        Add the word to the board
        Work out the points gained
        Add the points to your total
        Select more letter tiles
    ENDIF
```

Of course, the IF statement will almost certainly appear within a longer set of instructions. For example, the instructions for playing our guessing game with the young child may be given as:

```
    Hide a sweet in one hand
    Ask the child to guess which hand contains the sweet
    Wait for the child to reply
    IF the child guesses which hand the sweet is in THEN
        Give the sweet to the child
    ENDIF
    Ask the child if they would like to play again
```

Note that this algorithm does not explicitly say what happens when the child makes an incorrect guess. This is because no specific action needs to be carried out when an incorrect guess is made.

This longer list of instructions highlights the usefulness of the term ENDIF in separating the conditional command, Give the sweet to the child, from subsequent unconditional instructions, in this case, Ask the child if they would like to play again.

Activity 1.5

A simple game involves two players. Player 1 thinks of a number between 1 and 100, then Player 2 makes a single attempt at guessing the number. Player 1 responds to a correct guess by saying *Correct*. If the guess is incorrect, Player 1 makes no response. The game is then complete and Player 1 states the value of the number.

Write the set of instructions necessary to play the game. In your solution, include the statements:

```
    Player 1 says "Correct"
    Player 1 thinks of a number
    IF guess matches number THEN
```

The IF structure is also used in an extended form to offer a choice between two alternative actions. This expanded form of the IF statement includes another formal term, ELSE, and a second command. If the condition specified in the IF statement is true, then the command following the term THEN is executed, otherwise the

command following ELSE is carried out.

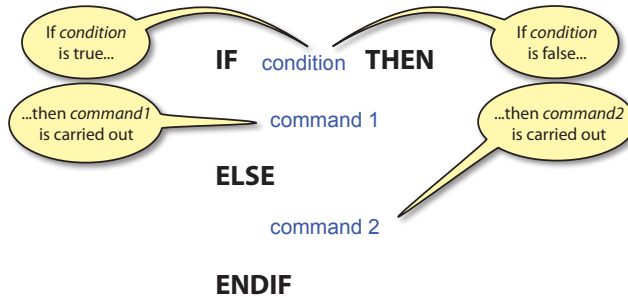
For instance, in our earlier example of playing a guessing game with a child, nothing happened if the child guessed wrongly. If the person holding the sweet were to eat it when the child's guess was incorrect, we could describe this setup with the following statement:

```
IF the child guesses which hand the sweet is in THEN
    Give the sweet to the child
ELSE
    Eat sweet yourself
ENDIF
```

The general form of this extended IF statement is shown in FIG-1.2.

FIG-1.2

The IF..THEN..ELSE Structure



Activity 1.6

In the game of Hangman, one player has to guess the letters in a word known to the second player. At the start of the game, player 2 draws one hyphen for each letter in the word. When player 1 guesses a letter which is in the word, player two writes the letter above the appropriate hyphen. When an incorrect letter is guessed, player 2 draws a body part of a hanging man (there are 6 parts in the simple drawing).

Write an IF statement containing an ELSE section which describes the alternative actions to be taken by player 2 when player 1 guesses a letter.

In the solution include the statements:

- Add letter at appropriate position(s)
- Add part to hanged man

When we have several independent selections to make, then we may use several IF statements. For example, when playing Monopoly, we may buy any unpurchased property we land on. In addition, we get another turn if we throw a double. This part of the game might be described using the following statements:

```
Throw the dice
Move your piece forward by the number indicated
IF you land on an unsold property THEN
    Buy the property
ENDIF
IF you threw doubles THEN
    Throw the dice again
ELSE
    Hand the dice to the next player
ENDIF
```

Because this form of the IF statement (with or without the ELSE option) always

offers two alternative actions, the structure is known as **binary selection**.

Multi-way Selection

Although a simple IF statement can be used to select one of two alternative actions, sometimes we need to choose between more than two alternatives (known as **multi-way selection**). For example, imagine that the rules of the simple guessing game mentioned in Activity 1.5 are changed so that there are three possible responses to Player 2's guess; these being:

- Correct
- Too low
- Too high

One way to create an algorithm that describes this situation is just to employ three separate IF statements:

```
IF the guess is equal to the number you thought of THEN
    Say "Correct"
ENDIF
IF the guess is lower than the number you thought of THEN
    Say "Too low"
ENDIF
IF the guess is higher than the number you thought of THEN
    Say "Too high"
ENDIF
```

This will work, but would not be considered a good design for an algorithm since, when the first IF statement is true, we still go on and check if the conditions in the second and third IF statements are true. Checking those last two statements would be a waste of time since, if the first condition is true, the others cannot be and therefore testing them serves no purpose. Where only one of the conditions being considered can be true at a given moment in time, these conditions are known as **mutually exclusive** conditions. The most effective way to deal with mutually exclusive conditions is to check for one condition, and only if this is not true, do we bother to examine the other conditions being tested. So, for example, in our algorithm for guessing the number, we might begin by writing:

```
IF guess matches number THEN
    Say "Correct"
ELSE
    ***Check the other conditions***
ENDIF
```

Of course a statement like Check the other conditions is too vague to be much use in an algorithm (hence the asterisks to emphasise the problem). But what are these other conditions? They are the guess is lower than the number Player 1 thought of and the guess is higher than the number Player 1 thought of.

We already know how to handle a situation where there are only two alternatives: use an IF statement. So selecting between *Too low* and *Too high* requires the statement

```
IF guess is less than number THEN
    Say "Too low"
ELSE
    Say "Too high"
ENDIF
```

Now, by replacing the phrase *****Check the other conditions***** in our original algorithm with our new IF statement we get:

```
IF guess matches number THEN
    Say "Correct"
ELSE
    IF guess is less than number THEN
        Say "Too low"
    ELSE
        Say "Too high"
    ENDIF
ENDIF
```

Notice that the second IF statement is now totally contained within the ELSE section of the first IF statement. This situation is known as **nested IF statements**. Where there are even more mutually exclusive alternatives, several IF statements may be nested in this way. However, in most cases, we're not likely to need more than two nested IF statements.

Activity 1.7

In an old TV programme called *The Golden Shot*, contestants had to direct a crossbow in order to shoot an apple. The player sat at home and directed the crossbow controller via the phone. Directions were limited to the following phrases: up a bit, down a bit, left a bit, right a bit, and fire.

Write a set of nested IF statements that determine which of the above statements should be issued.

Use statements such as:

```
IF the crossbow is pointing too high THEN
and
    Say "Left a bit"
```

As you can see from the solution to Activity 1.7, although nested IF statements get the job done, the general structure can be rather difficult to follow. A better method would be to change the format of the IF statement so that several, mutually exclusive, conditions can be declared in a single IF statement along with the action required for each of these conditions. This would allow us to rewrite the solution to Activity 1.7 as:

```
IF
    crossbow is too high:          Say "Down a bit"
    crossbow is too low:           Say "Up a bit"
    crossbow is too far right:     Say "Left a bit"
    crossbow is too far left:      Say "Right a bit"
    crossbow is on target:         Say "Fire"
ENDIF
```

Each option is explicitly named (ending with a colon) and only the one which is true will be carried out, the others will be ignored.

Of course, we are not limited to merely five options; there can be as many as the situation requires.

When producing a program for a computer, all possibilities have to be taken into account. Early adventure games, which were text based, allowed the player to type a command such as *Go East*, *Go West*, *Go North*, *Go South* and this moved the player's character to new positions in the imaginary world of the computer program. If the

player typed in an unrecognised command such as Go North-East or Move faster, then the game would issue an error message. This setup can be described by adding an ELSE section to the structure as shown below:

```

IF
  command is Go East:
    Move player's character eastward
  command is Go West:
    Move player's character westward
  command is Go North:
    Move player's character northward
  command is Go South:
    Move player's character southward
ELSE
  Display an error message
ENDIF

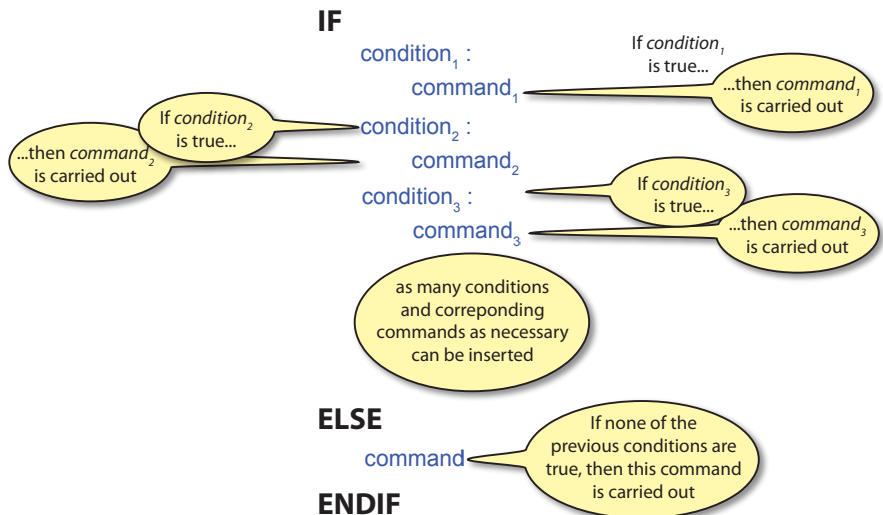
```

The additional ELSE option will be chosen only if none of the other options are applicable. In other words, it acts like a catch-all, handling all the possibilities not explicitly mentioned in the earlier conditions.

This gives us the final form of this style of the IF statement as shown in FIG-1.3.

FIG-1.3

The Multi-Way IF Structure



Activity 1.8

In the TV game Wheel of Fortune (where you have to guess a well-known phrase), you can, on your turn, either guess a consonant, buy a vowel, or make a guess at the whole phrase.

If you know the phrase, you should make a guess at what it is; if there are still many unseen letters, you should guess a consonant; as a last resort you can buy a vowel.

Write an IF statement in the style given above describing how to choose from the three options.

Complex Conditions

Often the condition given in an IF statement may be a complex one. For example, in the TV game Family Fortunes, you only win the star prize if you get 200 points and guess the most popular answers to a series of questions. This can be described in our more formal style as:

```
IF at least 200 points gained AND all most popular answers have been guessed
THEN
    winning team get the star prize
ENDIF
```

The AND Operator

Note the use of the word AND in the above example. AND (called a **Boolean operator**) is one of the terms used to link simple conditions in order to produce a more complex one (known as a **complex condition**).

The conditions on either side of the AND are called the **operands**. Both operands must be true for the overall result to be true. We can generalise this to describe the AND operator as being used in the form:

condition 1 AND condition 2

The result of the AND operator is determined using the following rules:

1. Determine the truth of condition 1
2. Determine the truth of condition 2
3. IF both conditions are true THEN
 the overall result is true
 ELSE
 the overall result is false
 ENDIF

For example, if a proximity light comes on when it's dark and it detects motion then we can describe the logic of the equipment as:

