Hands On DarkBASIC Pro

Volume 1

A Self-Study Guide to Games Programming

Alistair Stewart

Hands On DarkBASIC Pro

Volume 1

A Self-Study Guide to Games Programming

Alistair Stewart

Digital Skills
Milton
Barr
Girvan
Ayrshire
KA26 9TY

www.digital-skills.co.uk

DarkBASIC Professional is produced by The Game Creators Ltd

Printed September 2005
2nd Printing November 2005
3rd Printing January 2006

Title : Hands On DarkBASIC Pro Volume 1

ISBN : 1-874107-08-4

Other Titles Available:

Hands On Pascal
Hands On C++
Hands On Java
Hands On XHTML

Table Of Contents

Chapter 1 Designing Algorithms

Chapter 2 Starting DarkBASIC Pro

Chapter 3 Data

Chapter 4 Selection

Chapter 5 Iteration

Chapter 6 Drawing Statements

Chapter 7 Modular Programming

Chapter 8 String Functions

Chapter 9 Hangman

Chapter 10 Arrays

Chapter 11 Bull and Touch

Chapter 12 Advanced Data Types and Operators

Chapter 13 Bitmaps

Chapter 14 Video Cards and the Screen

Chapter 15 File Handling

Chapter 16 Handling Music Files

Chapter 17 Displaying Video Files

Chapter 18 Accessing the Keyboard

Chapter 19 Mathematical Functions

Chapter 20 Images

Chapter 21 Sprites1

Chapter 22 Sprites 2

Chapter 23 Animated Sprites

Chapter 24 Sound

Chapter 25 2D Vectors

Chapter 26 Space Duel

Chapter 27 Using the Mouse

Chapter 28 Pelmanism

Chapter 29 Using a Joystick

Acknowledgements

I would like to thank all those who helped me prepare the final draft of this book.

In particular, Virginia Marshall who proof-read the original script and Michael Kerr who did an excellent job of checking the technical contents.

Any errors that remain are probably due to the extra few paragraphs I added after all the proof-reading was complete!

Thanks also to The Game Creators Ltd for producing an excellent piece of software - DarkBASIC Professional - known as DarkBASIC Pro to its friends.

Finally, thank you to every one of you who has bought this book. Any constructive comments would be most welcome.

Email me at *alistair@digital-skills.co.uk.*

Introduction

Welcome to a book that I hope is a little different from any other you've come across. Instead of just telling you about software design and programming, it makes you get involved. There's plenty of work for you to do since the book is full of exercises - most of them programming exercises - but you also get a full set of solutions, just in case you get stuck!

Learn by Doing

The only way to become a programming expert is to practice. No one ever learned any skill by just reading about it! Hence, this is not a text book where you can just sit back in a passive way and read from cover to cover whilst sitting in your favourite chair. Rather it is designed as a teaching package in which you will do most of the work.

The tasks embedded in the text are included to test your understanding of what has gone before and as a method of helping you retain the knowledge you have gained. It is therefore important that you tackle each task as you come to it. Also, many of the programming exercises are referred to, or expanded, in later pages so it is important that you are familar with the code concerned.

What You Need

You'll obviously need a PC and a copy of DarkBASIC Pro.

You don't need any experience of programming, but knowing your bits from your bytes and understanding binary and hexadecimal number systems would be useful.

How to Get the Most out of this Text

Experience has shown that readers derive most benefit from this material by approaching its study in an organised way. The following strategy for study is highly recommended:

1. Read a chapter or section through without taking notes or worrying too much about topics that are not immediately clear to you. This will give you an overview of the contents of that chapter/section.

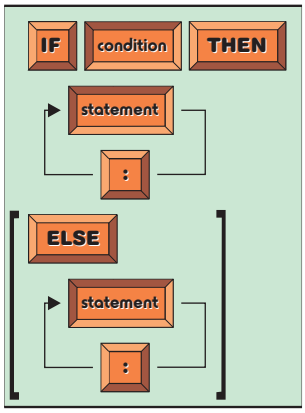
2. Re-read the chapter. This time take things slowly; make notes and summaries of the material you are reading (even if you understand the material, making notes helps to retain the facts in your long-term memory); re-read any parts you are unclear about.

3. Embedded in the material are a series of activities. Do each task as you reach it (on the second reading). These activities are designed to test your knowledge and understanding of what has gone before. Do not be tempted to skip over them, promise to come back to them later, or to make only a half-hearted attempt at tackling them before looking up the answer (there are solutions at the end of each chapter). Once you have attempted a task, look at the solution given. Often there will be important points emphasised in the solution which will aid higher understanding.

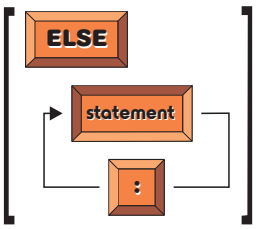
4. As you progress through the book, go back and re-read earlier chapters, since you will often get something new from them as your knowledge increases.

Language Syntax Diagrams

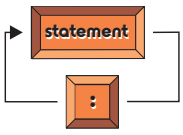
The text contains many syntax diagrams which give a visual representation of the format of various statements allowed in DarkBASIC Professional. These diagrams make no attempt to be complete, but merely act as a guide to the format most likely to be used. The accompanying text and example should highlight the more complex options available. Below is a typical diagram:



Each tile in the diagram holds a **token** of the statement. Raised tiles represent fixed terms in the statement, which must be entered exactly as shown. Sunken tiles represent tokens whose exact value is decided by you, the programmer, but again these values must conform to some stated rule.



Items enclosed in brackets may be omitted if not required. In this example we can see that ELSE and all the terms that follow may be omitted.



Where one or more tokens in a diagram may be repeated indefinitely, this is shown using the arrowed line. This example shows that any number of statements can be used so long as a colon appears between each statement.

Occasionally, a single line of code will have to be printed over two or more lines because of paper width restrictions; these lines are signified by a ↳ symbol. Enter these lines without a break when testing any of the programs in which they are used. For example, the code

```
SPRITE crosshairs,(JOYSTICK X()+1000)*xpixels#,
↳(JOYSTICK Y()+1000)*ypixels#,1
```

should be entered as a single line.

1

Designing Algorithms

Boolean expressions

Data Variables

Designing Algorithms

Desk Checking

IF Control Structure

FOR Control Structure

REPEAT Control Structure

Stepwise Refinement

Testing

WHILE Control Structure

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 is known as an **algorithm**.

There are a few points to note from the algorithm given above:

- There is one instruction per line
- Each instruction is unambiguous
- Each instruction is as short 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 managed 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**. DarkBASIC Pro 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.

Computer programs are normally copied (or **loaded**) from a magnetic 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 2,000 million (or 2 billion) instructions per second (2,000 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 games-related; an area of programming for which DarkBASIC Pro has been specifically designed.

Activity 1.3

1. A set of instructions that performs a specific task is known as what?

2. What term is used to describe a set of instructions used by a computer?

3. The speed of a computer is measured in what units?

Control Structures

Although writing algorithms and programming computers are certainly 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, the concepts are already familiar to you in everyday situations. If you examine any algorithm, no matter how complex, you will find it consists of three basic structures:

- **Sequence** where one statement 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 are explained in detail over the next few pages. All that is needed is to formalise the use of these structures 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 play Monopoly might begin with the sequence:

Choose your playing piece
Place your piece on the GO square
Get £1,500 from the bank

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 only be carried out 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 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

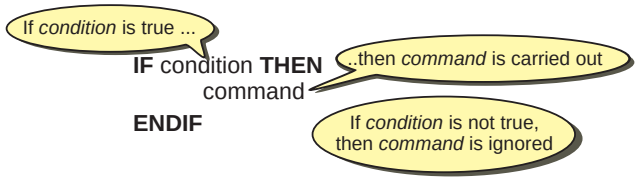
Notice that when we write a sentence containing the word IF, it consists of two main components:

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. The command given in the statement is only carried out if the condition is true and hence this type of instruction is known as an **IF** statement and the command as a **conditional instruction**. Although we could rewrite the above instruction in many different ways, when we produce a set of instructions in a formal manner, 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



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 conditional statement will almost certainly appear in a longer sequence 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
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
```

This longer sequence 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*. 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 that 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

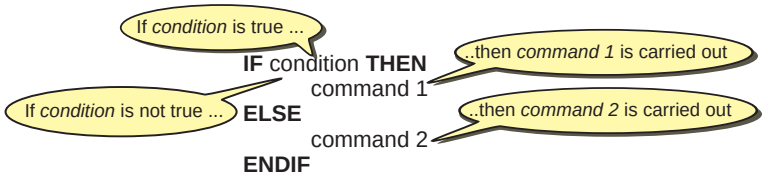
following statement:

```
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 ... ELSE Statement



Activity 1.6

Write an IF statement containing an ELSE section which describes the alternative actions to be taken when playing Hangman and the player trying to guess the word suggests a letter.

In the solution include the statements:

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

Choosing between two alternative actions is called **binary selection**.

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:

This set of instructions is not complete and is shown here only to illustrate the use of multiple IF statements in an algorithm.

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

Multi-way Selection

Although a single 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. After all, only one of the three conditions can be true at any one time.

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, are the other conditions 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). But what are these other conditions? They are *the guess is lower than the number you thought of* and *the guess is higher than the number you thought of*.

We already know how to handle a situation where there are only two alternatives: use an IF statement. So we can chose between *Too low* and *Too high* with 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:

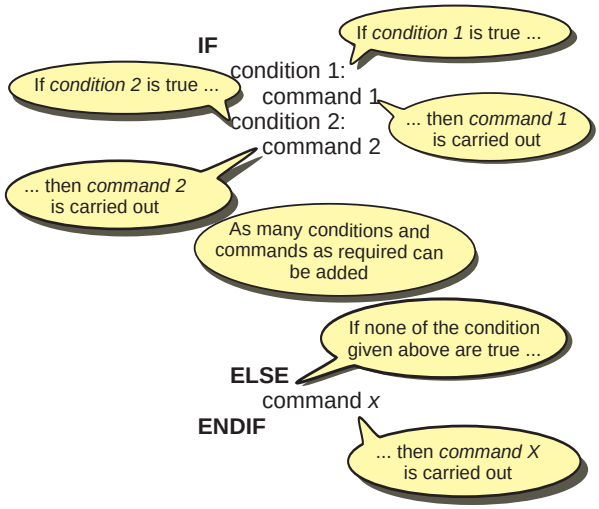
```
IF
    command is Go East:
        Move player's character eastward
    command is Go West:
        Move player's character westward
    command is Go North:
        Move player's character northward
    command is Go South:
        Move player's character southward
    ELSE
        Display an error 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 Third Version of the IF Statement



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
```

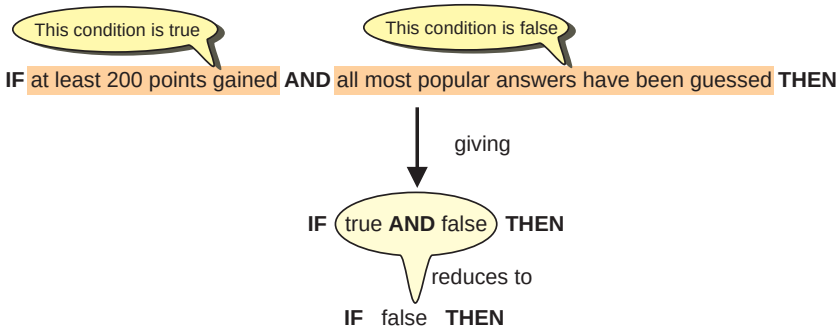
The result of the AND operator is determined using the following rules:

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 we assume the group reaching the final of the game show Family Fortunes has amassed 230 points but have not guessed all of the most popular answers, then a computer would determine the overall result of the IF statement given earlier as shown in FIG-1.4.

FIG-1.4

Calculating the Result of an AND Operation



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.

Activity 1.9

What are the other two possible combinations of true and false?

The results of the AND operator are summarised in TABLE-1.1.

TABLE-1.1

The AND Operator

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 the card game Snap, you win the cards on the table if you are first to place your hand over those cards, and the last two cards laid down are of the same value.

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 conditions 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 *GoTo Jail*

square or if you throw three doubles in a row. This can be written as:

```
IF player lands on Go To Jail OR player has thrown 3 pairs in a row THEN
    Player goes to jail
ENDIF
```

Like AND, the OR operator works on two operands:

```
condition 1   OR   condition 2
```

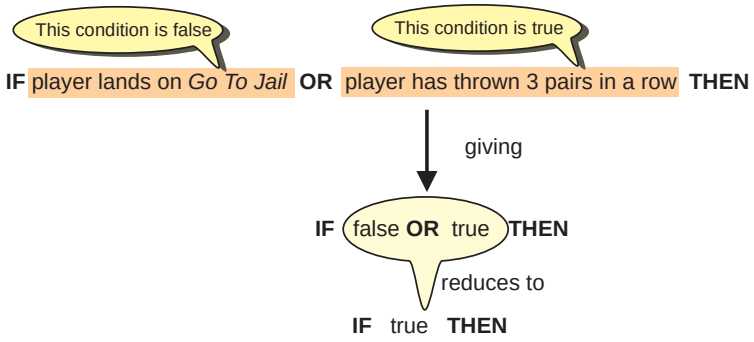
When OR is used, only one of the conditions involved needs to be true for the overall result to be true. Hence the results are determined by the following rules:

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-1.5.

FIG-1.5

Calculating the Result of an OR Operation



The results of the OR operator are summarised in TABLE-1.2.

TABLE-1.2

The OR Operator

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 *property mortgaged* is true, then *NOT property mortgaged* is false.