```
IF it's dark AND motion has been detected THEN
    Switch on light
ENDIF
```

Now, if we assume that at a particular moment in time it's dark but no motion has been detected then the above statement would be dealt with in the manner shown in FIG-1.4.

FIG-1.4

The AND Operator

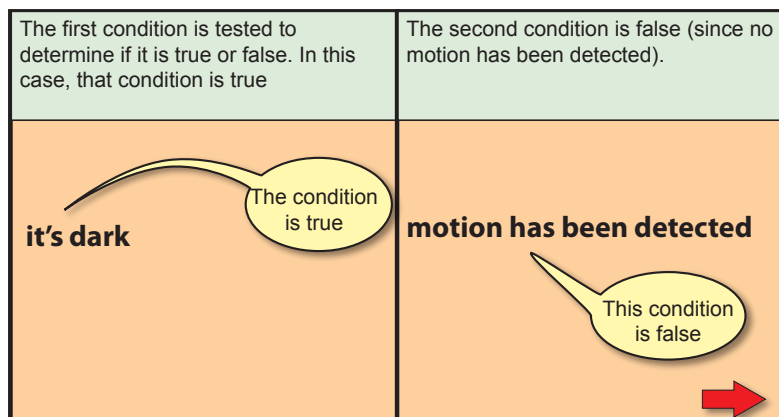
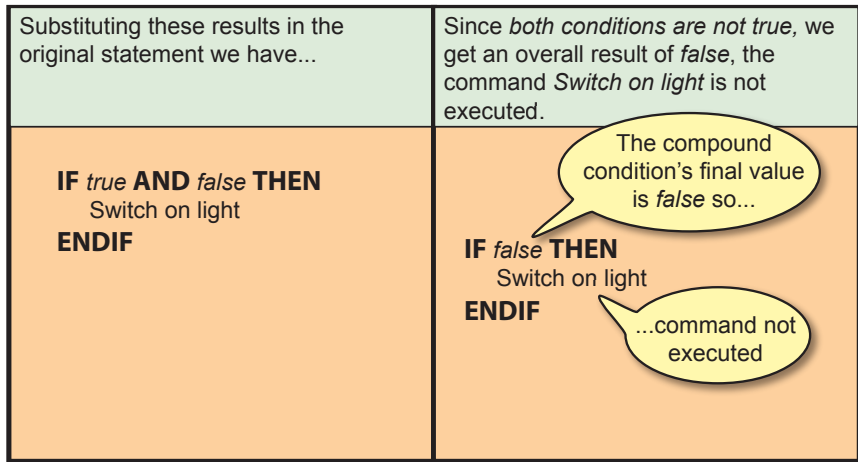


FIG-1.4
(continued)

The AND Operator



With two conditions there are four possible combinations. The first possibility is that both conditions are *false*; another possibility is that condition 1 is *false* but condition 2 is *true*, etc.

Activity 1.9

What are the other possible combinations for the two conditions?

All possibilities of the AND operator are summarised in FIG-1.5.

FIG-1.5

The AND Truthtable

Note that the result is *true* only when both conditions are *true*.

condition 1	condition 2	condition 1 AND condition 2
false	false	false
false	true	false
true	false	false
true	true	true

Activity 1.10

In Microsoft Windows applications, the program will request the name of the file to be opened if the **Ctrl** and **O** keys are pressed together.

Write an IF statement, which includes the term AND, summarising this situation.

The OR Operator

Simple conditions may also be linked by the Boolean OR operator. Using OR, only one of the two conditions specified needs to be true in order to carry out the action that follows. For example, in the game of *Monopoly* you go to jail if you land on the *Go To Jail* square or if you throw three doubles in a row. This can be written as:

```

IF player landed on Go To Jail OR player has thrown 3 pairs in a row THEN
    Move player to jail
ENDIF
    
```

Like AND, the OR operator works on two operands:

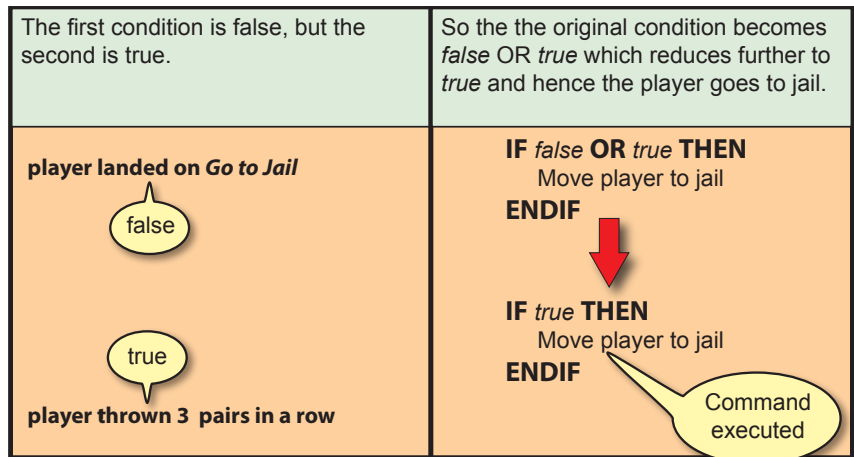
When OR is used, only one of the conditions involved needs to be true for the overall result to be true. Hence the results are determined by the following rules:

1. Determine the truth of condition 1
2. Determine the truth of condition 2
3. IF any of the conditions are true THEN
 the overall result is true
 ELSE
 the overall result is false
 ENDIF

For example, if a player in the game of *Monopoly* has not landed on the *Go To Jail* square, but has thrown three consecutive pairs, then the result of the IF statement given above would be determined as shown in FIG-1.6.

FIG-1.6

The **OR** Operator



The results of the OR operator are summarised in FIG-1.7.

FIG-1.7

The **OR** Truthtable

condition 1	condition 2	condition 1 OR condition 2
false	false	false
false	true	true
true	false	true
true	true	true

Activity 1.11

In *Monopoly*, a player can get out of jail if he throws a double or pays a £50 fine.

Express this information in an IF statement which makes use of the OR operator.

The NOT Operator

The final Boolean operator which can be used as part of a condition is NOT. This operator is used to reverse the meaning of a condition. Hence, if *it's dark* is true, then *NOT it's dark* is false. In fact, you can usually get away with just testing for the opposite condition rather than using NOT. For example, rather than write *NOT it's dark* (which isn't exactly regular English), you can write *it's light* - assuming light and dark are the only two options. Where there are many options to choose from, then

using NOT can make things a lot easier. It's a whole lot simpler to write something like

NOT day is Monday

than have to write

day is Tuesday OR day is Wednesday OR day is Thursday, etc.

Notice that the word NOT is always placed at the start of the condition and not where it would appear in everyday English (*day is NOT Monday*).

The NOT operator works on a single operand:

NOT condition

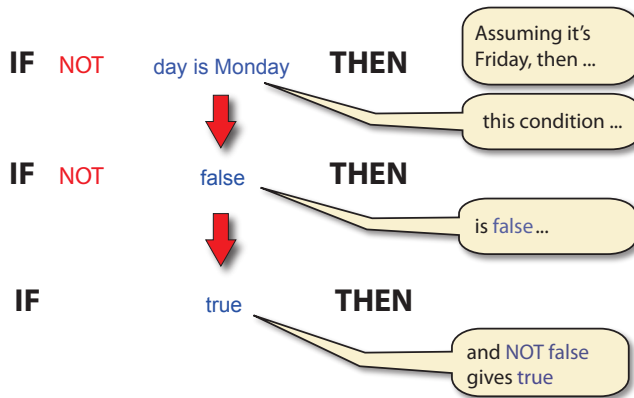
When NOT is used, the result given by the original condition (the bit without the NOT) is reversed. Hence the results are determined by the following rules:

1. Determine the truth of the original condition
2. Complement the result obtained in step 1

For example, if a player lands on a property that is not mortgaged, then the result of the IF statement given above would be determined as shown in FIG-1.8.

FIG-1.8

The NOT Operator



The results of the NOT operator are summarised in FIG-1.9.

FIG-1.9

The NOT Truthtable

condition	NOT condition
false	true
true	false

Activity 1.12

- a) Name the three types of control structures.
- b) Another term for *condition* is what?
- c) Name the two types of selection.
- d) What does the term *mutually exclusive conditions* mean?
- e) Give an example of a Boolean operator.
- f) What is a *conditional statement*?
- g) If two conditions are linked using the term AND, how many of the conditions must be true before the conditional statement is executed?

Iteration

There are certain circumstances in which it is necessary to perform the same sequence of instructions several times. For example, let's assume that a game involves throwing a dice three times and adding up the total of the values thrown. We could write instructions for such a game as follows:

```
Set the total to zero
Throw dice
Add dice value to total
Throw dice
Add dice value to total
Throw dice
Add dice value to total
Call out the value of total
```

You can see from the above that two instructions,

```
Throw dice
Add dice value to total
```

are carried out three times, once for each turn taken by the player. Not only does it seem rather time-consuming to have to write the same pair of instructions three times, but it would be even worse if the player had to throw the dice 10 times!

What is required is a way of showing that a section of the instructions is to be repeated a fixed number of times. Carrying out one or more statements over and over again is known as **looping** or **iteration**. The statement or statements we want to perform over and over again are known as the **loop body**.

Activity 1.13

What statements make up the loop body in our dice problem given above?

FOR..ENDFOR

When writing a formal algorithm in which we wish to repeat a set of statements a specific number of times, we use a FOR..ENDFOR structure. There are three separate parts to this structure. The first of these is placed just before the loop body and in it we state how often we want the statements in the loop body to be carried out. For the dice problem our statement would be:

```
FOR 3 times DO
```

Generalising, we can say this statement takes the form

```
FOR value times DO
```

where *value* would be some positive number.

Next come the statements that make up the loop body. These are indented:

```
FOR 3 times DO
  Throw dice
  Add dice value to total
```

Finally, to mark the fact that we have reached the end of the loop body statements we

add the word ENDFOR:

```
FOR 3 times DO
  Throw dice
  Add dice value to total
ENDFOR
```

Now we can rewrite our original algorithm as:

Note that ENDFOR is left-aligned with the opening FOR statement.

```
Set the total to zero
FOR 3 times DO
  Throw dice
  Add dice value to total
ENDFOR
Call out the value of total
```

The instructions between the terms FOR and ENDFOR are now carried out three times.

Activity 1.14

If the player was required to throw the dice 10 times rather than 3, what changes would we need to make to the algorithm?

If the player was required to call out the average of these 10 numbers, rather than the total, show what other changes are required to the set of instructions.

You can find the average of the 10 numbers by dividing the final total by 10.

We are free to place any statements we wish within the loop body. For example, the last version of our number guessing game produced the following algorithm:

```
Player 1 thinks of a number between 1 and 100
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
  Player 1 says "Correct"
ELSE
  IF guess is less than number THEN
    Player 1 says "Too low"
  ELSE
    Player 1 says "Too high"
  ENDIF
ENDIF
ENDIF
```

player 2 would have more chance of winning if he were allowed several chances at guessing *player 1*'s number. To allow several attempts at guessing the number, some of the statements given above would have to be repeated.

Activity 1.15

What statements in the algorithm above need to be repeated?

To allow for 7 attempts our new algorithm becomes:

```
Player 1 thinks of a number between 1 and 100
FOR 7 times DO
  Player 2 makes an attempt at guessing the number
  IF guess matches number THEN
    Player 1 says "Correct"
  ELSE
    IF guess is less than number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
ENDFOR
```

```
ENDIF
ENDIF
ENDFOR
```

Activity 1.16

Can you see a flaw in the algorithm?

If not, try playing the game a few times, playing exactly according to the instructions in the algorithm.

Activity 1.17

During a lottery draw, two actions are performed exactly 6 times. These are:

```
Pick out ball
Call out number on the ball
```

Add a FOR loop to the above statements to create an algorithm for the lottery draw process.

Occasionally, we may have to use a slightly different version of the FOR loop. Imagine we are trying to write an algorithm explaining how to decide who goes first in a game. In this game every player throws a dice and the player who throws the highest value goes first. To describe this activity we know that each player does the following task:

```
Player throws dice
```

But since we can't know in advance how many players there will be, we write the algorithm using the statement

```
FOR every player DO
```

to give the following algorithm

```
FOR every player DO
  Throw dice
ENDFOR
Player with highest throw goes first
```

If we had to save the details of a game of chess with the intention of going back to the game later, we might write:

```
FOR each piece on the board DO
  Write down the name and position of the piece
ENDFOR
```

Activity 1.18

A game uses cards with images of warriors. At one point in the game the player has to remove from his hand every card with an image of a knight. To do this the player must look through every card and, if it is a knight, remove the card.

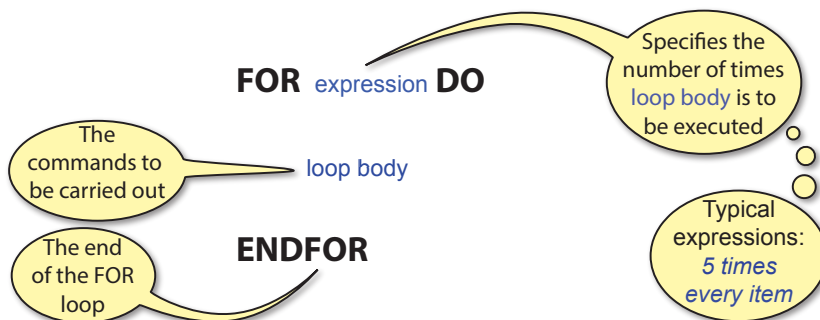
Write down a set of instructions which performs the task described above. Your solution should include the statements

```
FOR every card in player's hand DO and IF card is a knight THEN
```

The general form of the FOR statement is shown in FIG-1.10.

FIG-1.10

The FOR..ENDFOR Loop



Although the FOR loop allows us to perform a set of statements a specific number of times, this statement is not always suitable for the problem we are trying to solve.

For example, in the guessing game of Activity 1.15 we stated that the loop body was to be performed 7 times, but what if player 2 guesses the number after only three attempts? If we were to follow the algorithm exactly (as a computer would), then we must make four more guesses at the number even after we know the correct answer!

To solve this problem, we need another way of expressing looping which does not commit us to a specific number of iterations.

REPEAT.. UNTIL

The REPEAT .. UNTIL statement allows us to specify that a set of statements should be repeated until some condition becomes true, at which point iteration should cease.

The word REPEAT is placed at the start of the loop body and, at its end, we add the UNTIL statement. The UNTIL statement also contains a condition, which, when true, causes iteration to stop. This is known as the **terminating** (or exit) **condition**. For example, we could use the REPEAT.. UNTIL structure rather than the FOR loop in our guessing game algorithm. The new version would then be:

```
Player 1 thinks of a number between 1 and 100
REPEAT
  Player 2 makes an attempt at guessing the number
  IF guess matches number THEN
    Player 1 says "Correct"
  ELSE
    IF guess is less than number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
UNTIL player 2 guesses correctly
```

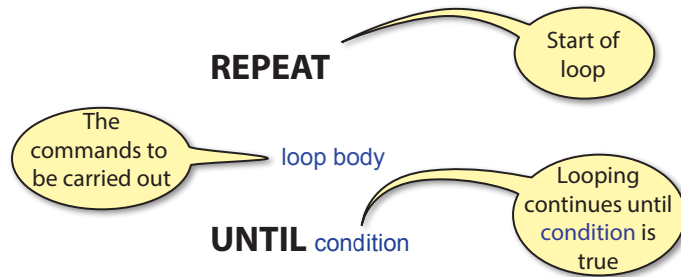
We could also use the REPEAT..UNTIL loop to describe how a slot machine (one-armed bandit) is played:

```
REPEAT
  Put coin in machine
  Pull handle
  IF you win THEN
    Collect winnings
  ENDIF
UNTIL you want to stop
```

The general form of this structure is shown in FIG-1.11.

FIG-1.11

The REPEAT..UNTIL Loop



The terminating condition may use the Boolean operators AND, OR and NOT as well as parentheses, where necessary.

Activity 1.19

Confronted with a pile of unordered books when looking for a specific publication, the only way to find the desired title is to examine each book in turn until the required one is found. Of course, there's a possibility that the book is not in the pile.

Using REPEAT..UNTIL, write the logic required to search for the book.

Returning to the number guessing game on the previous page, there is still a problem. By using a REPEAT .. UNTIL loop we are allowing *player 2* to have as many guesses as needed to determine the correct number. That doesn't lead to a very interesting game. Later we'll discover how we might solve this problem.

WHILE.. ENDWHILE

A final method of iteration, differing only subtly from the REPEAT.. UNTIL loop, is the WHILE .. ENDWHILE structure which has an **entry condition** at the start of the loop. The following example illustrates the usefulness of this new structure.

The aim of the card game of Pontoon is to attempt to make the value of your cards add up to 21 without going over that value. Each player is dealt two cards initially but can repeatedly ask for more cards by saying "twist". One player is designated the dealer. The dealer must twist while his cards have a total value of less than 16. So we might write the rules for the dealer as:

```
Calculate the sum of the initial two cards
REPEAT
    Take another card
    Add new card's value to sum
UNTIL sum is greater than or equal to 16
```

But there's a problem with the solution: if the sum of the first two cards is already 16 or above, we still need to take a third card (just work through the logic, if you can't see why). By using the WHILE..ENDWHILE structure we could describe the logic as

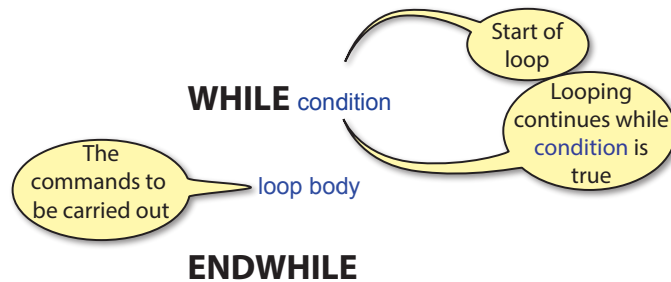
```
Calculate sum of the initial two cards
WHILE sum is less than 16 DO
    Take another card
    Add new card's value to sum
ENDWHILE
```

Now determining if the sum is less than 16 is performed before the *Take another card* instruction. If the dealer's two cards already add up to 16 or more, then the *Take another card* instruction will be ignored.

The general form of the WHILE.. ENDWHILE statement is shown in FIG-1.12.

FIG-1.12

The WHILE..
ENDWHILE Loop



In what way does this differ from the REPEAT statement? There are two differences:

- The condition is given at the beginning of the loop.
- Looping stops when the condition is false.

The main consequence of this is that it is possible to bypass the loop body of a WHILE structure entirely without ever carrying out any of the instructions it contains.

On the other hand, the loop body of a REPEAT structure will always be executed at least once.

Activity 1.20

A game involves throwing two dice. If the two values thrown are not the same, then the dice showing the lower value must be rolled again. This process is continued until both dice show the same value.

Write a set of instructions to perform this game.

Your solution should contain the statements

and Roll both dice
 Choose dice with lower value

Activity 1.21

- a) What is the meaning of the term **iteration**?
- b) Name the three types of looping structures.
- c) What type of loop structure should be used when looping needs to occur an exact number of times?
- d) What type of loop structure can bypass its loop body without ever executing it?
- e) What type of loop contains an exit condition?

Infinite Loops

If a loop can never exit, it is known as an **infinite loop**. As a general rule, infinite loops are caused by some error in the logic. For example, the algorithm

Think of a number
REPEAT
 Subtract 1 from the number
UNTIL the number is zero

will never be completed if the number you start with is already zero or less.

Data

We know we need to retain information. Look at your phone; packed with names, email addresses, phone numbers, and much more. Even when playing an old-fashioned board game we need to remember things such as the number you threw on the dice, where your piece is on the board and so on. These examples introduce the need to process facts and figures (known as **data**).

Every item of data has two basic characteristics :

 a name
and a value

The name of a data item is a description of the type of information it represents. Hence on a form we might see boxes labelled as *Forename*, *Surname*, *Address*, *Phone No*, etc. These are the data names. And, when we've completed the form, the boxes will contain the values we have entered. These entries are the data values. In programming, a data item is often referred to as a **variable**. This term arises from the fact that, although the name assigned to a data item cannot change, its value may vary. For example, the value assigned to a variable called *salary* may rise (or fall) over weeks, months or years.

Types of Data

Most computer programming languages need to be told what type of value is to be held in a variable - for example, it needs to know if a variable will hold a number or a message. Once the variable is set up for one type of value, it can't be used to hold any other type. Three of the basic data types recognised by a language such as AGK BASIC are:

integer	holds whole numbers only (eg -12, 0, 92).
real	(also known as floating point numbers) holds numbers containing fractions (-14.6, 0.005, 176.0) - notice that the fraction part may be .0.
string	holds zero or more characters. A character may be alphabetic, numeric, or punctuation marks (A, 7, *).

Other data types are possible, but we'll look at these in a later chapter.

Operations on Data

There are four basic operations that a computer can do with data. These are:

Input

This involves being given a value for a data item. For example, in our number-guessing game, the player who has thought of the original number is given the value

of the guess from the second player. When playing Noughts and Crosses adding an X (or O) changes the set up on the board. When using a computer, any value entered at the keyboard, or any movement or action dictated by a mouse or joystick would be considered as data entry. This type of action is known as an **input operation**.

Calculation

Most games involve some basic arithmetic. In Monopoly, the banker has to work out how much change to give a player buying a property. If a character in an adventure game is hit, points must be deducted from his strength value. This type of instruction is referred to as a **calculation operation**.

Comparison

Often values have to be compared. For example, we need to compare the two numbers in our guessing game to find out if they are the same. This is known as a **comparison operation**.

Output

The final requirement is to communicate with others to give the result of some calculation or comparison. For example, in the guessing game, player 1 communicates with player 2 by saying either that the guess is *Correct*, *Too high* or *Too low*.

In a computer environment, the equivalent operation would normally involve displaying information on a screen or printing it on paper. For instance, in a racing game your speed and time will be displayed on the screen. This is called an **output operation**.

When describing a calculation, it is common to use arithmetic operator symbols rather than English. Hence, instead of writing the word subtract we use the minus sign (-). However, programming languages use a slightly different set of symbols than standard mathematics (see FIG-1.13).

FIG-1.13

The Arithmetic Operators

English	Symbol
Multiply	*
Divide	/
Add	+
Subtract	-

Similarly, when we need to compare values, rather than use terms such as *is less than*, we use the *less than* symbol (<). A summary of these relational operators is given in FIG-1.14.

FIG-1.14

The Relational Operators

English	Symbol
is less than	<
is less than or equal to	<=
is greater than	>
is greater than or equal to	>=
is equal to	=
is not equal to	<>

As well as replacing the words used for arithmetic calculations and comparisons with

symbols, the term *calculate* or *set* is often replaced by the shorter but more cryptic symbol \rightarrow between the variable being assigned a value and the value itself. Using this abbreviated form, the instruction:

Calculate time to complete course as distance divided by speed

becomes

`time \rightarrow distance / speed`

Although the long-winded English form is more readable, this more cryptic style is briefer and is much closer to the code used when programming a computer.

Below we compare the two methods of describing our guessing game; first in English:

```
Player 1 thinks of a number between 1 and 100
REPEAT
  Player 2 makes an attempt at guessing the number
  IF guess matches number THEN
    Player 1 says "Correct"
  ELSE
    IF guess is less than number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
UNTIL player 2 guesses correctly
```

Using some of the symbols described earlier, we can rewrite this as:

```
Player 1 thinks of a number between 1 and 100
REPEAT
  Player 2 makes an attempt at guessing the number
  IF guess = number THEN
    Player 1 says "Correct"
  ELSE
    IF guess < number THEN
      Player 1 says "Too low"
    ELSE
      Player 1 says "Too high"
    ENDIF
  ENDIF
UNTIL guess = number
```

Activity 1.22

- a) What are the two main characteristics of any data item?
- b) When data is input, from where is its value obtained?
- c) Give an example of a relational operator.

Levels of Detail

When we start to write an algorithm in English, one of the things we need to consider is exactly how much detail should be included. For example, we might describe how to record a video on a digital camcorder as:

```
Insert memory stick
Choose appropriate recording settings
```


However, this lacks enough detail for anyone unfamiliar with the operation of the machine. Therefore, we could replace the first statement with:

```
Open the flap covering the memory chip slot
IF there is a chip already in the slot THEN
    Remove it
ENDIF
Place the new memory stick in slot
Close flap
```

and the second statement could be substituted by:

```
Set recording quality
Set exposure to automatic
Set focus to automatic
```

This approach of starting with a less detailed sequence of instructions and then, where necessary, replacing each of these with more detailed instructions can be used to good effect when tackling long and complex problems. By using this technique, we are defining the original problem as an equivalent sequence of simpler problems before going on to create a set of instructions to handle each of these simpler problems. This divide-and-conquer strategy is known as **stepwise refinement**. The following is a fully worked example of this technique:

Problem:

Describe how to make a cup of tea.

Outline Solution:

1. Fill kettle
2. Boil water
3. Put tea bag in teapot
4. Add boiling water to teapot
5. Wait 1 minute
6. Pour tea into cup
7. Add milk and sugar to taste

This is termed a **LEVEL 1 solution**.

As a guideline we should aim for a LEVEL 1 solution with between 5 and 12 instructions. Notice that each instruction has been numbered. This is merely to help with identification during the stepwise refinement process.

Before going any further, we must assure ourselves that this is a correct and full (though not detailed) description of all the steps required to tackle the original problem. If we are not happy with the solution, then changes must be made before going any further.

Next, we examine each statement in turn and determine if it should be described in more detail. Where this is necessary, rewrite the statement to be dealt with, and below it, give the more detailed version. For example. Fill kettle would be expanded thus:

1. Fill kettle
 - 1.1 Remove kettle lid
 - 1.2 Put kettle under tap
 - 1.3 Turn on tap
 - 1.4 When kettle is full, turn off tap
 - 1.5 Replace lid on kettle

The numbering of the new statement reflects that they are the detailed instructions

pertaining to statement 1. Also note that the number system is not a decimal fraction, so if there were to be many more statements they would be numbered 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, etc.

It is important that these sets of more detailed instructions describe how to perform only the original task being examined - they must achieve no more and no less. Sometimes the detailed instructions will contain control structures such as IFs, WHILEs or FORs. Where this is the case, the whole structure must be included in the detailed instructions for that task. Having satisfied ourselves that the breakdown is correct, we proceed to the next statement from the original solution.

2. Boil water
 - 2.1 Plug in kettle
 - 2.2 Switch on power at socket
 - 2.3 Switch on power at kettle
 - 2.4 When water boils switch off kettle

The next two statements expand as follows:

3. Put tea bag in teapot
 - 3.1 Remove lid from teapot
 - 3.2 Add tea bag to teapot
4. Add boiling water to teapot
 - 4.1 Take kettle over to teapot
 - 4.2 Add required quantity of water from kettle to teapot

But not every statement from a level 1 solution needs to be expanded. In our case there is no more detail to add to the statement

5. Wait 1 minute

and therefore, we leave it unchanged.

The last two statements expand as follows:

6. Pour tea into cup
 - 6.1 Take teapot over to cup
 - 6.2 Pour required quantity of tea from teapot into cup
7. Add milk and sugar as required
 - 7.1 IF milk is required THEN
 - 7.2 Add milk
 - 7.3 ENDIF
 - 7.4 IF sugar is required THEN
 - 7.5 Add sugar
 - 7.6 Stir tea
 - 7.7 ENDIF

Notice that this last expansion (step 7) has introduced IF statements. Control structures (i.e. IF, WHILE, FOR, etc.) can be introduced at any point in an algorithm.

Finally, we can describe the solution to the original problem in more detail by substituting the statements in our LEVEL 1 solution by their more detailed equivalent:

- 1.1 Remove kettle lid
- 1.2 Put kettle under tap
- 1.3 Turn on tap
- 1.4 When kettle is full, turn off tap
- 1.5 Place lid back on kettle
- 2.1 Plug in kettle
- 2.2 Switch on power at socket
- 2.3 Switch on power at kettle

- 2.4 When water boils switch off kettle
- 3.1 Remove lid from teapot
- 3.2 Add tea bag to teapot
- 4.1 Take kettle over to teapot
- 4.2 Add required quantity of water from kettle to teapot
- 5. Wait 1 minute
- 6.1 Take teapot over to cup
- 6.2 Pour required quantity of tea from teapot into cup
- 7.1 IF milk is required THEN
- 7.2 Add milk
- 7.3 ENDIF
- 7.4 IF sugar is required THEN
- 7.5 Add sugar
- 7.6 Stir tea
- 7.7 ENDIF

This is a LEVEL 2 solution. Note that a level 2 solution includes any LEVEL 1 statements which were not given more detail (in this case, *Wait 1 minute*).

For some more complex problems it may be necessary to repeat this process to more levels before sufficient detail is achieved. That is, statements in LEVEL 2 may be given more detail in a LEVEL 3 breakdown.

Activity 1.23

The game of battleships involves two players. Each player draws two 10 by 10 grids. Each of these have columns lettered A to J and rows numbered 1 to 10. In the first grid each player marks the position of warships. Ships are added as follows:

- 1 aircraft carrier 4 squares
- 2 destroyers 3 squares each
- 3 cruisers 2 squares each
- 4 submarines 1 square each

The squares of each ship must be adjacent and must be vertical or horizontal. The first player now calls out a grid reference.

The second player responds to the call by saying HIT or MISS. HIT is called if the grid reference corresponds to a position of a ship. The first player then marks this result on his second grid using an o to signify a miss and x for a hit (see diagram below).

	A	B	C	D	E	F	G	H	I	J
1										
2										
3			A	A	A	A				
4									S	
5	C	C						D		
6				S				D		
7		D	D	D				D		
8						C			S	
9		S				C				
10				C	C					

	A	B	C	D	E	F	G	H	I	J
1										O
2										
3							O			
4										
5										
6			X	X	X					
7								O		
8										
9										
10										

Vessels are positioned in the left-hand grid

Results of guesses are placed in the right-hand grid

continued on next page

Activity 1.23 (continued)

If the first player achieves a HIT then he continues to call grid references until MISS is called. In response to a HIT or MISS call the first player marks the second grid at the reference called: 0 for a MISS, X for a HIT.

When the second player responds with MISS the first player's turn is over, and the second player has his turn.

The first player to eliminate all segments of the opponent's ships is the winner. However, each player must have an equal number of turns, and if both sets of ships are eliminated in the same round the game is a draw.

The algorithm describing the task of one player is given in the instructions below. Create a LEVEL 1 algorithm by assembling the lines in the correct order, adding line numbers to the finished description.

```
Add ships to left grid
UNTIL there is a winner
Call grid position(s)
REPEAT
Respond to other player's call(s)
Draw grids
```

To create a LEVEL 2 algorithm, some of the above lines will have to be expanded to give more detail. More detailed instructions are given below for the statements Call grid position(s) and Respond to other player's call(s).

By reordering and numbering the lines below create LEVEL 2 details for these two statements.

```
UNTIL other player misses
Mark position in second grid with X
Get other player's call
Get reply
Get reply
ENDIF
Call HIT
Call MISS
Mark position in second grid with 0
WHILE reply is HIT DO
Call grid reference
Call grid reference
IF other player's call matches position of ship THEN
ENDWHILE
REPEAT
ELSE
```

Checking for Errors

Once we've created our algorithm we would like to make sure it is correct. Unfortunately, there is no foolproof way to do this! But we can at least try to find any errors or omissions in the set of instructions we have created.

We do this by going back to the original description of the task our algorithm is attempting to solve. For example, let's assume we want to check our number guessing

game algorithm. In the last version of the game we allowed the second player to make as many guesses as required until he came up with the correct answer. The first player responded to each guess by saying either “Too low”, “Too high” or “Correct”.

To check our algorithm for errors we must come up with typical values that might be used when carrying out the set of instructions and those values should be chosen so that each possible result is achieved at least once.

So, as well as making up values, we need to predict what response our algorithm should give to each value used. Hence, if the first player thinks of the value 42 and the second player guesses 75, then the first player will respond to the guess by saying “Too high”.

Our set of test values must evoke each of the possible results from our algorithm. One possible set of values and the responses are shown in FIG-1.15.

FIG-1.15

Test Data for the
Number Guessing Game
Algorithm

Test Data	Expected Results
number = 42 guess = 75	Says “Too high”
guess = 15	Says “Too low”
guess = 42	Says “Correct”

Once we’ve created test data, we need to work our way through the algorithm using that test data and checking that we get the expected results. The algorithm for the number game is shown below, this time with instruction numbers added.

1. Player 1 thinks of a number between 1 and 100
2. REPEAT
3. Player 2 makes an attempt at guessing the number
4. IF guess = number THEN
5. Player 1 says “Correct”
6. ELSE
7. IF guess < number THEN
8. Player 1 says “Too low”
9. ELSE
10. Player 1 says “Too high”
11. ENDIF
12. ENDIF
13. UNTIL guess = number

Next we create a new table (called a **trace table**) with the headings as shown in FIG-1.16.

FIG-1.16

A Trace Table

Instruction	Condition	T/F	Variables <i>number guess</i>	Output

Now we work our way through the statements in the algorithm filling in a line of the trace table for each instruction.

Instruction 1 is for player 1 to think of a number. Using our test data, that number will be 42, so our trace table starts with the line shown in FIG-1.17.

FIG-1.17

Working through a Trace 1

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	

The REPEAT word comes next. Although this does not cause any changes, nevertheless a 2 should be entered in the next line of our trace table. Instruction 3 involves player 2 making a guess at the number (this guess will be 75 according to our test data). After 3 instructions our trace table is as shown in FIG-1.18.

FIG-1.18

Working through a Trace 2

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	
2				
3			75	

Instruction 4 is an IF statement containing a condition. This condition and its result are written into columns 2 and 3 as shown in FIG-1.19.

FIG-1.19

Working through a Trace 3

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	
2				
3			75	
4	guess = number	F		

Because the condition is false, we now jump to instruction 6 (the ELSE line) and on to 7. This is another IF statement and our table now becomes that shown in FIG-1.20.

FIG-1.20

Working through a Trace 4

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		

Since this second IF statement is also false, we move on to statements 9 and 10. Instruction 10 causes output (speech) and hence we enter this in the final column as shown in FIG-1.21.

FIG-1.21

Working through a Trace 5

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high

Now we move on to statements 11,12 and 13 as shown in FIG-1.22.

FIG-1.22

Working through a Trace 6

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high
11				
12				
13	guess = number	F		

Since statement 13 contains a condition which is false, we return to statement 2, executing it and then moving on to 3 where we enter 15 as our second guess (see FIG-1.23).

FIG-1.23

Working through a Trace 7

Instruction	Condition	T/F	Variables <i>number guess</i>	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high
11				
12				
13	guess = number	F		
2				
3			15	

This method of checking is known as **desk checking** or **dry running**.

Activity 1.24

Create your own trace table for the number-guessing game and, using the same test data as given in FIG-1.15 complete the testing of the algorithm.

Were the expected results obtained?

Summary

- Computers can perform many tasks by executing different programs.
- An algorithm is a sequence of instructions which solves a specific problem.
- A program is a sequence of computer instructions which usually manipulates data and produces results.
- Three control structures are used in programs :
 - Sequence
 - Selection
 - Iteration

- A sequence is a list of instructions which are performed one after the other.
- Selection involves choosing between two or more alternative actions.
- Selection is performed using the IF statement.
- There are three forms of IF statement:

```
IF condition THEN
    instructions
ENDIF
```

```
IF condition THEN
    instructions
ELSE
    instructions
ENDIF
```

```
IF
    condition 1:
        instructions
    condition 2:
        instructions
    condition x :
        instructions
ELSE
    instructions
ENDIF
```

- Iteration is the repeated execution of one or more statements.
- Iteration is performed using one of three instructions:

```
FOR number of iterations required DO
    instructions
ENDFOR
REPEAT
    instructions
UNTIL condition
```

```
WHILE condition DO
    instructions
ENDWHILE
```

- A condition is an expression which is either *true* or *false*.
- Simple conditions can be linked using AND or OR to produce a complex condition.
- The meaning of a condition can be reversed by adding the word NOT.
- Data items (or variables) hold the information used by the algorithm.
Data item values may be:

```
Input
Calculated
Compared
or Output
```

- Calculations can be performed using the following arithmetic operators:

Multiplication	*
Addition	+
Division	/
Subtraction	-

- The order of priority of an operator may be overridden using parentheses.
- Comparisons can be performed using the relational operators:

Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	=
Not equal to	<>

- The symbol -> is used to assign a value to a data item. Read this symbol as is assigned the value.
- In programming, a data item is referred to as a variable.
- The divide-and-conquer strategy of stepwise refinement can be used when creating an algorithm.
- LEVEL 1 solution gives an overview of the sub-tasks involved in carrying out the required operation.
- LEVEL 2 gives a more detailed solution by taking each sub-task from LEVEL 1 and, where necessary, giving a more detailed list of instructions required to perform that sub-task.
- Not every statement needs to be broken down into more detail.
- Further levels of detail may be necessary when using stepwise refinement for complex problems.
- Further refinement may not be required for every statement.
- An algorithm can be checked for errors or omissions using a trace table.

Solutions

Activity 1.1

No solution required.

Activity 1.2

One possible solution is:

```
Fill A
Fill B from A
Empty B
Empty A into B
Fill A
Fill B from A
```

Activity 1.3

- An algorithm
- A computer program
- mips (millions of instructions per second)

Activity 1.4

```
Choose club
Take up correct stance beside ball
Grip club correctly
Swing club backwards
Swing club forwards, attempting to hit ball
```

The second and third statements could be interchanged.

Activity 1.5

```
Player 1 thinks of a number
Player 2 makes a guess at the number
IF guess matches number THEN
    Player 1 says "Correct"
ENDIF
Player 1 states the value of the number
```

Activity 1.6

```
IF letter appears in word THEN
    Add letter at appropriate position(s)
ELSE
    Add part to hanged man
ENDIF
```

Activity 1.7

```
IF the crossbow is on target THEN
    Say "Fire"
ELSE
    IF the crossbow is pointing too high THEN
        Say "Down a bit"
    ELSE
        IF the crossbow is pointing too low THEN
            Say "Up a bit"
        ELSE
            IF the crossbow is too far left THEN
                Say "Right a bit"
            ELSE
                Say "Left a bit"
            ENDIF
        ENDIF
    ENDIF
ENDIF
```

Activity 1.8

```
IF
    you know the phrase:
        Make guess at phrase
        there are many unseen letters:
            Guess a consonant
ELSE
    Buy a vowel
ENDIF
```

Activity 1.9

Other possibilities are:

Both conditions are true
condition 1 is true and condition 2 is false

Activity 1.10

```
IF Ctrl key pressed AND O key pressed THEN
    Request filename
ENDIF
```

Activity 1.11