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

In Monopoly a player can charge rent on a property as long as that property is not mortgaged. This situation can be described with the statement:

```
IF NOT property mortgaged THEN
      Rent can be charged
ENDIF
```

The NOT operator works on a single operand:

```
NOT condition
```

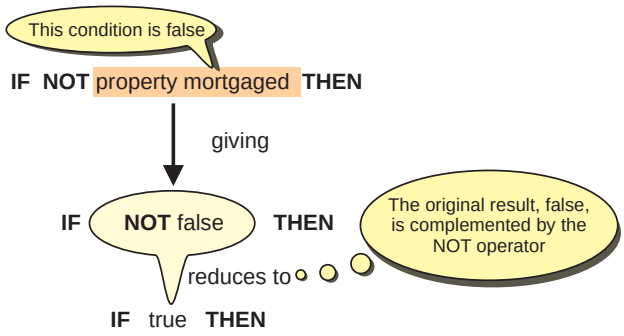
When NOT is used, the result given by the original condition is reversed. Hence the results are determined by the following rules:

1. Determine the truth of the 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-1.6.

FIG-1.6

Calculating the Result of a NOT Operation



The results of the NOT operator are summarised in TABLE-1.3.

TABLE-1.3

The NOT Operator

condition	NOT condition
false	true
true	false

Complex conditions are not limited to a single occurrence of a Boolean operator, hence it is valid to have statements such as:

```
IF player lands on Go To Jail OR player has thrown 3 pairs in a row OR
  player lifts a Go To Jail card
THEN
      Player goes to jail
ENDIF
```

Although us humans might be able to work all of this out in our heads without even a conscious thought, computers deal with such complex conditions in a slow, but methodical way.

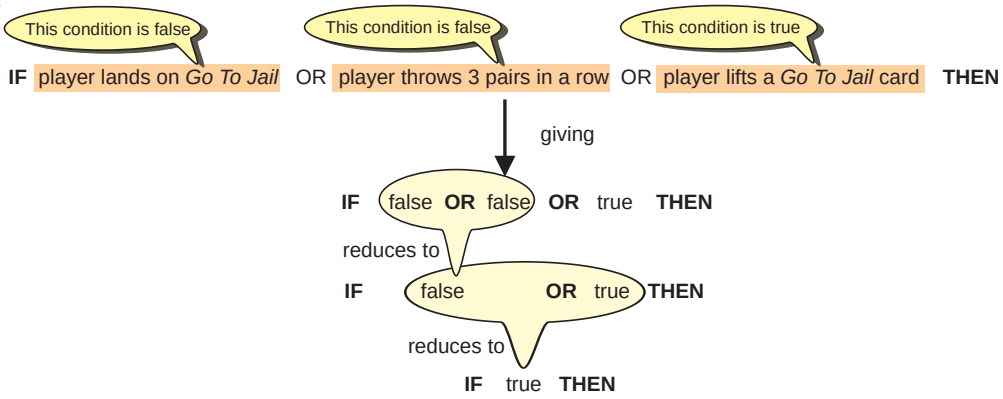
To calculate the final result of the condition given above, the computer requires several operations to be performed. These are performed in two stages:

1. Determine the truth of each condition
2. Determine the result of each OR operation, starting with the left-most OR

FIG-1.7

Using More than One OR Operator

For example, if a player lifts a *Go To Jail* card from the *Chance* pack, then the result of the IF statement given above would be determined as shown in FIG-1.7.



That might seem a rather complicated way of achieving what was probably an obvious result, but when the conditions become even more complex, this methodical approach is necessary.

Notice that when a complex condition contains only a single Boolean operator type (OR in the example above), that the expression is worked out from left to right. However, should the condition contain a mixture of OR, AND and NOT operators, NOT operations are performed first, ANDs second, and ORs last.

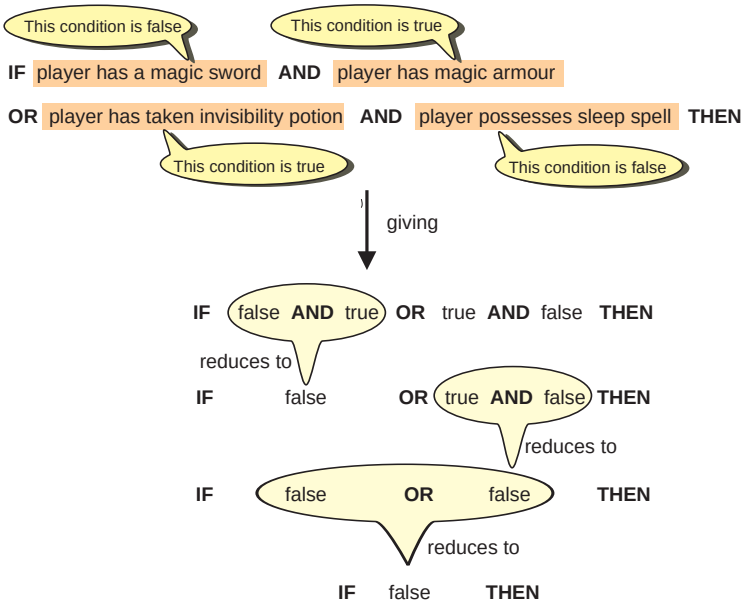
For example, if a game has the following rule

```
IF player has a magic sword AND player has magic armour OR
player has taken invisibility potion AND player possesses sleep spell
THEN
      Player can kill dragon
ENDIF
```

and a player has magic armour and has drunk the invisibility potion, then to determine if the player can kill the dragon, the process shown in FIG-1.8 is followed.

FIG-1.8

AND Operators have Priority



The final result shows that the player cannot kill the dragon.

Activity 1.12

A game has the following rule:

```
IF a player has an Ace AND player has King OR player has two Knaves THEN
    Player must pick up extra card
ENDIF
```

Using a similar approach to that shown in FIG-1.8 above, show the steps involved in deciding if the player should take an extra card assuming the player already has an Ace and one Knave.

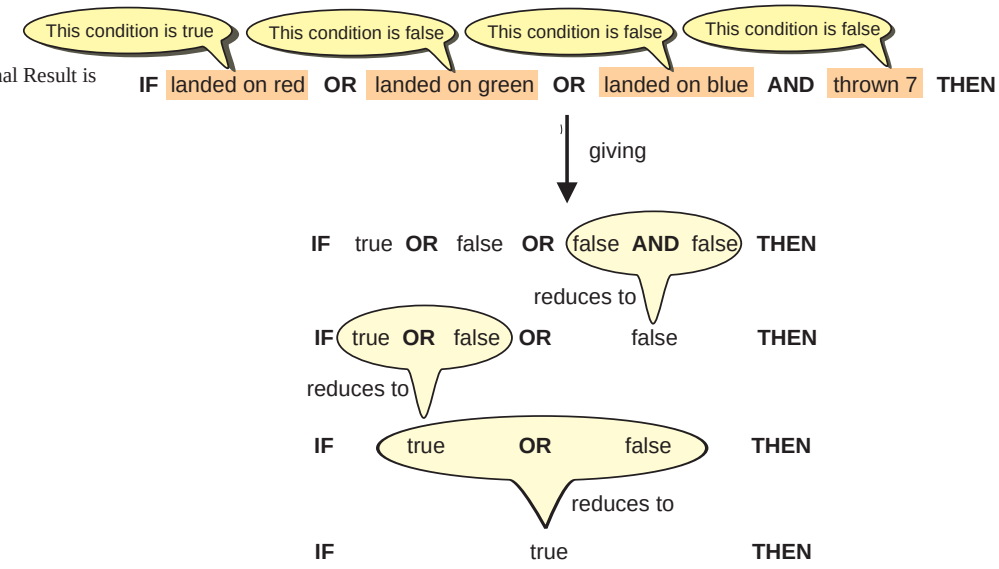
Sometimes the priority of operators works against what we are trying to express. For example, if a player receives a bonus if he lands on a red, green or blue square after throwing 7 on a pair of dice, then we might be tempted to write:

```
IF landed on red OR landed on green OR landed on blue AND thrown 7 THEN
    Add bonus to player's score
ENDIF
```

We would not expect a player landing on a red square after throwing 9 to receive the bonus. But, if we look at the calculation for such a situation, we get the result shown in FIG-1.9 which means that the bonus is incorrectly added to the player's score.

FIG-1.9

How the Final Result is Calculated



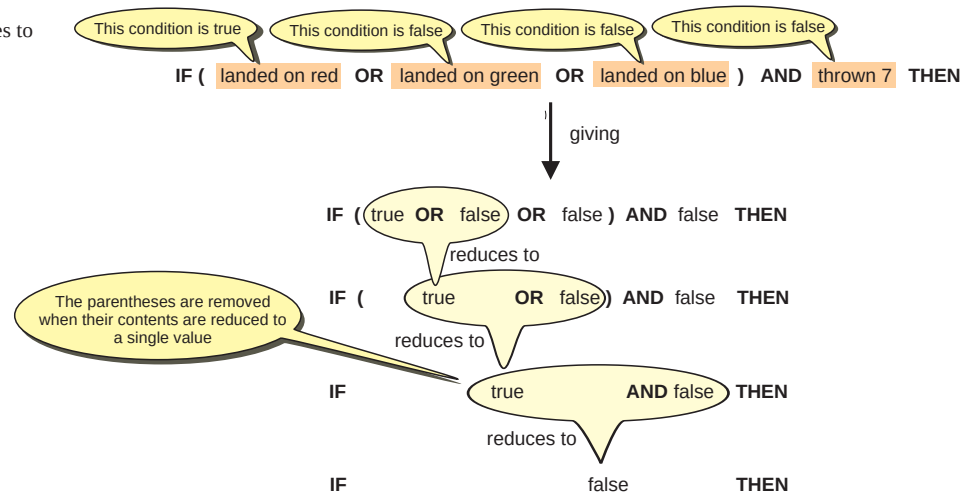
To achieve the correct results, we need the OR operations to be performed first and this can be done by giving the OR operators a higher priority than the AND. Luckily, operator priority can be modified by using parentheses. Operations in parentheses are always performed first. So, by rewriting our instruction as

```
IF (landed on red OR landed on green OR landed on blue) AND thrown 7 THEN
    Add bonus to player's score
ENDIF
```

the condition is calculated as shown in FIG-1.10.

FIG-1.10

Using Parentheses to Modify Operator Priority



Boolean operator priority is summarised in TABLE-1.4.

TABLE-1.4

Operator Priority

Priority	Operator
1	( )
2	NOT
3	AND
4	OR

Activity 1.13

The rules for winning a card game are that your hand of 5 cards must add up to exactly 43 (faces =10, Ace = 11) or you must have four cards of the same value. In addition, a player cannot win unless he has a Queen in his hand.

Express these winning conditions as an IF statement.

Activity 1.14

1. Name the three types of control structures.
2. Another term for condition is what?
3. Name the two types of selection.
4. What does the term *mutually exclusive conditions* mean?
5. Give an example of a Boolean operator.
6. If the terms AND and OR are included in a single complex condition, which of these operators will be performed first?
7. How can the order in which operations in a complex condition be changed?

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 that we want to perform over and over again are known as the **loop body**.

Activity 1.15

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 two parts to this statement. 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:

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

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

Now we can rewrite our original algorithm as:

```
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 1.16

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

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.

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
```

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 1.17

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
```

Activity 1.18

Can you see a practical problem with the algorithm?

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

Activity 1.19

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 1.20

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
```

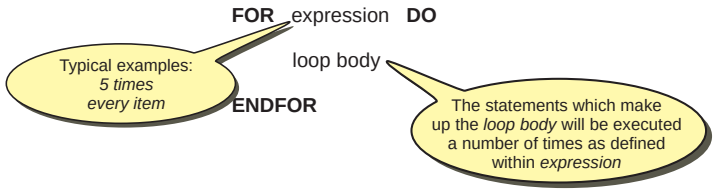
and

```
IF card is a knight THEN
```

The general form of the FOR statement is shown in FIG-1.11.

FIG-1.11

The FOR 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 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
```

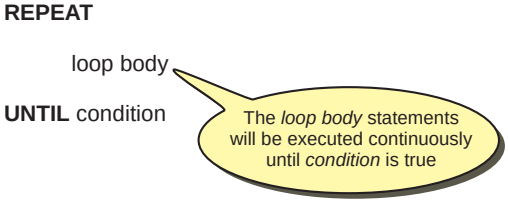
We could also use the REPEAT..UNTIL loop to describe how a slot machine (one-armed bandit) is played:

```
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-1.12.

FIG-1.12

The REPEAT Loop



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

Activity 1.21

A one-armed bandit costs 50p per play. A player has several 50p pieces and is determined to play until his coins are gone or until he wins at least £10.00. Write an algorithm describing the steps in this game. The algorithm should make use of the following statements:

```
Collect winnings
Place coin in machine
Pull arm
UNTIL all coins are gone OR winnings are at least £10.00
```

There is still a problem with our number-guessing game. 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 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 this solution implies that the dealer must take at least one card before deciding to stop. Using the WHILE..ENDWHILE structure we could describe the logic as

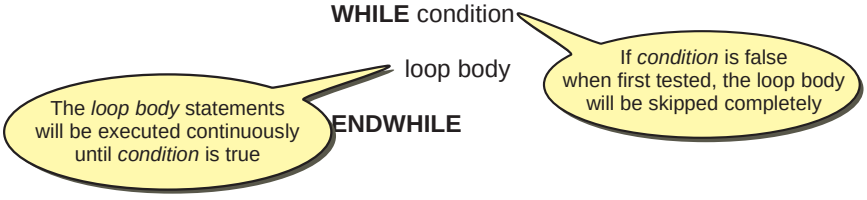
```
Calculate sum of the initial two cards
WHILE sum is less than 16 DO
      Take another card
      Add new card's value to sum
ENDWHILE
```

Now determining if the sum is less than 16 is performed before *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-1.13.

FIG-1.13

The WHILE 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, whereas the loop body of a REPEAT structure will always be executed at least once.

Activity 1.22

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

```
Roll both dice
```

and

```
Choose dice with lower value
```

Activity 1.23

1. What is the meaning of the term iteration?
2. Name the three types of looping structures.
3. What type of loop structure should be used when looping needs to occur an exact number of times?
4. What type of loop structure can bypass its loop body without ever executing it?
5. What type of loop contains an exit condition?

Data

Almost every game requires the players to remember or record some facts and figures. In our number guessing game described earlier, the players needed to remember the original number and the guesses made; in Hangman the word being guessed and the letters guessed so far must be remembered.

These examples introduce the need to process facts and figures (known as **data**). Every computer game has to process data. This data may be the name of a character, the speed of a missile, the strength of a blow, or some other factor.

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 character's *title*, *strength* and *charisma* are names of data items; "*Fred the Invincible*", *3*, and *9* are examples of the actual values which might be given to these data items.

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 *lives remaining*, will be reduced if the player's character is killed.

Activity 1.24

List the names of four data items that might be held about a player in a game of Monopoly.

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**.

Activity 1.25

Identify input, calculation, comparison and output operations when playing Hangman
For example, the algorithm needs to compare the letter guessed by the player with the letters in the word.

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 (-). A summary of the operators available are given in TABLE-1.5.

TABLE-1.5

Mathematical 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 TABLE-1.6.

TABLE-1.6

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 := 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 := 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 1.26

1. What are the two main characteristics of any data item?

2. When data is input, from where is its value obtained?

3. 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 programme on a video recorder as:

```
Put new tape in video
Set timer details
```

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

```
Press the eject button
IF there is a tape in the machine THEN
    Remove it
ENDIF
Place the new tape in the machine
```

and the second statement could be substituted by:

```
Switch to timer mode
Enter start time
Enter finish time
Select channel
```

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 tasks 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 Place lid back 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, the statement `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 need to be given more detail in a LEVEL 3 breakdown.

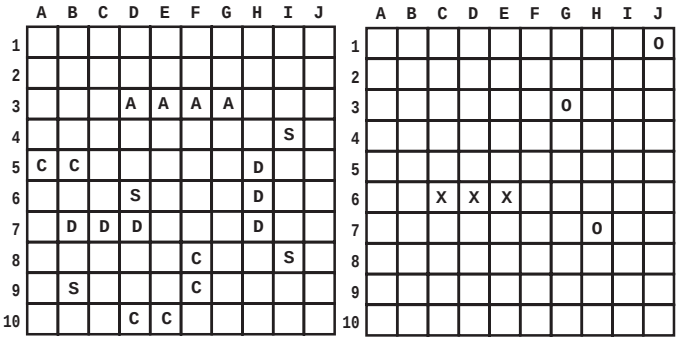
Activity 1.27

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 squares in the first grid to mark 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).



Vessels are positioned in the left-hand grid

Results of guesses are placed in the right-hand grid

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

continued on next page

Activity 1.27 (continued)

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 TABLE-1.7.

TABLE-1.7

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"

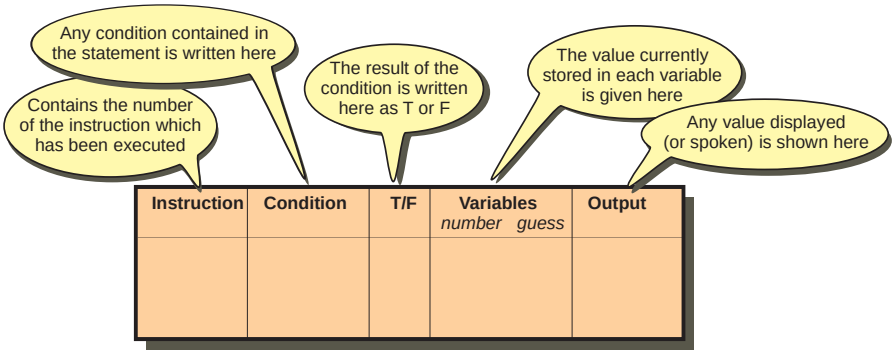
Once we've created test data, we need to work our way through the algorithm using that test data and checking that we get the expected results. The algorithm for the number game is shown below, this time with instruction numbers added.

```
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-1.14.

FIG-1.14

The Components of a Trace Table



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-1.15.

FIG-1.15

Tracing the First Statement

Instruction	Condition	T/F	Variables *number guess*	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-1.16.

FIG-1.16

Moving through the Trace

Instruction	Condition	T/F	Variables *number guess*	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-1.17.

FIG-1.17

Tracing a Condition

Instruction	Condition	T/F	Variables *number*	Variables *guess*	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-1.18.

FIG-1.18

Tracing a Second Condition

Instruction	Condition	T/F	Variables *number*	Variables *guess*	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-1.19.

FIG-1.19

Recording Output

Instruction	Condition	T/F	Variables *number*	Variables *guess*	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-1.20.

FIG-1.20

Reaching the end of the REPEAT .. UNTIL Structure

Instruction	Condition	T/F	Variables *number*	Variables *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			

Since statement 13 contains a condition which is false, we return to statement 2 and then onto 3 where we enter 15 as our second guess (see FIG-1.21).

FIG-1.21

Showing Iteration

Instruction	Condition	T/F	Variables *number*	Variables *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	

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

Activity 1.28

Create your own trace table for the number-guessing game and, using the same test data as given in TABLE-1.7 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	*
Division	/
Addition	+
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

1. An algorithm
2. A Computer program
3. 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 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 you are first to place your hand over
those cards AND the last two cards laid
down are of the same value
THEN
  You win the cards already played
ENDIF
```

Activity 1.11

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

Activity 1.12

Assuming the player has one Ace and one Knave the statement

```
IF a player has an Ace AND player has
King OR player has two Knaves
THEN
```

would reduce to

```
  IF true AND false OR false THEN
```

The AND operation is then performed giving:

```
  IF  false OR false THEN
```

Next, the OR operation is completed giving a final value of

```
  IF false THEN
```

and, therefore the player does not pick up an extra card.

Activity 1.13

```
IF (total of cards held is 43 OR hand has
4 cards of the same value ) AND hand
contains a Queen THEN
```

Activity 1.14

1. Sequence
   Selection
   Iteration

2. Boolean expression

3. Binary selection
   Multi-way selection

4. No more than one of the conditions can be true at any given time.

5. Boolean operators are: AND, OR, and NOT.

6. AND is performed before OR .

7. The order in which operations in a complex condition are calculated can changed by using parentheses.

Activity 1.15

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

Activity 1.16

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 1.17

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 1.18

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.

Activity 1.19

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

Activity 1.20

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

Activity 1.21

```
REPEAT
  Place coin in machine
  Pull arm
  IF a win THEN
      Collect winnings
  ENDIF
UNTIL all coins are gone OR winnings are
at least £10.00
```

Activity 1.22

```
Roll both dice
WHILE both dice do not match in value DO
  Choose dice with lower value
  Roll the chosen dice
ENDWHILE
```

Activity 1.23

1. Iteration means executing a set of instructions over and over again.

2. The three looping structures are:

   ```
   FOR .. ENDFOR
   REPEAT .. UNTIL
   WHILE .. ENDWHILE
   ```

3. The FOR .. ENDFOR structure.

4. The WHILE .. ENDWHILE structure.

5. The REPEAT .. UNTIL structure.

Activity 1.24

Number of properties held
Amount of money held
The playing token being used
The position on the board

Activity 1.25

Input:
- Letter guessed
- Word guessed

Calculations:
- Where to place a correctly guessed letter
- The number of wrong guesses made

Comparisons:
- The letter guessed with the letters in the word
- The word guessed with the word to be guessed
- The number of wrong guesses with the value 6 (6 wrong guesses completes the drawing of the hanged man)

Output:
- Hyphens indicating each letter in the word
- Gallows
- Body parts of the hanged man
- Correctly guessed letters

Activity 1.26

1. Name and value
2. From outside the system. In a computerised system this is often via a keyboard.
3. The relational operators are:
   <, <=, >, >=, =, and <>

Activity 1.27

The LEVEL 1 is coded as:

```
1. Draw grids
2. Add ships to left grid
3. REPEAT
4.      Call grid position(s)
5.      Respond to other player's call(s)
6. UNTIL there is a winner
```

The expansion of statement 4 would become:

```
4.1 Call grid reference
4.2 Get reply
4.3 WHILE reply is HIT DO
4.4     Mark position in second grid with X
4.5     Call grid reference
4.6      Get reply
4.7 ENDWHILE
4.8 Mark position in second grid with 0
```

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 1.28

Instruction	Condition	T/F	Variables *number*	Variables *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
11					
12					
13	guess = number	T			

The expected results were obtained.

2

Starting DarkBASIC Pro

Programming a Computer

Introduction

In the last chapter we created algorithms written in a style of English 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.

There are many programming languages; C++, Java, C#, and Visual Basic being amongst the most widely used. So how do we choose which programming language to use? Probably the most important consideration is the area of programming that is best suited to a given language. For example, Java is designed to create programs that can be executed on a variety of different computers, while C++ was designed for fast execution times.

We are going to use a language known as DarkBASIC Professional or just DarkBASIC Pro, which was designed specifically for writing computer games. Because of this, it has many unique commands for displaying graphics, controlling joysticks, and creating three dimensional images.

The Compilation Process

As we will soon see, DarkBASIC Pro uses statements that retain some English terms and phrases, so we can look at the set of instructions and make some sense of what is happening after only a relatively small amount of training.

Binary is a method of representing numbers using only the digits 0 and 1.

Unfortunately, the computer itself only understands instructions given in a **binary code** known as **machine code** and has no capability of directly following a set of instructions written in DarkBASIC Pro. But this need not be a problem. If we were given a set of instructions written in Russian we could easily have them translated into English and then carry out the translated commands.

This is exactly the approach the computer uses. We begin the process of creating a new piece of software by mentally converting our structured English into DarkBASIC Pro commands. These commands 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 into machine code. The original code is known as the **source code**; the machine code equivalent is known as the **object code**.

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 DarkBASIC Pro the incorrect command is highlighted in red.

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.

As we work on the computer entering a DarkBASIC Pro program, we need to save this source code to a file. This ensures that we have a copy of our work should there be a power cut or we accidentally delete the program from the computer's memory. DarkBASIC Pro refers to this as the **source file**.

But a second file, known as the **project file** is also produced. This second file is created automatically by DarkBASIC Pro and contains details of any images, sounds or other resources that might be used by your program.

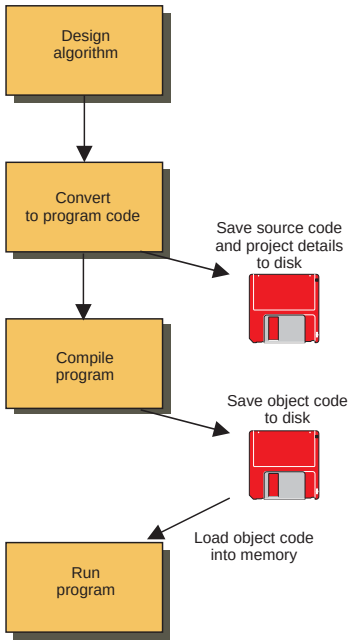
When we compile our program (translating it from source code to object code), yet another file is produced. This third file, the **executable file**, contains the object code and is, again, created automatically.

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

Creating Software



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

Activity 2.1

1. What type of instructions are understood by a computer?
2. What piece of software is used to translate a program from source code to object code?
3. Misspelling a word in your program is an example of what type of error?

Starting DarkBASIC Pro

Introduction

DarkBASIC Pro is based on one of the earliest computer languages, BASIC, but has been enhanced specifically to aid the creation of games programs.

The language was invented by Lee Bamber who formed a company to sell DarkBASIC Pro. Over the last few years the company has grown in size and expanded to sell other DarkBASIC related products, such as DarkMatter, which contains many 3D objects that can be used in DarkBASIC programs.

In fact, there are two versions of the language: DarkBASIC and DarkBASIC Professional. It's this second, enhanced version of the language we will be using here.

DarkBASIC Pro Files

Because a typical program written in DarkBASIC Pro is likely to contain images, sounds and even video, the DarkBASIC Pro package has to save much more than the set of instructions that make up your program; it also needs to store details of these images, sounds, etc.

To do this DarkBASIC Pro creates two files every time you produce a new program (see FIG-2.2).

The first of these files, known as the **project file**, contains details of the images and sounds used by your program, as well as other information such as the screen resolution and number of colours used. This file has a *.dbpro* extension.

The second file, known as the **source file**, contains only the program's code written in the DarkBASIC Pro language. This file has a *.dba* extension.

FIG-2.2

The Two Files Created by a DarkBASIC Pro Program

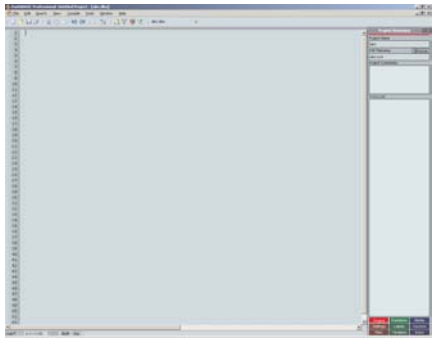
Getting Started with DarkBASIC Pro

When you first start up DarkBASIC Pro you should see one of the screens shown in FIG-2.3. Exactly which one you see depends on how often DarkBASIC Pro has been run on your computer. The first time the program is run, the display will match that shown on the left of FIG-2.3; every other time your screen will match that shown on the right.