```
IF double thrown OR fine paid THEN
    Player gets out of jail
ENDIF
```

Activity 1.12

- Sequence
Selection
Iteration
- Boolean expression
- Binary selection Multi-way selection
- No more than one of the conditions can be true at any given time.
- Boolean operators are: AND, OR, and NOT.
- A conditional statement is a statement which is executed only if a given set of conditions are met.
- Both conditions must be true.

Activity 1.13

```
Throw dice
Add dice value to total
```

Activity 1.14

Only one line, the FOR statement, would need to be changed, the new version being:

```
FOR 10 times DO
```

To call out the average, the algorithm would change to

```
Set the total to zero
FOR 10 times DO
    Throw dice
    Add dice value to total
ENDFOR
Calculate average as total divided by 10
Call out the value of average
```

Activity 1.15

In fact, only the first line of our algorithm is not repeated, so the lines that need to be repeated are:

```
Player 2 makes an attempt at guessing the number
IF guess matches number THEN
    Player 1 says "Correct"
ELSE
    IF guess is less than number THEN
        Player 1 says "Too low"
    ELSE
        Player 1 says "Too high"
    ENDIF
ENDIF
```

Activity 1.16

The FOR loop forces the loop body to be executed exactly 7 times. If the player guesses the number in less attempts, the algorithm will nevertheless continue to ask for the remainder of the 7 guesses.

Later, we'll see how to solve this problem.

4.7 ENDWHILE

4.8 Mark position in second grid with 0

Activity 1.17

```
FOR 6 times DO
  Pick out ball
  Call out number on the ball
ENDFOR
```

Activity 1.18

```
FOR every card in player's hand DO
  IF card is a knight THEN
    Remove card from hand
  ENDFOR
ENDFOR
```

Activity 1.19

```
REPEAT
  Read next book title
UNTIL required title found OR no books remaining
```

Activity 1.20

```
Roll both dice
WHILE dice values don't match DO
  Choose dice with lower value
  Throw chosen dice
ENDWHILE
```

Note that the WHILE line could have been written as

```
WHILE NOT dice values match DO
```

Activity 1.21

- Iteration** means executing a set of statements repeatedly.
- FOR..ENDFOR, REPEAT..UNTIL and WHILE..ENDWHILE
- The FOR..ENDFOR structure.
- The WHILE..ENDWHILE structure.
- The REPEAT..UNTIL structure.

Activity 1.22

- Its name and value.
- From outside the system. In a computerised setup, this is often entered from a keyboard.
- The relational operators are:
 - < (less than)
 - <= (less than or equal to)
 - > (greater than)
 - >= (greater than or equal to)
 - = (equal to)
 - <> (not equal to)

Activity 1.23

The LEVEL 1 is coded as:

- Draw grids
- Add ships to left grid
- REPEAT
 - Call grid position(s)
 - Respond to other player's call(s)
- UNTIL there is a winner

The expansion of statement 4 would become:

- Call grid reference
- Get reply
- WHILE reply is HIT DO
 - Mark position in second grid with X
 - Call grid reference
 - Get reply

The expansion of statement 5 would become:

5.1 REPEAT

5.2 Get other player's call

5.3 IF other player's call matches position of ship THEN

5.4 Call HIT

5.5 ELSE

5.6 Call MISS

5.7 ENDIF

5.8 UNTIL other player misses

Activity 1.24

Instruction	Condition	T/F	Variables number guess	Output
1			42	
2				
3			75	
4	guess = number	F		
6				
7	guess < number	F		
9				
10				Too high
11				
12				
13	guess = number	F		
2				
3			15	
4	guess = number	F		
6				
7	guess < number	T		
8				Too low
11				
12				
13	guess = number	F		
2				
3			42	
4	guess = number	T		
5				Correct
12				
13	guess = number	T		

2

Starting AGK

In this Chapter:

- Understanding Compilation
- Getting Started with AGK
- Creating a First Project
- Installing an App on a Device
- Creating Output
- Adding Comments
- Changing Output Colour, Size and Spacing
- Adjust an App Window's Properties
- Adding a Splash Screen

Programming a Computer

Introduction

In the last chapter we created algorithms written in a style known as **structured English**. But if we want to create an algorithm that can be followed by a computer, then we need to convert our structured English instructions into a programming language.

A housekeeping program is one which performs mundane chores such as file copying, data communications, etc. and has little user input.

There are many programming languages; C, C++, Java, C#, and Visual Basic being amongst the most widely used. So how do we choose which programming language to use? Each language has its own strengths. For example, Java allows multi-platform programs to be created easily, while C is ideal for creating housekeeping applications. So, when we choose a programming language, we want one that is best suited to the task we have in mind.

We are going to use a programming language known as AGK BASIC. This language was designed specifically for writing computer games which can then be used on a wide range of devices - anything from your regular computer to a tablet or even a smartphone. Because of this, AGK BASIC has many unique commands for displaying graphics on various screen resolutions and for handling a wide range of input methods - anything from a standard mouse to a touch screen or an accelerometer.

The Compilation Process

When you begin the process of creating a game using AGK, several files are automatically created. One of these files is designed to hold your program code; the others hold additional details required by the project. These extra files have their contents created automatically by AGK so we need not worry about them at this stage.

Because each game that we create consists of several files, we refer to this collection of files as a **project**. One of these files (always named *main.agc* in every project) contains the actual program code.

Each new project is automatically assigned its own folder.

As we will soon see, the programming language AGK BASIC uses statements that retain some English terms and phrases. This means we can look at the set of instructions and make some sense of what is happening after only a relatively small amount of training.

Unfortunately, the processor inside a digital device (computer, tablet, or smartphone) understands only instructions given in the form of a sequence of 1's and 0's in a format known as **machine code**. The device has no capability of directly following a set of instructions written in AGK BASIC. But this need not be a problem; we simply need to translate the AGK BASIC statements into machine code (just as we might have a piece of text translated from Russian to English).

We begin the process of creating a new piece of software by mentally converting our structured English algorithm (which we will have already created) into a sequence of AGK BASIC statements. These statements are entered using a text editor which is nothing more than a simple word-processor-like program allowing such basic operations as inserting and deleting text. Once the complete program has been entered, we get the machine itself to translate those instructions from AGK BASIC

into **machine code**. The original program code is known as the **source code**; the machine code is known as the **object code** and the saved version of this as the **executable file**.

The translator (known as a **compiler**) is simply another program installed in the computer. After typing in our program instructions, we feed these to the compiler which produces the equivalent instructions in machine code. These instructions are then executed by the computer and we should see the results of our calculations appear on the screen (assuming there are output statements in the program).

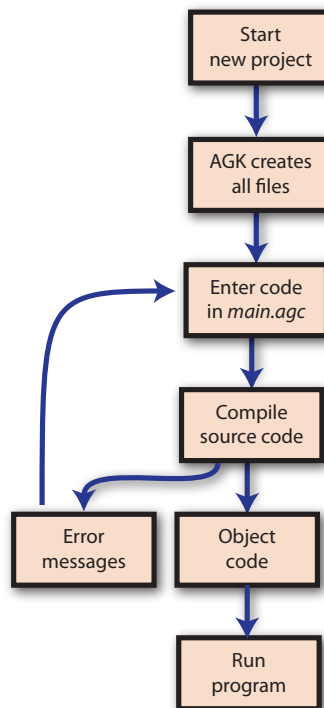
The compiler is a very exacting task master. The structure, or **syntax**, of every statement must be exactly right. If you make the slightest mistake, even something as simple as missing out a comma or misspelling a word, the translation process will fail. When this happens in AGK, a window appears giving details of the error. A failure of this type is known as a **syntax error** - a mistake in the grammar of your commands. Any syntax errors have to be corrected before you can try compiling the program again.

When you are working on a project, it is best to save your work at regular intervals. That way, if there is a power cut, you won't have lost all your code!

When the program code is complete, we compile our program (translating it from source code to object code). When the translation process is finished, yet another file is produced. This new file (which has an *.exe* extension), contains the object code. To run our program, the source code in the executable file is loaded into the computer's memory (RAM) and the instructions it contains are carried out. The whole process is summarised in FIG-2.1.

FIG-2.1

The Compilation Process



If we want to make changes to the program, we load the source code into the editor, make the necessary modifications, then save and recompile our program, thereby replacing the old version of source and executable files.

Activity 2.1

- a) What type of instructions are understood by a computer?
- b) What piece of software is used to translate a program from source code to object code?
- c) Misspelling a word in your program is an example of what type of error?

Starting AGK

Introduction

AGK is an Integrated Development Environment (IDE) software package designed to create 2D games that can then be run on various hardware devices. IDE simply means that the editing, compiling and testing are all achieved while working from within a single package.

AGK allows programs to be written in either BASIC or C++. This book covers only the BASIC language aspect of AGK.

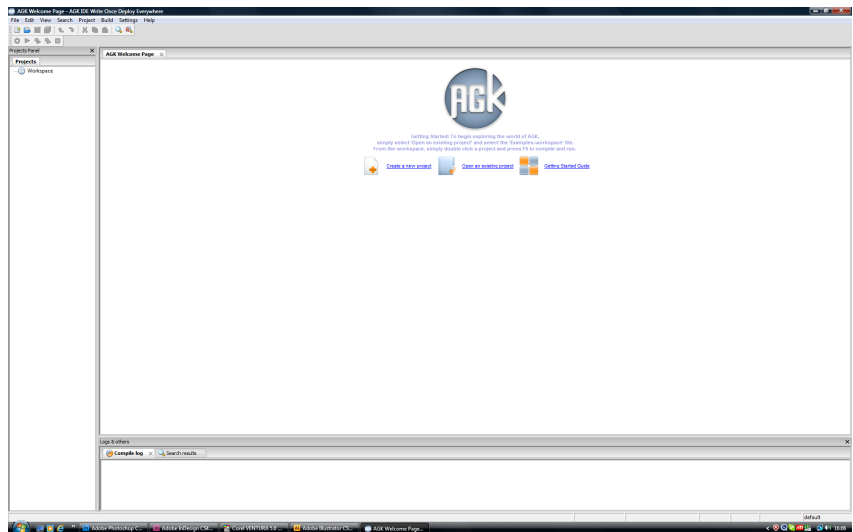
AGK was created by Lee Bamber, CEO of The Game Creators Ltd and was derived from his earlier creation, DarkBASIC which is a programming language designed to develop games for the PC platform only.

Starting Up AGK

Once you've installed AGK, running the package will present you with the screen shown in FIG-2.2.

FIG-2.2

The AGK Startup Screen

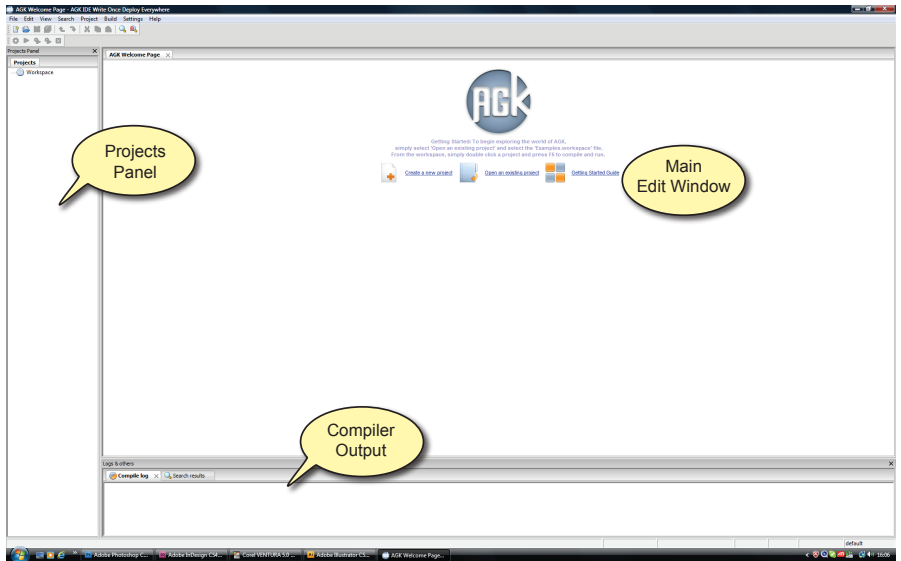


At the centre of the application window is the **Tip of the Day** window. If you don't want this to appear every time you start up AGK, just deselect the *Show tips at startup* check box. Once you close the **Tip of the Day** window, you are left with the three main areas of the AGK IDE (see FIG-2.3):

- The Main Edit Window - This where your program code is displayed once you start working on a project.
- The Project Panel - This displays a tree structure of the files within the project(s) currently open. It only shows the names of those files containing code; the other files created by a project are not listed.
- Compiler Output Panel - This panel (labelled as *Logs and others*) is used primarily to display information output by the compiler.

FIG-2.3

AGK Layout



The steps required to create your first project are shown in FIG-2.4.

FIG-2.4

Creating a New Project

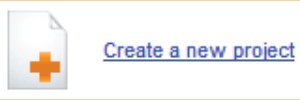
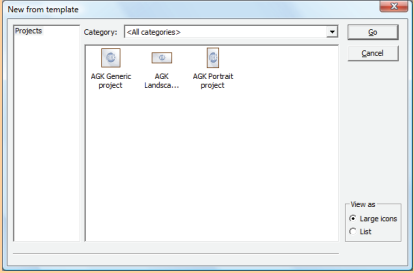
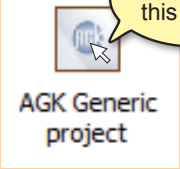
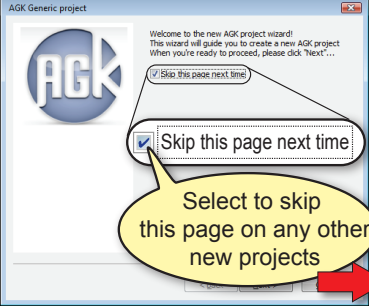
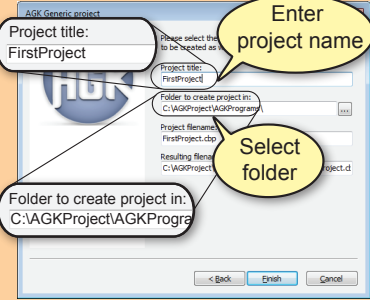
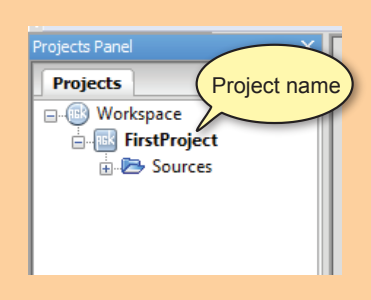
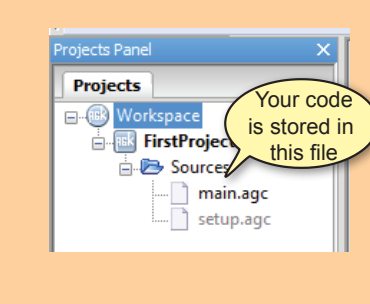
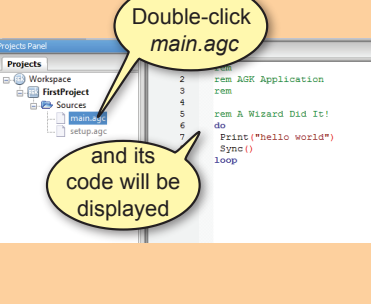

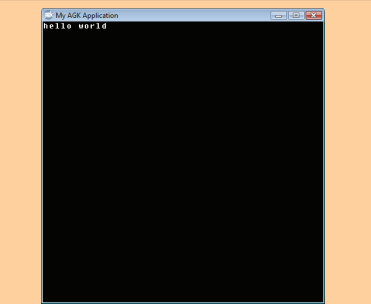
<p>Since this is our first project, we click on the Create a new Project option in the Main Edit Window (File New Project would work too).</p>	<p>This displays the Create from Template window which offers three different layout styles for your new project's display.</p>
	
<p>For this project, AGK Generic project is selected by double clicking that option.</p>	<p>This starts up the AGK project wizard. The first screen simply states that the wizard has started.</p>
 <p>Double-click this option</p>	 <p>Select to skip this page on any other new projects</p>

FIG-2.4
(continued)

Creating a New Project

➡ The project will create a new subfolder off the folder you select here. That subfolder will have the same name as your project.

➡ From now on we'll refer to this as the **CRB** button.

<p>The second page of the wizard is where the project name and folder are selected. Other details are filled in automatically.</p>	<p>The Projects Panel now shows the new project and a folder called Sources.</p>
	
<p>Clicking on the Sources folder reveals the two source code files used by the project. <i>main.agc</i> will contain your code.</p>	<p>Double clicking on <i>main.agc</i> in the Project Panel opens its contents in a tabbed panel within the main edit window.</p>
	 <pre> 1 rem AGK Application 2 rem 3 rem 4 rem A Wizard Did It! 5 do 6 setup.agc 7 Print("hello world") 8 Symt() 9 loop </pre>
<p>In fact, the AGK wizard has created a simple program within <i>main.agc</i>. This code can be run by pressing the Run button.</p>	<p>The sample program opens a small window to display its output.</p>
	

The new window created by running the sample program can be closed in the standard way by clicking on the X button at the top right.

This project will be used for the remaining programming activities in this chapter.

Activity 2.2

Before you start up AGK, create a main folder called *HandsOnAGK* on your disk drive. We'll use this as the main folder for all the AGK projects we are going to create throughout this book.

Load AGK then create, compile, and run your first project (named *FirstProject*) exactly as described in FIG-2.4, closing the app window once it has been run.

You may have noticed that the AGK software displayed messages in the compiler output area at the bottom of the screen (titled as *Logs & others*) to tell you that the app had been compiled and broadcast.

The Program Code

FIG-2.5 shows the code in *main.agk* that was automatically generated for us.

FIG-2.5

The Generated Code