FIG-2.3

The Start-Up Screen in DarkBASIC Pro

DarkBASIC Pro Start-Up Screen (First Start-Up Only)



DarkBASIC Pro Start-Up Screen (Subsequent Start-Ups)

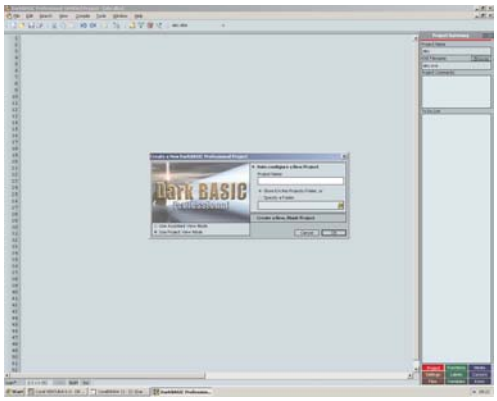
First Start-Up

If this is the first time DarkBASIC Pro has been run on your machine, as well as the main window, the **Assistant Window** also shows on the right-hand side.

If you close down the Assistant Window the display changes to match that shown in FIG-2.4, showing the Project Dialog box.

FIG-2.4

The Project Dialog Box



Subsequent Start-Ups

When DarkBASIC Pro is started up for the second (or subsequent) time, use the FILE | NEW PROJECT option from the main menu, or click on the *New Project* icon near the top left corner, to display the *Project Dialog* box.

Specifying a Project

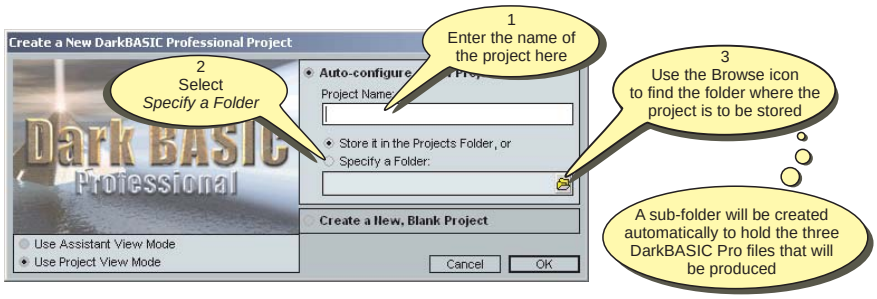
The next stage is to create a project file by filling in the details required by the Project Dialog box.

First the name to be given to the project is entered. This should be something meaningful like *Hangman* or *SpaceMonsters*.

Next the *Specify a Folder* radio button is selected and the folder in which the DarkBASIC Pro projects are to be saved is entered. The folder specified must already exist. See FIG-2.5 for a summary of these steps.

FIG-2.5

Filling in the *New Project* Dialog Box



Once the OK button in the Project Dialog box is clicked, the dialog box disappears and you are left with the main edit area where the program code is entered. Line numbers appear to the left of this area.

A First Program

Before we begin looking in detail at the commands available in DarkBASIC Pro, we'll have a quick look at a simple program and show you how to type it in, run it and save the code.

The program in LISTING-2.1 gets you to enter your name at the keyboard and then displays a greeting on the screen.

LISTING-2.1

A First Program

```
Rem Project: First
Rem Created: 02/10/2004 07:35:27
Rem ***** Main Source File *****

REM *** A program to read and display your name ***
INPUT "Enter your name : ",name$
PRINT "Hello " ,name$, " welcome to DarkBASIC Pro."
WAIT KEY
END
```

An Explanation of the Code

DarkBASIC Pro allows words to be given in either upper or lower case.

REM

This is short for REMARK and is used to indicate a comment within the program. Comments are totally ignored when the source code is translated into object code and are only included for the benefit of anybody examining the program code, giving an explanation of what the program does.

When you type in a program, you'll see that the instructions are colour-coded with keywords appearing in blue.

INPUT

This is a keyword in DarkBASIC Pro. Keywords are words recognised by the programming language as having a specific meaning.

All keywords are shown throughout this text in uppercase, but lowercase characters are also acceptable.

The INPUT keyword tells the computer to allow the user to enter a value from the keyboard.

The need for a space after the colon will become clear when you run this program.

"Enter your name:"	This message is displayed on the screen as a prompt, telling the user what information is to be entered. Messages are always enclosed in double quotes (" ") and are more generally known as **strings**.
name$	This is the variable in which the value entered by the user will be stored.
PRINT	This command is used to tell the computer to display information on the screen.
"Hello "	This is the first piece of information to be displayed
,	Items of data are separated from each other by commas.
name$	The value held in the variable *name$* is to be displayed. This will be whatever value the user typed in when the earlier INPUT statement was executed.
" welcome to DarkBASIC Pro."	Another data item to be displayed.
WAIT KEY	This command contains two key words which tell the computer to wait for a key to be pressed before continuing to the next instruction.
END	Marks the end of the program.

Activity 2.2

In this Activity you are going to type in and run the program given in LISTING-2.1.

Create a folder in the C: drive (or elsewhere) named *DarkBasicProjects*

Start up DarkBASIC Pro.

Bring up the Project Dialog box shown in FIG-2.4.

Name the project *first.dbpro*, select *Specify a Folder*; browse to your *DarkBASICProjects* folder and click OK.

The first three lines of the program will appear automatically (only the date and time will differ from that in LISTING-2.1).

Type in the remainder of the program as shown in LISTING-2.1.

Execute the program by pressing the F5 key or clicking on the Run icon.

When requested, type in your name. You should then see a message including your name displayed on the screen.

Finally, press any key to finish the program and return to the editor.

Activity 2.3

1. PRINT is what type of word?

2. Messages enclosed in quotes are known as what?

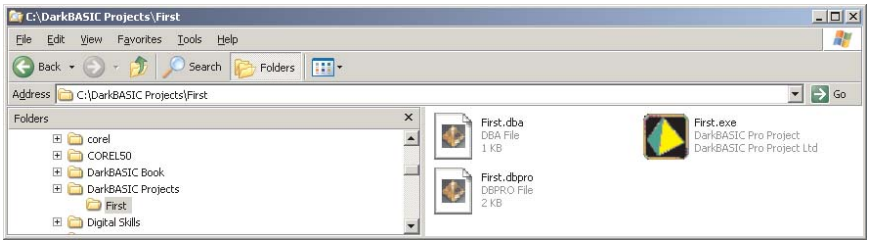
3. The WAIT KEY instruction causes what to happen?

If we use Windows Explorer to examine our *DarkBasicProjects* folder we'll see that a new sub-folder called *first* has been created.

Inside that new folder are three files (see FIG-2.6).

FIG-2.6

Files Created by DarkBASIC Pro



first.dbpro	This is the project file.
first.dba	This is the file containing the source code.
first.exe	This is the machine code version of your program. It's the code in this file that is actually executed when you run your program.

If you ever want to give away your completed programs to other people, you only need to give them a copy of the .exe file. This contains everything they need to run your program without allowing them to see your original DarkBASIC Pro code.

When opening the file in Notepad, change the *File of Type* entry to *All Files.*

Activity 2.4

Without closing down DarkBASIC Pro, load up Notepad (it's in Accessories) and open up the file *first.dba*.

Notice that the file only contains the three REM statements which were generated automatically when you opened your new project. None of the lines you typed in are present.

Saving Your Project

When you've typed in your program you need to save both the project and the source files. To do this, select FILE|SAVE PROJECT and then FILE|SAVE SOURCE.

Activity 2.5

Save your project and source files as described above.

First Statements in DarkBASIC Pro

Introduction

Learning to program in DarkBASIC Pro is very simple compared to other languages such as C++ or Java. Unlike most other programming languages, it has no rigid structure that must be adhered to. In fact, there are only two statements that you should include at this stage. These are given below.

Ending a Program

The END Statement

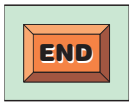
The first statement we examine is the one that should come at the end of any program you write. It consists of the single keyword END and, as you might have guessed, marks the end of your program.

We have already seen this statement in LISTING-2.1.

Some of the statements available in DarkBASIC Pro have quite a complex syntax so, to help show exactly what options are available when using a statement, we'll use informal syntax diagrams. FIG-2.7 shows a syntax diagram for the END statement.

FIG-2.7

The END Statement



These diagrams contain one or more tiles. A raised tile (like the one above) signifies a DarkBASIC Pro keyword. The order of the tiles signifies the order in which the keywords must be placed when using this statement in your program.

So the diagram above tells us that the END statement contains only the single word END.

The WAIT KEY Statement

We can make a program pause until a key is pressed using the WAIT KEY statement. The program will only continue after a key has been pressed. Any key on the keyboard will do.

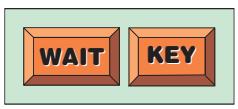
For example, in the program given in LISTING-2.1, the computer will pause after the PRINT statement is executed.

For most simple programs, you need to include a WAIT KEY statement immediately before the END statement, otherwise your program will finish and close down before you get a chance to view what is being displayed on the screen.

The syntax for this statement is shown in FIG-2.8.

FIG-2.8

The WAIT KEY Statement



Adding Comments

It is important that you add comments to any programs you write. These comments should explain 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.

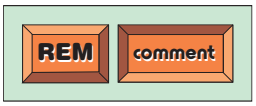
Comments are totally ignored by the translation process as it turns DarkBASIC Pro statements into machine code. The purpose of comments is to make a program more readable to other people who may have to modify a program after you've moved on to other things.

In DarkBASIC Pro there are three ways to add comments:

- Add the keyword REM. The remainder of the line becomes a comment (see FIG-2.9).

FIG-2.9

The REM Comment



Notice that this syntax diagram introduces the sunken tile. Sunken tiles signify details that are determined by the programmer. Hence, the programmer gets to choose exactly what comment should be added after the keyword REM. For example:

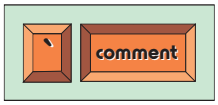
Adding asterisks to a comment helps it to stand out.

```
REM *** Program to display numbers ***
```

- Add an opening quote 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-2.10).

FIG-2.10

The ‘ Comment



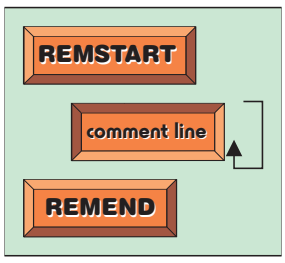
For example:

```
‘ Get details from keyboard
```

- 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-2.11).

FIG-2.11

The REMSTART .. REMEND Comment



This 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 comment lines:

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

Activity 2.6

1. How are keywords shown in a syntax diagram?

2. What does a sunken tile in a syntax diagram represent?

3. How is a repeatable element in a statement represented in a syntax diagram?

Outputting to the Screen

Introduction

Even the simplest program will require information to be displayed on the screen.

In DarkBASIC Pro the simplest way to display information on the screen is to use the PRINT statement. Other statements exist which allow changes to the colour, font and style of displayed characters to be specified.

A description of most of these statements are given over the next few pages

The PRINT Statement

As we saw in LISTING-2.1, information can be displayed using the PRINT statement.

To use it, we start with the keyword PRINT, followed by whatever information we want to display. For example, the statement

```
PRINT "Hello"
```

displays the word *Hello* on the screen. The quotes themselves are not displayed. Absolutely any set of characters can appear between the quotes, including spaces.

Although a set of characters, or strings, must be enclosed in double quotes, if you want to display a number, quotes are not required. For example, the following are valid statements:

```
PRINT 12
PRINT 3.1416
PRINT -7.0
```

It is possible to display several pieces of information using a single PRINT statement by separating each value to be displayed by a comma:

```
PRINT 12,7,1.2
```

Unfortunately, all the values in this statement will be displayed without any spaces between them giving the impression of one large number (1271.2) rather than three separate values.

To solve this problem we need to display some spaces between the numbers:

```
PRINT 12,"   ",7,"   ",1.2
```

Spaces are just strings - like any other sequence of characters - and must be enclosed in double quotes.

When several values are displayed by a single PRINT statement they appear on a single line of the screen, but by using several PRINT statements we can make the data appear over several lines:

```
PRINT 12
PRINT 1
PRINT 1.2
```

To turn this into a complete program we just need to add the WAIT KEY and END statements as shown in LISTING-2.2.

LISTING-2.2

Displaying Numbers

REM statements generated when you start a new project have been omitted from the listing.

```
REM *** Print some numbers ***
PRINT 12
PRINT 7
PRINT 1.2

REM *** End program ***
WAIT KEY
END
```

Activity 2.7

Start up a new DarkBASIC Pro project.

To do this select File | New Project.

In the *Project* dialog box that appears, call the project *printing.dbpro*; select *Specify a Folder*; browse to your *DarkBASICProjects* folder and click OK.

Type in and test the program given in LISTING-2.2.

Remember to save the Source and Project files when you have finished.

Creating Blank Lines

The PRINT statement can even be used without any data values being given, as in the line

```
PRINT
```

This has the effect of creating a blank line on the screen. Hence, the lines

```
PRINT 1
PRINT
PRINT 2
```

would display the values 1 and 2 with a blank line between them.

Activity 2.8

Modify your last program so that a blank line appears between each number displayed.

Ending the PRINT Statement with a Semicolon

If you end a PRINT statement with a string and a semicolon, the output produced by the next PRINT statement will be displayed on the same line. For example, the lines:

```
PRINT 12,"";
PRINT 7
PRINT 1.2
```

would produce the output

```
127
1.2
```

As you will see later, this apparently useless option can be used to great effect.

Activity 2.9

Create a new project called *Printing2*.

Write a program which displays the numbers 1, 2 and 3 on the same line. There should be a small gap between each number.

Change your program so that the numbers 1, 2 and 3 are displayed on separate lines.

Modify the code again so that the program pauses before each number is shown. (HINT: You'll need to add a WAIT KEY statement after each PRINT statement.)

Activity 2.10

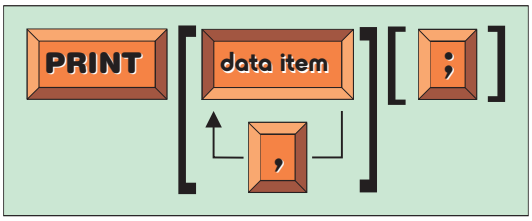
Write a program (call the project *Shapes*) to display the following three shapes (pause the program between each):

```
a)    **********
      **********
      **********
b)    *
      **
      ***
      ****
      *****
c)         *
          **
         ***
        ****
       *****
```

The format of the PRINT statement is shown in FIG-2.12.

FIG-2.12

The PRINT Statement



This diagram introduces two new concepts. Items within the brackets are optional and may be omitted. Any number of data items can be displayed, but each must be separated from the next by a comma.

Activity 2.11

Using the information given in the PRINT statement's syntax diagram, which of the following PRINT statements are invalid?

a) `PRINT`
b) `PRINT "Start game"`
c) `PRINT 7;`
d) `PRINT "";`
e) `PRINT 6,5,4;`

Positioning Text on the Screen

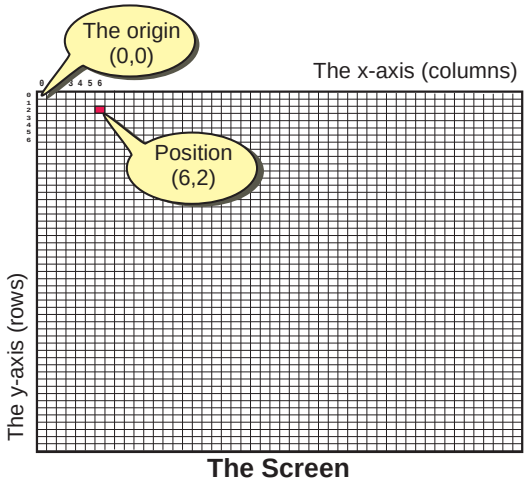
In DarkBASIC Pro the screen is treated like a piece of paper divided into thousands of small squares, as shown in FIG-2.13. These small invisible squares are known as **pixels** (derived from the phrase **picture elements**). An individual pixel is identified by giving its position on the screen.

A pixel's position is given by the column number (also known as the position on the x-axis) followed by the row number (the position on the y-axis) separated by a comma.

The top left pixel is at position (0,0). This point is known as the **origin**.

FIG-2.13

The Screen is Made Up of Pixels



Exactly how many pixels are on the screen depends on the screen resolution (which we will examine later) but there will be at least 640 columns by 480 rows.

The SET CURSOR Statement

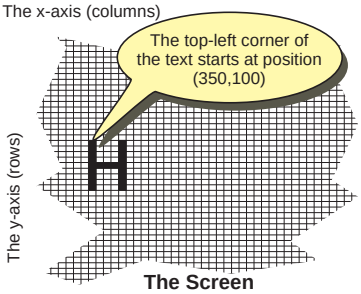
Normally, the first text that we output to the screen will start at the origin, but we can change this by using the SET CURSOR statement which allows us to specify where on the screen the next PRINT statement will begin its output. For example, the statements

```
SET CURSOR 350, 100
PRINT "HELLO"
```

displays the word HELLO, with the top-left comer of the H starting at position (350,100) (i.e. at column 350, row 100) as shown in FIG-2.14.

FIG-2.14

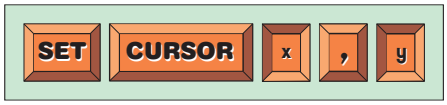
Positioning Text Using the SET CURSOR Statement



The format for the SET CURSOR statement is shown in FIG-2.15.

FIG-2.15

The SET CURSOR Statement



In the diagram above:

x,y	is a pair of integer values specifying the position to which the cursor is to be moved.

Activity 2.12

Create a new project (*corners.dbpro*) that displays the letters A, B, C and D so that one letter appears at each corner of the screen.

(You'll have to use trial and error to find the correct positions)

Activity 2.13

Since we can output at any position on the screen, this allows us to display different values at the same position on the screen.

Create a new project (*overwrite.dbpro*) containing the following code:

```
REM *** Output two strings at the same location ***
SET CURSOR 100,100
PRINT "Hello"
WAIT KEY
SET CURSOR 100,100
PRINT "Goodbye"
REM *** End program ***
WAIT KEY
END
```

Check the output produced by running this program.

The TEXT Statement

The effects of the SET CURSOR and PRINT statements are combined in the TEXT command which takes both the value to be displayed and the position at which the data is to be displayed. For example, the statement

```
TEXT 350, 100, "HELLO"
```

has the same effect as the SET CURSOR example given earlier, although you may find that the program uses a different screen resolution when the output is displayed.

Your screen will almost certainly use a different resolution when using the TEXT statement than it did in previous programs. This means that in this Activity you'll have to change the coordinates from those used in the previous example.

Activity 2.14

Change your *corners.dbpro* project so that it uses the TEXT command to position the letters in the corners of the screen.

There are a few differences between the PRINT and TEXT commands.

Firstly, TEXT makes use of a graphics display mode to create output, PRINT does not. Because of this, the screen resolution in Activity 2.14 may differ from that used by the PRINT statement and how output is handled will change.

Activity 2.15

Change your *overwrite.dbpro* project replacing the SET CURSOR and PRINT commands with equivalent TEXT statements.

How does the result differ from before?

The second difference is that the TEXT command will only display strings, so a line such as

```
TEXT 100, 100, 12
```

where the statement attempts to display the value 12 is not acceptable and will cause an error message to appear when you attempt to run the program. Of course, by enclosing the 12 in quotes you turn it from a number into a string and this would be accepted:

```
TEXT 100, 100, "12"
```

A final difference is that the TEXT command can only be used to display a single value at a time. Hence, a statement such as

```
TEXT 100, 100, "Hello", "again"
```

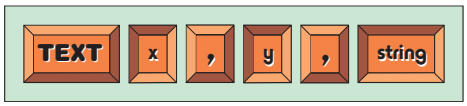
would fail since there are two strings in the command. Again, this could be corrected, this time by joining the two strings:

```
TEXT 100, 100, "Hello again"
```

The syntax for the TEXT statement is given in FIG-2.16.

FIG-2.16

The TEXT Statement



In the diagram above:

x,y	is a pair of integer values specifying the position to which the cursor is to be moved.
string	is the string value to be displayed on the screen. All strings should be enclosed in double quotes.

The CENTER TEXT Command

Like most programming languages, DarkBASIC Pro keywords use American spelling. Hence, CENTER and not CENTRE.

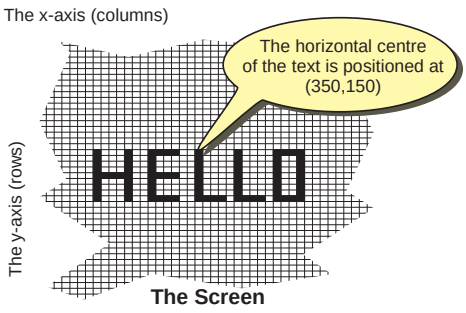
Whereas the TEXT command starts output at the specified position, CENTER TEXT, which uses the same format as TEXT, centres the output horizontally round the value given for the x-axis. Hence, the statement

```
CENTER TEXT 350, 150, "Hello"
```

will display the word *Hello* as shown in FIG-2.17.

FIG-2.17

Positioning Text Using CENTER TEXT



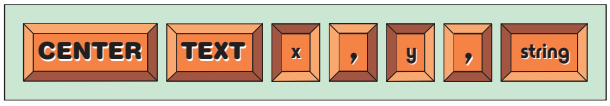
Activity 2.16

Write a program (project *centre.dbpro*) to place the word *MIDDLE* at the centre of the screen.

The format of the CENTER TEXT statement is given in FIG-2.18.

FIG-2.18

The CENTER TEXT Statement



In the diagram above:

x,y	is a pair of integer values specifying the position where the horizontal centre of the string is to be output.
string	is the string value to be displayed on the screen.

Changing the Output Font

When you display text on your computer, you can choose the size, style, and font of that text.

We can change the font style and size used when outputting text by using the SET TEXT FONT and SET TEXT SIZE commands. Once a new font and size has been set, any subsequent output statements will be done in this style.

The SET TEXT FONT Statement

You have to add a font name in quotes to the end of this statement. Any values output after this will be shown in that font. For example,

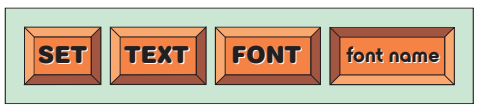
```
SET TEXT FONT "Courier New"
```

will result in the Courier New font being used by any subsequent output.

The format for this instruction is given in FIG-2.19.

FIG-2.19

The SET TEXT FONT Statement



In the diagram above:

font name	is a string (enclosed in quotes) giving the name of the font to be used for subsequent output.

The SET TEXT SIZE Statement

The text size is given in **points** (a point being 1/72 of an inch). For example,

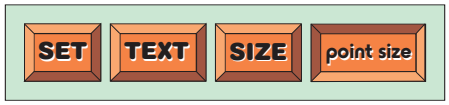
```
SET TEXT SIZE 20
```

will result in subsequent output using characters that are 20/72 of an inch tall.

The format of this statement is given in FIG-2.20.

FIG-2.20

The SET TEXT SIZE Statement



In the diagram above:

point size	is an integer value specifying the size of font (in points) to be used for subsequent output.

The SET TEXT TO Statement

You can also set the text style to produce italics, bold, or bold italics output as well as the normal default style. This is achieved using the SET TEXT TO commands. There are four options:

```
SET TEXT TO BOLD
SET TEXT TO ITALIC
SET TEXT TO BOLDITALIC
SET TEXT TO NORMAL
```

The following program (LISTING-2.3) outputs the word *HELLO* in large, bold, Courier New font:

LISTING-2.3

Setting Text Size, Font and Style

```
REM *** Use Courier New size 20 bold ***
SET TEXT FONT "Courier New"
SET TEXT SIZE 20
SET TEXT TO BOLD
PRINT "HELLO"
WAIT KEY

REM *** Change to italics ***
SET TEXT TO ITALIC
PRINT "HELLO"
WAIT KEY

REM *** Change to bold italics ***
SET TEXT TO BOLDITALIC
PRINT "HELLO"
WAIT KEY

REM *** Change to normal ***
SET TEXT TO NORMAL
PRINT "HELLO"
REM *** End the program ***
WAIT KEY
END
```

Activity 2.17

Type in and test the program given in LISTING-2.3. Name the project *fonts.dbpro*.

Change the code so that all of the text is displayed in Times New Roman.

The format for this statement is shown in FIG-2.21.

FIG-2.21

The SET TEXT TO Statement

This diagram introduces another new feature. The braces are used to enclose items which are mutually exclusive alternatives. In other words, the statement is completed by choosing one of the options given in the braces.

Changing Colours

So far we've had white text on a black background, but you're free to choose any colours you want for both the text and the background. Before we see how to do that in DarkBASIC Pro, let's start with some basic facts about colour.

How Colours are Displayed

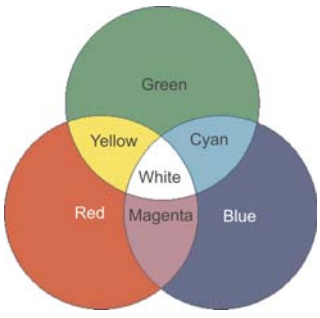
Have a close look at your computer monitor. It's in full colour, showing almost every colour and shade your eye is capable of seeing. And yet your screen can generate only three basic colours: red, green and blue.

Every other colour that you see on the screen is made up from those three colours. For example, to show the colour yellow, the screen combines the colours red and green; red, green and blue together produce white; when all three basic colours are switched off, we have black.

This is known as the **additive colour process** and the colours red, green and blue are known as the **primary colours**. The basic colours that can be constructed from these three primary colours are shown in FIG-2.22.

FIG-2.22

The Additive Colour Process



As you can see from the figure above, green and blue combine to give a colour

called cyan, while red and blue give magenta.

To create other colours and shades we need only to vary the brightness of the primary colours. Hence, to create orange we use an intense red, a less intense green, and no blue.

In computer systems the colour of any spot on the screen is recorded as a series of three numbers. These numbers represent the intensities of the red, green and blue (RGB) components (in that order) that make up the colour of the spot. Each number can range between 0 and 255; 0 means that the colour is not used, while 255 means that the colour is at full brightness. Hence, a bright yellow spot on the screen will be recorded as 255, 255, 0, meaning that the red and green are at full intensity, and the blue is switched off.

The RGB Statement

In DarkBASIC Pro we can define any colour using the RGB statement. This statement takes three values, enclosed in parentheses. These values define the intensities of the red, green and blue components that make up the required colour. The RGB statement combines these three components into a single integer value which it returns as a result of calling this statement. For example, the statement

```
PRINT RGB(255,255,0)
```

will display the integer value representing the colour yellow.

Activity 2.18

Create a new project (*colours.dbpro*) containing the following code:

```
PRINT RGB (255,255,0)
WAIT KEY
END
```

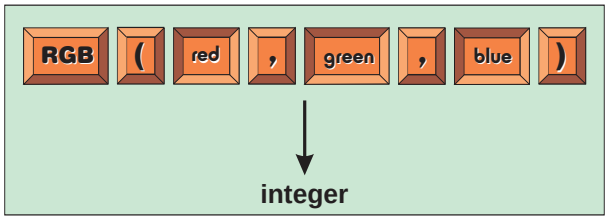
What value is displayed?