```
rem
rem AGK Application
rem
rem A Wizard Did It!
do
  Print("hello world")
  Sync()
loop
```

The line numbers that also appear in the edit window are not part of the code and are only there to help you identify the position of any line within the code.

Let's take a look at the code that was already generated for us and see what each of the lines means. The first lines are:

```
rem
rem AGK Application
rem

rem A Wizard Did It!
```

Blank lines and any lines starting with the term `rem` (short for REMARK) are treated as a comment by the compiler. Comments are there only for the benefit of us humans who happen to read the program code and are entirely ignored by the compiler when translating the instructions into machine code. Good comments will tell us the overall aim of the program as well as the purpose of individual sections of code. Comments can appear anywhere within a program.

```
do

loop
```

These two terms mark the start and end of an infinite loop - notice that no condition is given. Most AGK programs contain this loop which is designed to make sure all the code between these lines is continually executed until the user closes the app window. Without a loop of some type your program would start and finish so quickly that you would never have time to see what was displayed in the app window.

```
Print ("Hello world")
```

The `Print()` statement is used to state that some piece of information is to be displayed in the app window. The information itself is specified within a set of round brackets (more properly called **parentheses**). When that information contains letters (as opposed to numbers), then those letters must be enclosed in double quotes. Hence, the statement given above is an instruction to display the words *Hello world* on the screen. Note that the quotes themselves are not displayed.

```
Sync()
```

The `Sync()` statement is a command to update the contents of the app window. If you make any changes to what is displayed on the screen (for example, by executing a `Print()` statement), then you need to follow this with the statement `Sync()`. Without `Sync()` the screen display will not be updated.

Notice that the `Sync()` statement makes use of parentheses although no values are placed within them. However, omitting these parentheses would create a syntax error.

Activity 2.3

Change the `Print()` statement within `main.agc` so that the text enclosed in the double quotes reads *My first app*. This time click the **Compile** then **Run** buttons to compile and run the modified program. Was the new text displayed in the app window?

Select **File|Save** to save your modified program.



Compile Button



Run Button

Using the **Compile** button and then **Run** button separates the compilation and execution stages of the process into two distinct steps.

Running Your App to a Tablet or Smartphone

Although producing a true app for your smartphone or tablet is quite complex, you can, nevertheless, watch your app run on such a device. To do this, you need to first load the app AGK Player from the app store used by your device. For example, on an Android device, you will find AGK Player in Google Play.



Compile and Broadcast Button

With AGK Player running on your target device and the app code you want to run on it loaded into AGK on your PC, press AGK's **Compile and Broadcast** button.

This will transfer the AGK program from the PC to your device through your WiFi setup. That means you either need to have a WiFi router attached to your PC or be using a laptop with built-in WiFi. The AGK Player app will detect your program being broadcast, download it, and run it on your device.

However, things are a bit more complicated if you have an Apple device. Apple won't support the AGK Player in their app store. As an alternative you can download the AGK Viewer from their store. The viewer isn't ideal but it will let you see a low-grade version of your app running on an Apple device. To run the AGK Player on your Apple device you will need to register as a developer. Details of how to do this are on the Game Creators' web site.

Activity 2.4

Make sure you have the AGK app player running on your device.

With the latest version of the project you created in Activity 2.3 showing on the AGK IDE, press the **Compile and Broadcast** button. Check that the program is now showing on your device.

Your program is not yet a true app - you can't save it on your device - it can only be executed using AGK Player. To create a true app for your device visit The Game Creators' web site for details.

First Statements in AGK BASIC

Introduction

Learning to program in AGK BASIC is very simple compared to other languages such as C++ or Java. Unlike most other programming languages, it has no rigid structure that the program itself must adhere to.

Now we need to start looking at the formal statements allowed in AGK BASIC and see how they can be used in a program.

Print()

We've already come across the `Print()` statement in our first program, so we already know that it is used to display information on the screen, but we need to know its exact format so that we don't create a syntax error by making a mistake in constructing the statement. The format of the `Print()` statement is shown in FIG-2.6.

FIG-2.6



`Print()`

This type of diagram is known as a **syntax diagram** for the obvious reason that it shows the syntax of the statement.

Each enclosed value in the diagram is known as a **token** (there are four tokens in the `Print()` statement). When you use a `Print()` statement in your program, its tokens must conform to those shown in the diagram. Some of the tokens must be an exact match for those in the diagram: **Print**, (, and) while others (only **value** in this case) have their actual value determined by the programmer.

Fixed values are shown in rounded-corners boxes, user-defined values are shown in regular boxes. In the case of the `Print()` statement, the term *value* is used to mean an integer value, a real value or a string value.

So, using the syntax diagram as a guide, we can see that the following are valid `Print()` statements:

```
Print("Hello world")
Print(12)
Print(0)
Print(-34.6)
```

while the following are not:

```
Print 36           (parentheses are missing)
Print(Goodbye)    (no quotes)
Print('Help!')   (single quotes used)
```

Activity 2.5

Which of the following are NOT valid `Print()` statements:

- a) `Print("-9.7")`
- b) `Print(0.0)`
- c) `Print(23, 51)`

Spaces

We can add spaces to a statement as long as those spaces do not split a single token into separate parts. So, for example, it is quite valid to write the line

```
Print ( 123 )
```

since each token can easily be identified, but

```
Pr int ( 12 3 )
```

is not acceptable because the `Print` and `123` tokens have both been split into two parts.

Spaces can be omitted as long as doing so does not make it impossible to tell where one token ends and another begins. This is really only a problem when two or more adjacent tokens are constructed entirely from letters or numbers. So if we have a statement which begins with the code

```
if x = 3
```

then writing

```
ifx=3
```

would be invalid because the compiler would not be able to recognise the `if` and `x` as two separate tokens. On the other hand,

```
Print(123)
```

is correct because no adjacent tokens are constructed from alphanumeric characters.

Multiple Output

When we use two or more `Print()` statements, each value printed will be displayed on a separate line. For example, when the lines

```
Print("Hello")
Print("Goodbye")
```

are included in a program, they will create the output

```
Hello
Goodbye
```

Activity 2.6

Modify your program so that the main code now reads

```
do
    Print("First line")
    Print("Second line")
    Sync()
loop
```

Compile and run the program.

► Alphabetic and numeric characters are collectively known as **alphanumeric characters**.

You may want to save your project after each Activity by selecting

File|Save

Each message is on a separate line because the `Print()` statement always displays a new line character after the value specified and this causes the screen cursor to move to a new line.

Adding Comments

It is important that you add comments to any programs you write. These comments should explain the purpose of the program as a whole as well as what each section of code is doing. It's also good practice, when writing longer programs, to add comments giving details such as your name, date, programming language being used, hardware requirements of the program, and version number.

In AGK BASIC there are four ways to add comments:

FIG-2.7

rem

Add the keyword `rem`. The remainder of the line becomes a comment (see FIG-2.7).

`rem` `text`

FIG-2.8

Apostrophe
Comments



Add an apostrophe character (you'll find this on the top left key, just next to the 1). Again the remainder of the line is treated as a comment (see FIG-2.8).

`'` `text`

FIG-2.9

// Comments

Add two forward slashes followed by the descriptive text (see FIG-2.9).

`//` `text`

FIG-2.10

remstart..remend

Add several lines of comments by starting with the term `remstart` and ending with `remend`. Everything between these two words is treated as a comment (see FIG-2.10).

`remstart`
`text`
`remend`

This last diagram introduces another symbol - a looping arrowed line. This is used to indicate a section of the structure that may be repeated if required. In the diagram above it is used to signify that any number of comment lines can be placed between the `remstart` and `remend` keywords. For example, we can use this statement to create the following comment which contains three lines of text:

```
REMSTART
  This program is designed to play the game of
  battleships.
  Two peer-to-peer computers are required.
REMEND
```

PrintC()

The `PrintC()` statement is similar to `Print()` but does not add a new line character to the output. This means that each `PrintC()` statement's output is positioned on the screen immediately after the previous value. Hence,

```
PrintC("A")
```



```
PrintC("B")
```

would display AB

Activity 2.7

Change the two `Print()` statements in your program to `PrintC()` statements and observe the difference in output when the program is run.

Other Statements which Modify Output

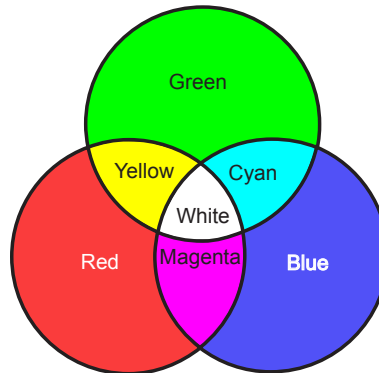
Other statements allow us to make various changes to how the information appearing on our screen is presented. We can change its colour, size, transparency and even the space between the characters.

Before we get started on instructions involving colour, perhaps it might be useful to go over a few basic facts about colour.

All colours you see on a monitor or TV are derived from the three primary colours red, green and blue. By varying the brightness of each of these three colours we can achieve almost any colour or shade the eye is capable of seeing. For example, mixing just red and green gives us yellow; blue and green gives us a colour called cyan, and blue and red gives magenta (see FIG-2.11).

FIG-2.11

Colours



Notice that all three colours together give white. The absence of all three colours gives black.

By varying the intensity (brightness) of each primary colour, we can create any shades or hues we require. AGK allows the intensity to vary between 0 (no colour) to 255 (full intensity). So pure white is achieved by setting all three colours to an intensity value of 255. For shades of grey, all three colours must have identical brightness values, but the lower that value, the darker the shade of grey.

SetPrintColor()

The `SetPrintColor()` sets the colour of all output created using the `Print()` and `PrintC()` statements. It can also be used to set the transparency of the text.

FIG-2.12

SetPrintColor()



This syntax diagram introduces the use of square brackets. Tokens within square brackets are optional and can be omitted when using the statement.

In the above diagram:

► The value names start with i to indicate that integer values are required. Where a real number is needed, the value name will start with an f (for *float*). String values will start with an s.

ired	is an integer value giving the strength of the red component within the colour. This value should be in the range 0 to 255. 0 - no red; 255 - full red.
igreen	is an integer value (0 to 255) giving the strength of the green component.
iblue	is an integer value (0 to 255) giving the strength of the blue component.
itrans	is an integer value (0 to 255) giving the amount of transparency. 0 - invisible, 255 - opaque.

Since the transparency value is optional and therefore can be omitted (in which case transparency stays at its current setting), we can use the statement simply to set the colour of any text being displayed by the `Print()` or `PrintC()` statements.

For example,

```
SetPrintColor(0,0,0)      rem *** sets text to black
SetPrintColor(255,255,255) rem *** sets text to white
SetPrintColor(255,0,0)   rem *** sets text to red
```

The `SetPrintColor()` statement must appear before the `Print()` or `PrintC()` statements whose output you wish to affect.

The statement only takes effect after a `Sync()` statement is executed.

Activity 2.8

Add a `SetPrintColor()` statement to your program, placing it immediately before your two `PrintC()` statements. Choose any colour values you wish.

Compile and run the program to check that the output is correct.

Once the colour has been set, all subsequent output will be in the specified colour. This means that there is no real need to place the `SetPrintColor()` statement inside the `do .. loop` structure where it will be executed every time the loop is repeated. Instead, that line of code can be moved to immediately before the `do` statement. Placed here, the statement will be performed only once, at the start of the program.

Activity 2.9

Reposition your `SetPrintColor()` statement, placing it on the line above `do`.

Compile and run the program again.

There should be no change to the output.

If there was no change to the output, what was the point of moving the statement?

The more lines of code that need to be executed, the slower a program runs. Let's say the statements within the loop are executed 200 times before you terminate the program. With the `SetPrintColor()` inside the loop, it would have been executed 200 times; with it outside the loop it is executed only once - so the program becomes more efficient.

If we include a value for *itrans* when we use `SetPrintColor()`, we can set the transparency of all text on the screen. The default transparency is 255, meaning the output is fully opaque. With a value of zero, the text would be invisible.

Activity 2.10

Modify the `SetPrintColor()` statement in your program, adding 126 as the transparency value.

Run the program and see what effect the changes have made to the output.

Try other transparency values to see their effect.

SetPrintSize()

The `SetPrintSize()` statement (see FIG-2.13) sets the size of the text displayed by a `Print()` or `PrintC()` statement.

FIG-2.13

SetPrintSize()

`SetPrintSize` () `size` ()

where:

size is a real number setting the size of characters. The default value for characters is about 3.5.

Activity 2.11

Add the line

```
SetPrintSize(8.6)
```

immediately after your `SetPrintColor()` statement (reset the transparency value to 255).

Compile and run the program. What do you notice about the quality of the text produced?

The reason that the text seems blurred when it is enlarged is that the text itself is stored as an image. Enlarging that image causes blurring.

SetPrintSpacing()

This statement (see FIG-2.14) adjusts the spacing between the characters shown on the screen.

FIG-2.14

SetPrintSpacing()

`SetPrintSpacing` () `gap` ()

where:

gap is a real number giving the gap between characters. The default

is zero. Larger values widen the gap; negative values cause the gap to decrease and even to make letters overlap.

Activity 2.12

Add a `SetPrintSpacing()` statement to your program, placing it before the `do .. loop` structure. Set the gap size to 5.5.

Compile and run the program to check how the output is changed.

Change the value used to -2.5 and observe the effect on the output.

Message()

Another way of displaying text on the screen is to use the `Message()` statement. This creates a more prominent output, placing the text in a separate window. The format of the `Message()` statement is shown in FIG-2.15.

FIG-2.15

`Message()`

```
Message ( ( stext ) )
```

where

stext is a string containing the message to be displayed.

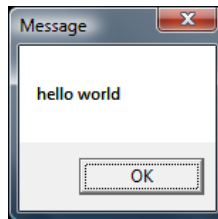
For example, the line

```
Message("hello world")
```

produces the output shown in FIG-2.16 when run on a PC.

FIG-2.16

A Typical
Message Window



The exact style of the window produced depends on the device on which your app is being run.

SetClearColor()

You will have noticed that the window created by your AGK app always has a black background. This default color can be changed using the `SetClearColor()` statement which has the format shown in FIG-2.17.

FIG-2.17

`SetClearColor()`

```
SetClearColor ( ( ired , igreen , iblue ) )
```

where:

ired is an integer value (0 to 255) giving the strength of the red component.

igreen is an integer value (0 to 255) giving the strength of the green component.

ibblue is an integer value (0 to 255) giving the strength of the blue component.

ClearScreen()

The `SetClearColor()` statement only works when followed by a `Sync()` or a `ClearScreen()` statement which has the same effect. The format for the `ClearScreen()` statement is given in FIG-2.18.

FIG-2.18

ClearScreen()

ClearScreen ()

So to create a yellow background on the screen, we would start our program with the lines:

```
SetClearColor()  
ClearScreen()
```

Often this statement will appear at the start of a program, but you may wish to change the colour at a later stage perhaps to indicate that a game has entered a new phase.

Activity 2.13

Change the background of the app window to red and test your program.

Positioning the Print() Statements

We have placed the various statements affecting the colour, size and spacing of our text before the `do..loop` structure on the basis that these commands need only be performed once. So you may be tempted to think that surely we can do the same thing with the `Print()` and `Sync()` statements since the displayed text remains unchanged throughout the running of the program. Let's see what happens when we try this.

As you can see from the output produced, for a simple program such as this, moving the statements has had no effect on the output produced. We are left with an empty `do..loop` which makes sure that the program does not terminate before we click the app window's close button.

Activity 2.14

Move the `PrintC()` and `Sync()` statements in your program so that they are positioned immediately before the `do` statement.

What effect does this have when you run your program?

Although we now know that it is possible to place the `Print()` and `Sync()` statements outside the `do` loop it is usually not a good idea to do so in any but the simplest programs since it can create other problems which we will discuss in a later chapter.

Summary

- Programs are written using a programming language.
- Programming language code must be translated into machine code before the program can be executed by the computer.
- The stored program code is known as the source file; the stored machine code

as the object file.

- Each line of a program must conform to the rules of syntax.
- An error in how a line is written is known as a syntax error.
- AGK programs can be written in BASIC or C++.
- The collection of files created when writing an AGK app is known as a project.
- The main file in an AGK project is *main.agc* which contains the program code.
- The AGK development package is an Integrated Development Environment. This allows edit, compiling and testing to be performed from within the same program.
- To download an app to your digital device, the player must be installed and running on that device and the app broadcast from the AGK IDE.
- When an app is being tested it creates an app window.
- Comments can be added to your code using `rem`, ```, or `remstart..remend`.
- Comments help us understand the purpose of a piece of code but are ignored by the compiler.
- Use `Print()` to display information on the screen.
- Use `PrintC()` to display information without moving to a new line afterwards.
- Use `SetPrintColor()` to set the colour used when displaying text.
- Use `SetPrintSize()` to set the size of future text output.
- Use `SetPrintSpacing()` to set the spacing between characters in future text output.
- Use `Message()` to display a message in a separate window.
- Use `SetClearColor()` to set a background colour for the app screen.

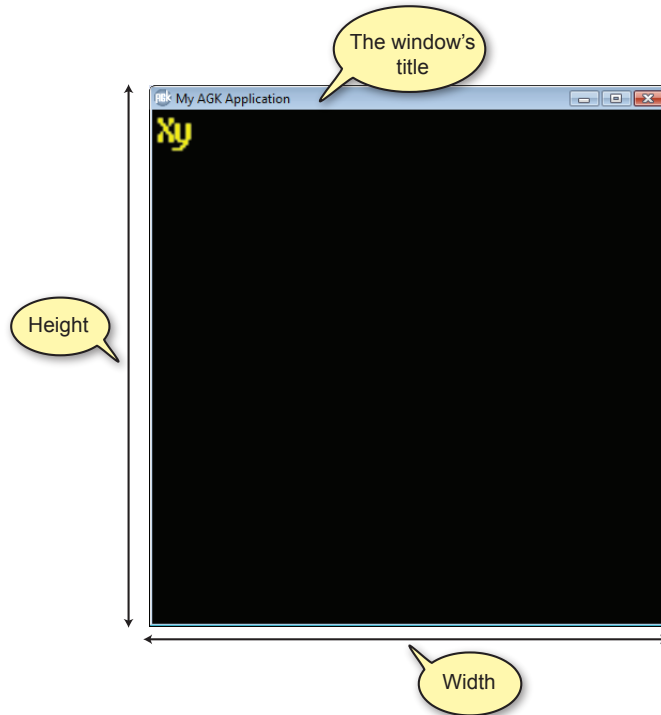
The Second Source File

Every project you create actually contains a second *.agc* file. You can see it listed in the Projects Panel immediately below *main.agc*.

Although you are not free to add lines of code to this file as you can with *main.agc*, you are allowed to change the values given. Those values determine the title and dimensions of the window in which your app appears when run under Microsoft Windows. For example, the window of a typical program (see FIG-2.16) reflects the details given in *setup.agc*.

FIG-2.16

The App Window



By changing the values specified in the first three lines of *setup.agc* (ignoring the `rem` lines), we can change the characteristics of the window.

Activity 2.15

Double click *setup.agc* in the Projects Panel to display its code. Change the appropriate existing lines to read:

```
title=My First App
width=320
height=480
```

Make sure the only spaces with these lines are those in the title.

Compile and run your program to see what changes this has made.

These characteristics given in *setup.agc* only affect the layout of the window on your PC. Other statements (covered later) need to be included in your program to set the app screen size on a tablet or phone.

A Splash Screen

A common feature of many games is a **splash screen**. A splash screen is simply a graphic that displays for a few moments at the start of the game. Typically a splash screen will contain an image giving the flavour of the game play that is about to follow as well as the name of the game and the publishing company.

AGK allows you to add a splash screen to your game without any coding whatsoever.

If you load Windows Explorer and have a look in the folder created by AGK to hold the files belonging to your project (*HandsOnAGK/FirstProject*), you should see contents similar to that shown in FIG-2.17.

FIG-2.17

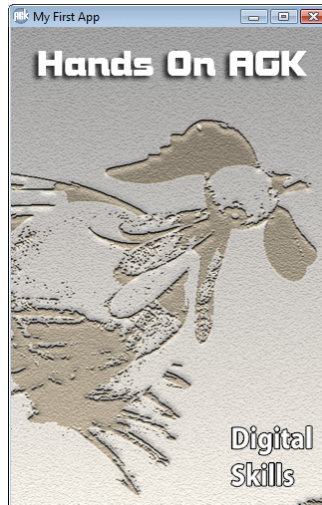
AGK Project's Files

Name	Date modified	Type	Size
media	19/07/2011 07:50	File Folder	
FirstProject.byc	19/07/2011 07:50	BVC File	2 KB
FirstProject.cbp	17/07/2011 15:53	CBP File	1 KB
FirstProject.dbpro	19/07/2011 07:50	DBPRO File	1 KB
FirstProject.exe	08/07/2011 14:55	Application	1,099 KB
FirstProject.layout	19/07/2011 07:59	LAYOUT File	1 KB
main.agc	19/07/2011 07:50	AGC File	1 KB
setup.agc	27/06/2011 17:16	AGC File	1 KB

The splash screen graphic file must be placed in the project's main folder. The file must be in **PNG** format and be called *AGKSplash.png*. No other name is acceptable. The image is best set to the same size as the window dimensions (in our case, 480 x 320). An example of a splash screen is shown in FIG-2.18.

FIG-2.18

A Splash Screen



Rather than create your own image, you can use the one supplied in the downloads that accompany this book.

Activity 2.16

Open a paint program you have available and create a 480 pixels high by 320 pixels wide image. Save the file in PNG format in the folder *HandsOnAGK/FirstProject* naming the file *AGKSplash.png*.

In AGK, recompile your program and run it. You should see your splash screen appear when the app window first opens.

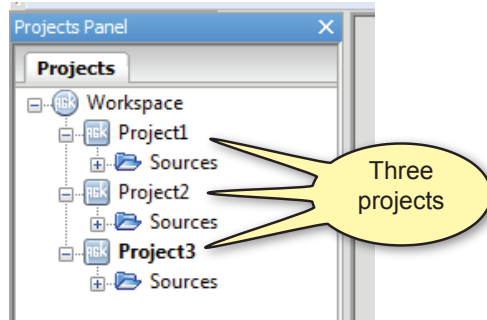
Starting a New Project

When you first start up AGK for a work session, we've already seen that it will give you the option to create a new project. Should you want to create more new projects during that session, you can do so from the main menu (**File|New|Project**).

However, the Projects Panel will display all of the projects you have been using (see FIG-2.19).

FIG-2.19

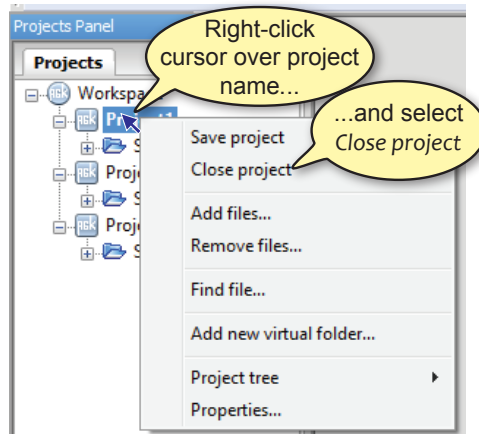
Multiple Projects



Having several projects open at the same time can be a bit confusing when you first start using AGK, so the best option is to close projects that you are not currently working on. FIG-2.20 shows how to close a project from the Projects Panel.

FIG-2.20

Multiple Projects



From now on, make sure you always close any old project before starting a new one.

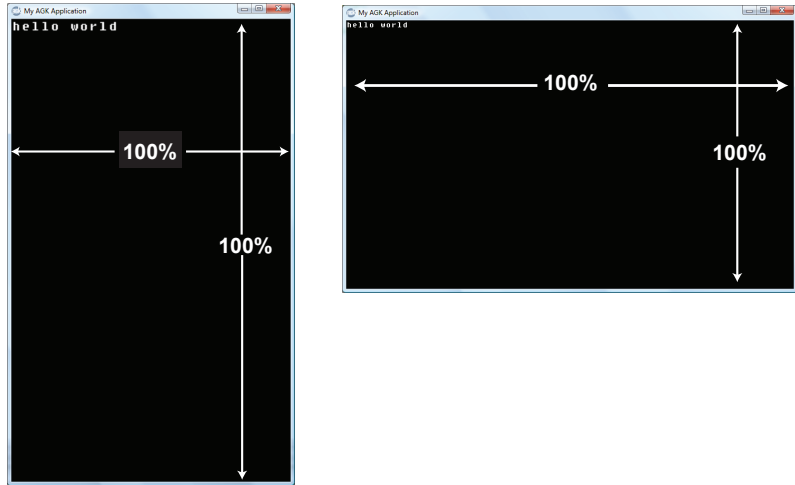
App Window Properties

Measurements

By default, AGK apps use a percentage measurement system. This means that no matter the actual dimensions of the app window, AGK always treats the width as 100% and the height as 100% (see FIG-2.21).

FIG-2.21

The Screen's Percentage Measurement System



When you want to position an item on the screen it is done using percentage measurements. For example, the position (50,50) represents the middle of the app window irrespective of the window's actual dimensions.

Percentage values are also used when setting the size of various visual elements. For example, earlier in this chapter we made use of the `SetPrintSize()` statement to resize the text created by any subsequent `Print()` statement. The value supplied to this statement represents the high of the text as a percentage of the screen height. Of course, this means that text set to a height of 4 will appear taller in a long window and smaller in a short window. In fact, you can see that in the “Hello world” text visible in FIG-2.21 above.

All programs in this book use the default percentage system.

SetDisplayAspect()

When using the percentage measuring system, the `setup.agc` file is used to set the actual size of the app window on your PC, but if you intend to transfer that app to another device such as a smartphone or tablet, you should explicitly set the aspect ratio (width to height) using the `SetDisplayAspect()` statement (see FIG-2.22).

FIG-2.22

SetDisplayAspect()

```
SetDisplayAspect ( ratio )
```

where:

ratio is a real number giving the width to height ratio. For example, iPhone and iPad have an aspect ratio of 4.0/3.0 (1.3333).

Use zero as the *ratio* value if you want the width and height values in the `setup.agc` file to be used to determine the aspect ratio. Use -1 if you want the app to fill the whole screen irrespective of aspect ratio. Using this last option may distort visual

elements of the app if the device's aspect ratio is different to that used when developing the app (like watching an old 4/3 TV program on your widescreen TV).

SetVirtualResolution()

If you would rather work with a resolution based on pixels, have your program execute the `SetVirtualResolution()` statement when it starts up. The statement's format is shown in FIG-2.23.

FIG-2.23

SetVirtualResolution()

```
SetVirtualResolution ( ( iwidth , iheight ) )
```

where:

iwidth is an integer value giving the nominal width of the app window in pixels.

iheight is an integer value giving the nominal height of the app window in pixels.

If you were writing an app for the original iPhone, you would set the resolution to 320×480 using the line:

```
SetVirtualResolution(320,480)
```

When you are developing your app on your PC, the app window will take on the actual size specified in the `SetVirtualResolution()` statement. However, when you transfer the app to another device, the app will expand (or contract) to fit that device's screen. For example, if you run your 320×480 app on a newer iPhone with its 640×960 resolution, your AGK will automatically expand to fill the full screen.

This is why the term **virtual resolution** is used; this development resolution may in fact be different from the actual resolution used when the app is running on a device other than your PC.

The only problem arises when the device on which your app is running has a different aspect ration (width / height) than that specified in the `SetVirtualResolution()` statement. Expanding the app's resolution from 320×480 to 640×960 isn't a problem because both have an aspect ratio of 3/4. But if we were to try and run the same app on an original Asus EEE Transformer which has a resolution of 1280×800 (an aspect ratio of 8/5) then we have a problem. Expanding the app to fill a 8/5 screen would cause distortion of any images being displayed (circles would become ovals!). AGK handles this by creating as large a 3/4 ratio images as possible and adding a border to the remainder of the screen.

When you use `SetVirtualResolution()` in your app, all screen positions and sizes are given in **virtual pixels**.

SetBorderColor()

You can specify the border colour to be used when you app runs on a device with a different aspect ratio to that specified in the app's code using the `SetBorderColor()` statement (see FIG-2.24).

FIG-2.24

SetBorderColor()

```
SetBorderColor ( ( ired , igreen , iblue ) )
```

where:

- ired** is an integer variable (0 to 255) giving the intensity of the red component of the border colour to be used. 0: no red; 255: full red.
- igreen** is an integer variable (0 to 255) giving the intensity of the green component of the border colour. 0: no green; 255: full green.
- ibblue** is an integer variable (0 to 255) giving the intensity of the blue component of the border colour. 0: no blue; 255: full blue.

To create a grey border we could use a statement such as:

```
SetBorderColor(120,120,120)
```

SetWindowTitle()

For apps that are running in a windows based environment (on PCs or Macs), you can set the title that appears at the top of the window using the `SetWindowTitle()` statement (see FIG-2.25).

FIG-2.25

SetWindowTitle()

```
SetWindowTitle ( stext )
```

where

- stext** is a sting containing the text to appear in the window title bar.

A typical statement would be:

```
SetWindowTitle("Jigsaw Game")
```

Further screen-handling statements are covered in Chapter 19.

Summary

- By default, AGK uses a percentage coordinate system within the app window.
- Use `SetVirtualResolution()` to use a virtual pixel coordinate system.
- Use `SetDisplayAspect()` to set the width to height ratio of the screen/window.
- Use `SetBorderColor()` to specify a colour for any part of the physical screen of included in the app's output area.
- Use `SetWindowTitle()` to specify a title for any windows-based app.

Solutions

Activity 2.1

- Machine code instruction. These are stored as a sequence of binary digits.
- A compiler.
- A syntax error.

Activity 2.2

No solution required.

Activity 2.3

Your code should now read (rem statements have been omitted):

```
do
  Print("My first app")
  Sync()
loop
```

Compile and run your code.

The new text should be displayed in the app window when the program is run.

Select **File|Save**

Activity 2.4

Activity 2.5

- Valid. Any characters can be enclosed in quotes - including numeric ones.
- Valid. A real number.
- Invalid. Only a single value can be displayed.

Activity 2.6

Your program code should be:

```
do
  Print("First line")
  Print("Second line")
  Sync()
loop
```

The output should be:

```
First line
Second line
```

Activity 2.7

Program code:

```
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The output should be:

```
First lineSecond line
```

If you want a space between the two outputs, you would need to include a space inside the quotes at the end of the first piece of text or at the start of the second.

Activity 2.8

Program code (your colour values will be different):

```
do
  SetPrintColor(255,255,0) rem *** yellow ***
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

Activity 2.9

Program code (your colour values will be different):

```
SetPrintColor(255,255,0) rem *** yellow ***
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

Activity 2.10

Program code (your colour values will be different):

```
SetPrintColor(255,255,0,126) rem *** yellow ***
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The text output will appear darker as the black background shows through.

Activity 2.11

Program code (your colour values will be different):

```
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The text will appear larger but somewhat blurred.

Activity 2.12

Program code (your colour values will be different):

```
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
SetPrintSpacing(5.5)
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

The characters in the output text will be widely spaced.

The `SetPrintSpacing()` line should then be changed to

```
SetPrintSpacing(-2.5)
```

The characters will now bunch together.

Activity 2.13

Program code:

```
SetClearColor(255,0,0)
ClearScreen()
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
SetPrintSpacing(-2.5)
do
  PrintC("First line")
  PrintC("Second line")
  Sync()
loop
```

Activity 2.14

Program code:

```
SetClearColor(255,0,0)
ClearScreen()
SetPrintColor(255,255,0,126) rem *** yellow ***
SetPrintSize(8.6)
SetPrintSpacing(-2.5)
PrintC("First line")
PrintC("Second line")
Sync()
do
loop
```

The output remains unchanged.

Activity 2.15

The app window title and dimensions should be changed.

Activity 2.16

No solution required.