Change the values in the RGB command to 255,0,255. What value is displayed this time?

The syntax for the RGB statement is shown in FIG-2.23.

FIG-2.23

The RGB Statement



In the diagram:

red	is an integer value between 0 and 255
green	is an integer value between 0 and 255
blue	is an integer value between 0 and 255.

The arrowed line and the term *integer* signify that this statement returns an integer value.

How do you find out the red, green and blue values of some particularly nice shade of orange? Luckily, the DarkBASIC Pro editor can help. If you are busy typing in a program and suddenly need to supply the three values required by an RGB statement, you can simply right-click in the edit window. The resulting pop-up menu (see FIG-2.24) has an RGB Color Picker option which, when selected, displays a colour palette (see FIG-2.25).

FIG-2.24

The DarkBASIC Pro Editor's Pop-Up Menu

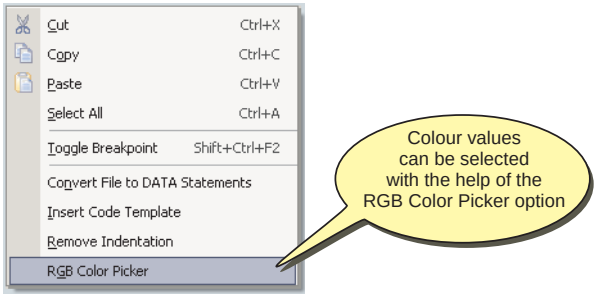
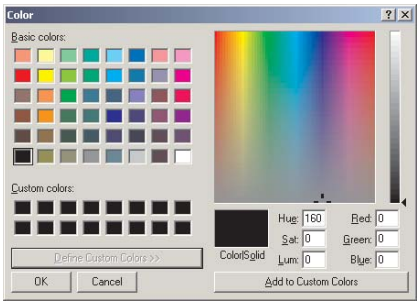


FIG-2.25

The Colour Palette Box



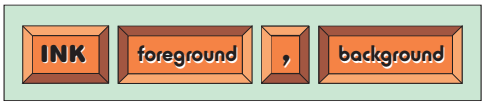
Selecting a colour from this palette and clicking OK automatically produces an RGB statement in your program code with the appropriate values to match the colour selected. We'll use this in the next Activity.

The INK Statement

In DarkBASIC Pro we can change both the colour used when writing text onto the screen (known as the foreground colour) and the colour behind that text (known as the background colour) using the INK. command. This command takes the general form shown in FIG-2.26.

FIG-2.26

The INK Statement



In the diagram:

foreground	is an integer value representing the colour to be used for the foreground.
background	is an integer value representing the colour to be used for the background.

The colour values themselves are created using the RGB command. So to have our text output in yellow on a red background we would use the command:

```
INK RGB(255,255,0), RGB(255,0,0)
```

Where you want to use black, rather than use RGB (0, 0, 0) you may simply enter the value zero. For example, to change the foreground to blue and the background to black, we would use the statement

```
INK RGB(0,0,255) ,0
```

Once you have set the ink colour, any output you do to the screen will be in that colour. For example, we would expect the program in LISTING-2.4 to display the word *HELLO* in yellow on a red background.

LISTING-2.4

Setting Foreground and Background Colours

```
REM *** Set yellow foreground and red background ***
INK RGB(255,255,0) , RGB(255,0,0)
PRINT "HELLO"

REM *** End program ***
WAIT KEY
END
```

Activity 2.19

Type in and execute the program in LISTING-2.4 (project *colours2.dbpro*).

What colour is the background on the screen?

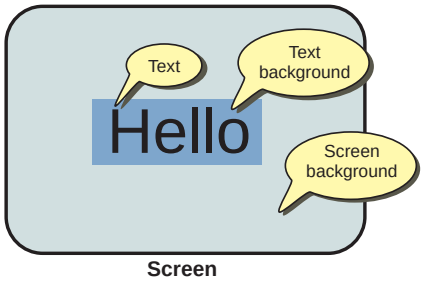
Delete the first RGB command within the INK statement and use the *RGB Color Picker* option to replace it with a colour of your choice.

Notice that the background colour in the INK command was set to red and yet the colour behind the letters is still black. If you want to know why, read on!

There are two main areas to any text that appears on the computer screen: the text and the text background (see FIG-2.27).

FIG-2.27

Text Areas



Screen

The foreground colour setting determines the colour of the text itself while the background colour sets the colour used in the text background. However, normally the text background is transparent so setting the background colour appears to have no effect. Usually, a transparent background will be exactly what we want, since it allows us to do things such as place text on top of an image, and have the image still show through the text (see FIG-2.28) but, as we'll see in a moment, we can change this transparent background setting.

Activity 2.20

Modify your previous program so that the word *GOODBYE* is displayed in green after the existing word *HELLO*.

FIG-2.28

Text with a Transparent Background

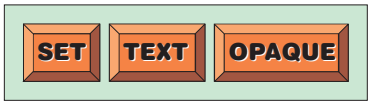
We'll see how to place images on the screen later.

The SET TEXT OPAQUE Statement

We can create a block of colour around any text we display by using the SET TEXT OPAQUE command. The colour used in the text background will be that defined as the background colour in your INK command. This statement has the format shown in FIG-2.29.

FIG-2.29

The SET TEXT OPAQUE Statement



For example, if a program contains the statements

```
SET TEXT OPAQUE
INK RGB(0,0,255), RGB(255,255,0)
PRINT "Hello"
```

the word *Hello* should appear in blue with a yellow background around the text.

Activity 2.21

Add the line SET TEXT OPAQUE to start of your previous program.

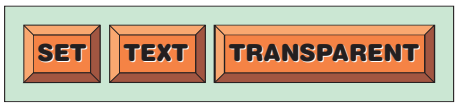
Change the program so that the word *GOODBYE* shows in cyan with a magenta background.

The SET TEXT TRANSPARENT Statement

Although text normally has a transparent background, if you use the SET TEXT OPAQUE command, every output statement executed later will have a coloured background. To return to a transparent background you need to use the statement SET TEXT TRANSPARENT which has the format shown in FIG-2.30.

FIG-2.30

The SET TEXT TRANSPARENT Statement



For example, if a program contains the statements

```
SET TEXT OPAQUE
INK RGB(0,0,255), RGB(255,255,0)
PRINT "Hello"
SET TEXT TRANSPARENT
PRINT "Goodbye"
```

Hello will have a yellow background while the word *Goodbye* would be surrounded by the black background of the screen.

The CLS Statement

Although when you first run your program it will start with a blank screen, you can clear everything from the screen at any point in your program by using the CLS statement (derived from **CL**ear **S**creen). To use the command, just write the term:

```
CLS
```

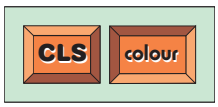
This gives a empty black screen. However, if you don't want the screen to be black, you can clear the screen to another colour by specifying a colour setting in conjunction with the CLS statement. For example, to create a green screen, use the line:

```
CLS RGB(0,255,0)
```

The format for this statement is shown in FIG-2.31.

FIG-2.31

The CLS Statement



In the diagram:

colour	is an integer value representing a colour. The screen will be filled with this colour after the CLS statement has been executed.

The program in LISTING-2.5 displays the word HELLO several times using both opaque and transparent modes. The screen colour is set to red.

LISTING-2.5

Using Transparent and Opaque Text

```
REM *** clear screen to red ***
CLS RGB(255,0,0)
REM *** Change text to yellow and the background to green ***
INK RGB(255,255,0), RGB(0,255,0)

REM *** Output the word HELLO with a transparent background ***
PRINT "HELLO"

REM *** Output the word HELLO twice with opaque background ***
SET TEXT OPAQUE
PRINT "HELLO"
PRINT "HELLO"

REM *** Return to transparent output ***
SET TEXT TRANSPARENT
PRINT "HELLO"

REM *** End the program ***
WAIT KEY
END
```

The output from this program is shown in FIG-2.32.

FIG-2.32

Changing Background Transparency

Activity 2.22

Type in the program in LISTING-2.5 (*backgrounds.dbpro*) and check out the results you obtain.

Activity 2.23

Create a new project (Box) which produces the following output.

```
**************
*     BOX      *
**************
```

Use Courier New, size 20, bold for the text.

The screen background should be red.

The asterisks should be yellow and the word *BOX* in blue with a black background.

Summary

- The CLS statement clears the screen using a given colour.
- The PRINT statement can be used to print any type of value.
- A single PRINT statement can display many values.
- The PRINT statement moves the cursor to a new line unless it finishes with a semicolon.
- The SET CURSOR statement moves the cursor to any position on the screen.
- The TEXT statement will output a single string at any position on the screen.
- The CENTER TEXT statement will output a string centred round a specified position.
- The INK statement sets the foreground and background colours used.
- The SET TEXT FONT statement sets the font to be used when displaying information.
- The SET TEXT SIZE statement sets the size to be used in text output.
- The text size is given in points (1/72 of an inch).
- The SET TEXT BOLD statement sets the text style to be used for output to bold.
- The SET TEXT BOLDITALIC statement sets the text style to be used for output to bold italics.
- The SET TEXT ITALIC statement sets the style to be used for output to italics.
- The SET TEXT NORMAL statement sets the style to be used for output to normal.

- The SET TEXT OPAQUE statement creates a background colour round any text that is output.
- The SET TEXT TRANSPARENT statement makes text background transparent.
- The WAIT KEY statement causes the program to halt until any key is pressed.
- The END statement marks the end of the program.

Some Display Techniques

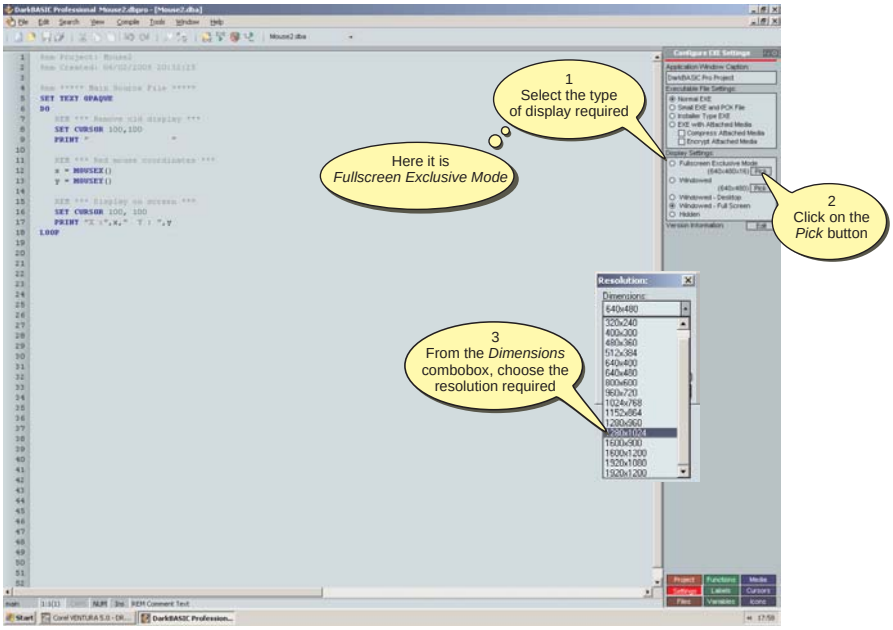
Screen Resolution

Earlier in this chapter you saw how the screen resolution changed when we started using the TEXT command in place of PRINT and SET CURSOR.

Luckily, we can choose which resolution we want the program's output to use by clicking on the brown *Settings* button at the bottom right of screen. In the resulting *Configure EXE Settings* window we can choose the resolution we want to use (see FIG-2.33).

FIG-2.33

Setting the Screen Resolution



You should consider the following when choosing a resolution.

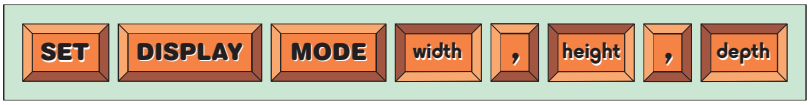
- Output looks better in higher screen resolutions.
- Everything looks smaller in higher resolutions.
- The memory on your video card may limit what resolutions can be used.
- High resolutions take longer to update, so a visually complex game may seem slower in higher resolutions.

The SET DISPLAY MODE Statement

It is also possible to set the screen resolution and colour depth from within your program using the SET DISPLAY MODE statement which has the format shown in FIG-2.34.

FIG-2.34

The SET DISPLAY MODE Statement



In the diagram:

width	is an integer value representing the width of the display mode required given in pixels.
height	is an integer value representing the height of the display mode required given in pixels.
depth	is an integer value representing the number of bits used to represent a single pixel on the screen. Typical values are 16, 24 or 32.

To set the screen to a resolution of 1280 by 1024 using 32 bit pixels we would use the line

```
SET DISPLAY MODE 1280,1024,32
```

It is only valid to chose a resolution which can be achieved by your video card and screen. Attempt to set an invalid resolution will produce an error message.

Choosing a Text Font

The SET TEXT FONT statement allows us to choose a font for any text that we intend to output. However, that choice depends on what fonts are available on your computer. You need to also consider what fonts are available on any other computers that your final software is to be run on. If your game makes use of a font such as *Kidnap* and that font is not available to someone who has bought your program, then the *Kidnap* font will be missing when your game runs on the buyer's machine.

Most fonts are **proportional fonts**. That is, the horizontal width of a character depends on what that character is. Hence, w's take up more width than i's. You can see this in the two lines below:

wwwwww
iiiiii

But some fonts are **mono-spaced**. In this style every character takes up the same width, as you can see below:

```
WWWWWW
iiiiii
```

When you're working in the DarkBASIC Pro editor entering the lines of your program, the text is displayed in a mono-spaced font, but the default font used by your program when outputting to the screen is a proportional font.

Erasing Text

Back in Activity 2.15, we saw that when the TEXT command is used to output more than one item to the same area of the screen it created an unreadable blob. We need some way of getting rid of the old text before outputting new text at the same position.

There are two ways to erase text from the screen.

The first of these is to overwrite the text with spaces with the text background set to opaque. This is demonstrated in LISTING-2.6.

LISTING-2.6

Erasing Text Using Opaque Spaces

```
REM *** Set background colour ***
CLS RGB (126,126,126)

REM *** Set text font, size, and background colour ***
SET TEXT FONT "Arial"
SET TEXT SIZE 36
INK RGB(255,0,0),RGB(126,126,126)

REM *** Output Text ***
TEXT 100,100,"Hello"
WAIT KEY

REM *** Remove text by writing opaque spaces ***
REM *** at the same position as the original text ***
SET TEXT OPAQUE
TEXT 100,100,"     "

REM *** End program ***
WAIT KEY
END
```

Activity 2.24

Type in and test the program given above (*TextGone*).

There should be 5 spaces between the quotes in the second TEXT statement.

What problem arises? Try to cure the problem.

A second method of erasing text is to overwrite with exactly the same text, but this time in the background colour. The logic of our strategy is:

Clear screen in required background colour
Set text font, size and colour
Output text
Set foreground colour to match background colour
Output text at same position as before

This logic is implemented in LISTING-2.7.

LISTING-2.7

Erasing Text Using The Same Text in the Background Colour

```
REM *** Set background colour ***
CLS RGB (126,126,126)

REM *** Set text font, size, and background colour ***
SET TEXT FONT "Arial"
SET TEXT SIZE 36
INK RGB(255,0,0),0

REM *** Output Text ***
TEXT 100,100,"Hello"
WAIT KEY

REM ** Remove text by writing it again in background colour ***
INK RGB(126,126,126),0
TEXT 100,100,"Hello"

REM *** End program ***
WAIT KEY
END
```

Activity 2.25

Modify your previous project to match the code given above.

Run the program and check that the text (*Hello*) is correctly erased.

Modify the TEXT statements in the program so that word *Goodbye* is erased from position 100,80.

Shadow Text

We can create shadowed text by writing the same text in different colours at slightly offset positions. This needs the following logic:

Set foreground colour to black
Output text
Set foreground colour to red (or some other colour)
Output text at a slightly different position from before

which is coded as:

```
REM *** Shadow Text ***
INK RGB(0,0,0),0
TEXT 102, 102, "Hello"
INK RGB(255,0,0) ,0
TEXT 100, 100, "Hello"
```

Activity 2.26

Add the code above to your existing program.
Try modifying the offset value of the black text and see what effect this has on the display.

Embossed Text

By creating two versions of a text, we achieved shadowed text; by creating three copies, we can produce an embossed effect.

To do this we need the following logic:

Clear the screen to grey (or some other colour)
Set foreground to black
Output required text
Set foreground to white
Output required text at an offset position
Set foreground to match background
Output required text at a position between the black and white output.

The code for this is:

```
CLS RGB(126,126,126)
REM *** Embossed Text ***
INK RGB(0,0,0),0
TEXT 201,201,"Goodbye"
INK RGB(255,255,255),0
TEXT 199,199,"Goodbye"
INK RGB(126,126,126),0
TEXT 200,200,"Goodbye"
```

Activity 2.27

Add the code above to your existing program.

Try modifying the font, size and colours used as well as the offset values to create the best effect.

Summary

- The screen resolution used by your program can be set manually using the Settings button.
- The screen resolution can be set from within your program using the SET DISPLAY MODE statement.
- In proportional fonts the width of a character depends on the shape of the character.
- In mono-spaced fonts all characters have the same width.
- Text can be erased from the screen by overwriting it with opaque spaces.
- Text can be removed from the screen by overwriting it with the same text in the background colour.
- Shadow text can be created by outputting a darker version of the text and then overwriting it with the same text slightly offset from the original and in a different colour.
- Embossed text can be created by outputting dark, light, and background coloured versions of the text. The dark version is written first, then the offset light text and finally the background coloured text at a mid point between the dark and light text.

Solutions

Activity 2.1

1. Machine code (or object code) instructions
2. Compiler
3. A syntax error

Activity 2.2

No solution required.

Activity 2.3

1. A keyword
2. Strings
3. Causes the program to pause until a key is pressed.

Activity 2.4

No solution required.

Activity 2.5

No solution required.

Activity 2.6

1. Keywords are shown in raised tiles

2. A sunken tile represent information whose exact value is determined by the programmer.

3. Repeatable elements are shown using a looping arrowed line.

Activity 2.7

No solution required.

Activity 2.8

The program code is:

```
REM *** Print some numbers ***
PRINT 12
PRINT
PRINT 7
PRINT
PRINT 1.2
REM *** End program ***
WAIT KEY
END
```

Activity 2.9

Version 1:

```
REM *** Display numbers on the same line ***
PRINT 1, " ", 2, " ", 3
REM *** End program ***
WAIT KEY
END
```

Version 2:

```
REM *** Display numbers on the separate
lines
PRINT 1
PRINT 2
PRINT 3
REM *** End program ***
WAIT KEY
END
```

Version 3:

```
REM *** Display numbers on the separate
lines ***
PRINT 1
WAIT KEY
PRINT 2
WAIT KEY
PRINT 3
REM *** End program ***
WAIT KEY
END
```

Activity 2.10

Program code:

```
REM *** Shape 1 ***
PRINT "**********"
PRINT "**********"
PRINT "**********"
PRINT "**********"
WAIT KEY
REM *** Shape 2 ***
PRINT "*"
PRINT "**"
PRINT "***"
PRINT "****"
PRINT "*****"
WAIT KEY
REM *** Shape 3 ***
PRINT "    *"
PRINT "   **"
PRINT "  ***"
PRINT " ****"
PRINT "*****"
REM *** End program ***
WAIT KEY
END
```

The last shape may not be exact. See Choosing a Text Font later in this chapter.

Activity 2.11

None of the PRINT statements are invalid

Activity 2.12

The exact values will vary according to your screen resolution.

The following code will fit a 1280 by 1024 screen

```
REM *** A top left ***
PRINT "A"
REM *** B top right ***
SET CURSOR 1260,0
PRINT "B"
```

```
REM *** C bottom left ***
SET CURSOR 0,990
PRINT “C”
REM *** D bottom right ***
SET CURSOR 1260,990
PRINT “D”
REM *** End program ***
WAIT KEY
END
```

Activity 2.13

The word *Goodbye* overwrites and removes the word *Hello* from the screen.

Activity 2.14

The code for a resolution of 1280 by 1024 is:

```
REM *** A top left ***
TEXT 0,0,"A"
REM *** B top right ***
TEXT 1260,0,"B"
REM *** C bottom left ***
TEXT 0,990,"C"
REM *** D bottom right ***
TEXT 1260,990,"D"
REM *** End program ***
WAIT KEY
END
```

You may find that this program uses a different resolution than the earlier version did.

Activity 2.15

The program code is:

```
REM *** Output two strings at same
location ***
TEXT 100, 100, “Hello”
WAIT KEY
TEXT 100, 100, “Goodbye”
REM *** End program ***
WAIT KEY
END
```

The second string writes on top of the first without removing it. We'll see a cure for this later in the chapter.

Activity 2.16

For 1248 by 1024, the program code is:

```
CENTER TEXT 623,500, “MIDDLE”
REM *** End program ***
WAIT KEY
END
```

Activity 2.17

The second line of the LISTING-2.3 should be changed to

```
SET TEXT FONT “Times New Roman”
```

Activity 2.18

```
PRINT RGB(255,255,0)
```

displays the value 4294967040

```
PRINT RGB(255,0,255)
```

displays 4294902015

Activity 2.19

The background remains black.

Activity 2.20

The program code is:

```
INK RGB(255,255,0), RGB(255,0,0)
PRINT “Hello”
REM *** Set green foreground ***
INK RGB(0,255,0),0
PRINT “Goodbye”
REM *** End program ***
WAIT KEY
END
```

Activity 2.21

```
REM *** Yellow foreground and red background
***
SET TEXT OPAQUE
INK RGB(255,255,0), RGB(255,0,0)
PRINT “Hello”
REM *** Cyan foreg'nd & magenta backg'nd ***
INK RGB(0,255,255),RGB(255,0,255)
PRINT “Goodbye”
REM *** End program ***
WAIT KEY
END
```

Activity 2.22

No solution required

Activity 2.23

The program code is:

```
REM *** Clear screen to red ***
CLS RGB(255,0,0)
REM *** Set text characteristics ***
SET TEXT FONT “Courier New”
SET TEXT TO BOLD
SET TEXT SIZE 20
REM *** Set colours (yellow and red) ***
INK RGB(255,255,0),RGB(255,0,0)
REM *** Output box ***
TEXT 0, 0, “**********”
TEXT 0,20, “*        *”
TEXT 0,40, “**********”
REM *** Set opaque text ***
SET TEXT OPAQUE
REM *** Set colours (blue and black) ***
INK RGB(0,0,255),RGB(0,0,0)
REM *** Output text ***
TEXT 34,18,"BOX"
REM *** End program
WAIT KEY
END
```

Activity 2.24

The problem can be cured by adding more spaces to the second TEXT statement.

Activity 2.25

```
REM *** Set background colour ***
CLS RGB(126,126,126)
REM *** Set text font and size ***
SET TEXT FONT “Arial”
SET TEXT SIZE 36
INK RGB (255, 0,0) ,0
REM *** Output Text ***
TEXT 100,80, “Goodbye”
WAIT KEY
REM ** Remove text by writing it again in
background colour ***
INK RGB(126,126,126),0
TEXT 100,80, “Goodbye”
REM *** End program ***
WAIT KEY
END
```

Activity 2.26

Existing code is in grey:

```
REM *** Set background colour ***
CLS RGB (126,126,126)
SET TEXT SIZE 36
REM *** Set text font and size ***
SET TEXT FONT “Arial”
SET TEXT SIZE 36
INK RGB (255, 0,0) ,0
REM *** Output Text ***
TEXT 100,80, “Goodbye”
WAIT KEY
REM ** Remove text by writing it again in
background colour ***
INK RGB(126,126,126),0
TEXT 100,80, “Goodbye”
WAIT KEY
REM *** Shadow text ***
INK RGB (0,0,0) ,0
TEXT 102,102, “Hello”
INK RGB (255, 0,0) ,0
TEXT 100, 100, “Hello”
REM *** End program ***
WAIT KEY
END
```

Activity 2.27

Existing code is in grey:

```
REM *** Set background colour ***
CLS RGB (126,126,126)
SET TEXT SIZE 36
REM *** Set text font and size ***
SET TEXT FONT “Arial”
SET TEXT SIZE 36
INK RGB (255, 0,0) ,0
REM *** Output Text ***
TEXT 100,80, “Goodbye”
WAIT KEY
REM ** Remove text by writing it again in
background colour ***
INK RGB(126,126,126),0
TEXT 100,80, “Goodbye”
WAIT KEY
REM *** Shadow text ***
INK RGB (0,0,0) ,0
TEXT 102,102, “Hello”
INK RGB (255, 0,0) ,0
TEXT 100, 100, “Hello”
REM *** Embossed Text ***
INK RGB(0,0,0),0
TEXT 201, 201, “Goodbye”
INK RGB(255,255,255),0
TEXT 199, 199, "Goodbye"
INK RGB(126,126,126),0
TEXT 200, 200, "Goodbye"
REM *** End program ***
WAIT KEY
END
```

8

Selection

Arithmetic Operators

Assignment Statement

Constants

Creating Random Numbers

Input Statement

RANDOMIZE and RND Statements

READ, DATA and RESTORE Statements

String Operations

Testing Sequential Structures

Variables

Variable Names

Program Data

Introduction

Every computer game has to store and manipulate facts and figures (more commonly known as **data**). For example, a program may store the name of a player, the number of lives remaining or the time the player has remaining in which to complete a task.

We group information like this into three basic types:

integer - any whole number, positive, negative or zero
real - any number containing a decimal point
strings - any collection of characters (may include numeric characters)

For example, if player *Daniel McLaren* had *3* lives and *10.6* minutes to complete a game, then:

3 is an example of an integer value,
10.6 is a real value,
and *Daniel McLaren* is an example of a string.

Activity 3.1

Identify which type of value each of the following is:

a) -9
b) abc
c) 18
d) 12.8
e) ?
f) 0
g) -3.0
h) Mary had a little lamb
i) 4 minutes
j) 0.023

Constants

When a specific value appears in a computer program's code it is usually referred to as a **constant**. Hence, in the statement

```
PRINT 7
```

the value 7 is a constant. More specifically, we may refer to constant's type. In the line

```
PRINT "Charlotte", 15, 42.7
```

Charlotte is a **string constant,** *15*, an **integer constant**, and *42.7*, a **real constant**. Notice that in DarkBASIC Pro, string constants always appear within double quotes.

Activity 3.2

Identify the constant types in the following line of code:

```
PRINT "Mary is ", 12, " years old"
```

Variables

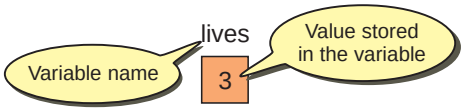
Most programs not only need to display data, but also need to store data and calculate results. To do this in DarkBASIC Pro we need to use a **variable**. A variable is simply somewhere to store a value. Every variable in a program is assigned a unique name and can store a single value. That value might be an integer, a real or a string but each variable is designed to store only one type of value. Hence, a variable designed to store an integer value cannot store a string.

Integer Variables

In DarkBASIC Pro variables are created automatically as soon as we mention them in our code. For example, let's assume we want to store the number of lives allocated

FIG-3.1

Storing Data in a Variable



to a game player in a variable called *lives*. To do this in DarkBASIC Pro we simply write the line:

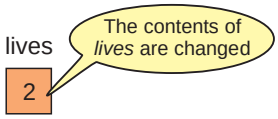
```
lives = 3
```

This sets up a variable called *lives* and stores the value 3 in that variable (see FIG-3.1)

This is known as an **assignment statement** since we are assigning a value (3) to a variable (*lives*).

FIG-3.2

Changing the Value in a Variable



You are free to change the contents of a variable at any time by just assigning it a different value. For example, we can change the contents of *lives* with a line such as:

```
lives = 2
```

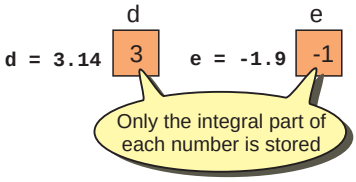
When we do this any previous value will be removed and the new value stored in its place (see FIG-3.2).

The variable *lives* is designed to store an integer value. In the lines below, a, b, c, d, and e are also integer variables. So the following assignments are correct

```
a = 200
b = 0
c = -8
```

FIG-3.3

Trying to Copy a Real Value to an Integer Variable



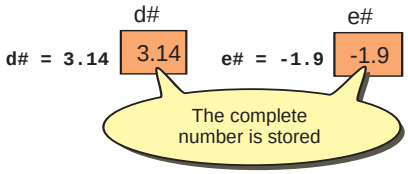
but the lines below are wrong

```
d = 3.14
e = -1.9
```

since they attempt to store real constants in variables designed to hold an integers. DarkBASIC Pro won't actually report an error if you try out these last two examples, it simply ignores the fractional part of the numbers and ends up storing 3 in *d* and 1 in *e* (see FIG-3.3).

FIG-3.4

Creating Real Variables



Real Variables

If you want to create a variable capable of storing a real number, then we must end the variable name with the hash (#) symbol. For example, if we write

```
d# = 3.14
e# = -1.9
```

we have created variables named d# and e#, both capable of storing real values(see FIG-3.4).

Any number can be stored in a real variable, so we could also write a statement such as:

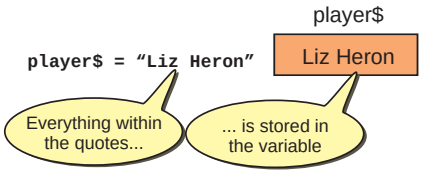
```
d# = 12
```

and this will be stored as 12.0.

If any value can be stored in a real variable, why bother with integer variables? Actually, you should always use integer values wherever possible because the computer is much faster at handling integer values than reals which require much more processing whenever you want to do any calculations. Also, real numbers can be slightly inaccurate because of rounding errors within the machine. For example, the value 2.3 might be stored as 2.2999987.

FIG-3.5

Creating String Variables



String Variables

Finally, if you want to store a string value, you need to use a string variable. String

variable names must end with a dollar ($) sign. The value to be stored must be enclosed in double quotes. We could create a string variable named *player$* and store the name *Liz Heron* in it using the statement:

```
player$ = “Liz Heron”
```

The double quotes are not stored in the variable (see FIG-3.5).

Absolutely any value can be stored in a string variable as long as that value is enclosed in double quotes. Below are a few examples:

```
a$ = “?>%”
b$ = “Your spaceship has been destroyed”
c$ = “That costs $12.50"
```

Activity 3.3

Which of the following are valid DarkBASIC Pro statements that will store the specified value in the named variable?

a) `a = 6`
b) `b = 12.89`
c) `c$ = Hello`
d) `d# = 5`
e) `e$ = ‘Goodbye’`
f) `f# = -12.5`

Using Meaningful Names

It is important that you use meaningful names for your variables when you write a program. This helps you remember what a variable is being used for when you go back and look at your program a month or two after you wrote it.

So, rather than write statements such as

```
a = 3
b = 120
c = 2000
```

a better set of statements would be

```
lives = 3
points = 120
timeremaining = 2000
```

which give a much clearer indication of what the variables are being used for.

Naming Rules

DarkBASIC Pro, like all other programming languages, demands that you follow a few rules when you make up a variable name. These rules are:

- The name should start with a letter.
- Subsequent characters in the name can be a letter, number, or underscore
- The final character can be a # (when creating real variables) or $ (when creating string variables).
- Upper or lower case letters can be used, but such differences are ignored. Hence, the terms *total* and *TOTAL* refer to the same variable.

- The name cannot be a DarkBASIC Pro keyword.

This means that variable names such as

```
a
bc
de_2
fgh$
iJKlmnp#
```

are valid, while names such as

```
2a
time remaining
```

are invalid.

The most common mistake people make is to have a space in their variable names (e.g. *fuel level*). This is not allowed. As a valid alternative, you can replace the space with an underscore (*fuel_level*) or join the words together (*fuellevel*). Using capital letters for the joined words is also popular (*FuelLevel*).

Note that the names *no*, *no#* and *no$* represent three different variables; one designed to hold an integer value (*no*), one a real value (*no#*) and the last a string (*no$*).

Activity 3.4

Which of the following are invalid variable names:

a) `x`
b) `5`
c) `"total"`
d) `al2$`
e) `total score`
f) `ts#o`
g) `end`
h) `G2_F3`

Summary

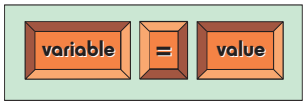
- Fixed values are known as constants.
- There are three types of constants: integer, real and string.
- String constants are always enclosed in double quotes.
- The double quotes are not part of the string constant.
- A variable is a space within the computer's memory where a value can be stored.
- Every variable must have a name.
- A variable's name determines which type of value it may hold.
- Variables that end with the # symbol can hold real values.
- Variables that end with the $ symbol can hold string values.
- Other variables hold integer values.

- The name given to a variable should reflect the value held in that variable.
- When naming a variable the following rules apply:

  The name must start with a letter
  Subsequent characters in the name can be numeric, alphabetic or the underscore character.
  The name may end with a # or $ symbol.
  The name must not be a DarkBASIC Pro keyword.

FIG-3.6

The Assignment Statement



Allocating Values to Variables

Introduction

There are several ways to place a value in a variable. The DarkBASIC Pro statements available to achieve this are described below.

The Assignment Statement

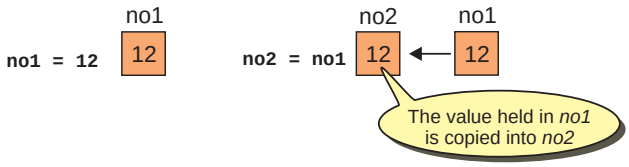
In the last few pages we've used DarkBASIC Pro's assignment statement to store a value in a variable. This statement allows the programmer to place a specific value in a variable, or to store the result of some calculation.

In its simplest form the assignment statement has the form shown in FIG-3.6.

The value copied into the variable may be one of the following types:

FIG-3.7

Copying One Variable's Value to Another Variable



- a constant
- another variable
- an arithmetic expression

Examples of each are shown below.

Assigning a Constant

This is the type of assignment we've seen earlier, with examples such as

```
name$ = "Liz Heron"
```

where a fixed value (a constant) is copied into the variable. Make sure that the constant is the same type as the variable. For instance, the statement

```
desc = "tall"
```

is invalid since it attempts to copy a string constant ("tall") into an integer variable (*desc*). Not every mistake will be signalled by the compiler. For example, if we try to assign a real constant to an integer variable as in the statement

```
result = 12.79
```

the integer variable result stores only the integral part of the constant (i.e. 12), the fractional part being lost.

However, an integer value may be copied into a real variable, as in the line:

```
result# = 33
```

The program deals with this by storing the value assigned to *result#* as 33.0.

TABLE-3.1

Arithmetic Operators

Operator	Function	Example
+	Addition	no1 = no2 + 5
-	Subtraction	no1 = no2 - 9
*	Multiplication	ans = no1 * no2
/	Division	r1# = no1 / 2
mod	Remainder	ans = no2 mod 3
^	Power	ans = 2 ^ 24

Activity 3.5

What are the minimum changes required to make the following statements correct?

1. `desc = “tall”`
2. `result = 12.34`

Copying a Variable’s Value

Once we’ve assigned a value to a variable in a statement such as

```
no1 = 12
```

we can copy the contents of that variable into another variable with a command such as:

```
no2 = no1
```

The effect of these two statements is shown in FIG-3.7.

As before, you must make sure the two variables are of the same type, although the contents of an integer variable may be copied to a real variable as in the lines:

```
ans# = no1
```

Although not invalid, trying this the other way round (real copied to integer) as in

```
ans# = 12.94
no1 = ans#
```

will cause *no1* to store only the integral part of *ans#* contents (i.e. 12).

Activity 3.6

Assuming a program starts with the lines:

```
no1 = 23
weight# = 125.8
description$ = "sword"
```

which of the following instructions would be invalid?

a) `no2 = no1`
b) `no3 = weight#`
c) `result = description$`
d) `ans# = no1`
e) `abc$ = weight#`
f) `m# = description$`

Copying the Result of an Arithmetic Expression

Another variation for the assignment statement is to perform a calculation and store the result of that calculation. Hence we might write

```
no1 = 7 + 3
```

which would store the value 10 in the variable *no1*.

The example shows the use of the addition operator, but there are 5 possible operators that may be used when performing a calculation. These are shown in TABLE-3.1.

The result of most statements should be obvious. For example, if a program begins with the statements

```
no1 = 12
no2 = 3
```

and then contains the line

```
total = no1 - no2
```

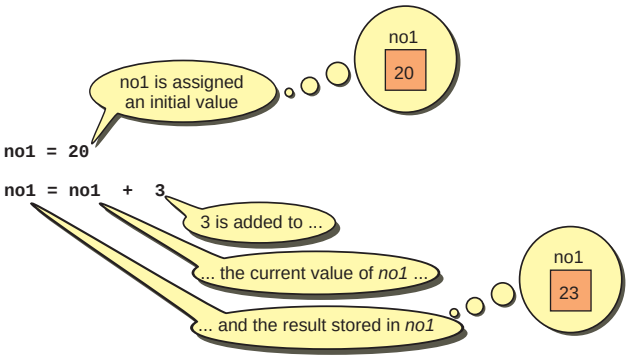
then the variable *total* will contain the value 9, while the line

```
product = no1 * no2
```

stores the value 36 in the variable *product*.

FIG-3.8

Adding to a Variable's Contents



The remainder operator (**mod**) is used to find the integer remainder after dividing one integer into another. For example,

```
ans = 9 mod 5
```

assigns the value 4 to the variable *ans* since 5 divides into 9 once with a remainder of 4. Other examples are given below:

```
6 mod 3        gives 0
7 mod 9        gives 7
123 mod 10     gives 3
```

If the first value is negative, then any remainder is also negative:

```
-11 mod 3      gives -2
```

Activity 3.7

What is the result of the following calculations:

a) `12 mod 5` c) `5 mod 11`
b) `-7 mod 2` d) `-12 mod -8`

The power operator (`^`) allows us to perform a calculation of the form x^y. For example, a 24-bit address bus on the microprocessor of your computer allows 2^{24} memory addresses. We could calculate this number with the statement:

```
addresses = 2^24
```

However, the results of some statements are not quite so obvious. The line

```
ans# = 19/4
```

will result in the value 4.0 being stored in *ans#* since the division operator always returns an integer result if the two values involved are both integer. On the other hand, if we write

```
ans# = 19/4.0
```

and thereby use a real value, then the result stored in *ans#* will be 4.75.

When a real value is copied into an integer variable, the fractional part of the value being copied is lost. For example, the variable *result* would contain the value 4 after executing the line

```
result = 19/4.0
```

When using the division operator, a second situation that you must guard against is division by zero. In mathematics, dividing any number by zero gives an undefined result, so computers get quite upset if you try to get them to perform such a calculation. Hence, the line

```
ans = 10/0
```

would cause a program to crash when it attempted to perform that line in the program. You might be tempted to think that you would never write such a statement, but a more likely scenario is that your program contains a line such as

```
ans = no1 / no2
```

and if *no2* contains the value zero attempting to execute the line will still cause the program to terminate.

Some statements may not appear to make sense if you are used to traditional algebra. For example, what is the meaning of a line such as:

```
no1 = no1 + 3
```

In fact, it means *add 3 to no1*. See FIG-3.8 for a full explanation.

Another unusual assignment statement is:

```
no1 = -no1
```

The effect of this statement is to change the sign of the value held in *no1*. For example, if *no1* contained the value 12, the above statement would change that value to -12. Alternatively, if *no1* started off containing the value -12, the above statement would change *no1*'s contents to 12.

Activity 3.8

Assuming a program starts with the lines:

```
no1 = 2
v# = 41.09
```

what will be the result of the following instructions?

a) `no2 = no1^4`
b) `x# = v#*2`
c) `no3 = no1/5`
d) `no4 = no1 + 7`
e) `m# = no1/5`
f) `v2# = v# - 0.1`

Of course, an arithmetic expression may have several parts to it as in the line

```
answer = no1 - 3 / v# * 2
```

and, how the final result of such lines is calculated is determined by **operator precedence**.

Operator Precedence

If we have a complex arithmetic expression such as

```
answer# = 12 + 18 / 3^2 - 6
```

then there's a potential problem about what should be done first. Will we start by adding 12 and 18 or subtracting 6 from 2, raising 3 to the power 2, or even dividing 18 by 3. In fact, calculations are done in a very specific order according to a fixed set of rules. The rules are that the power operation (^) is always done first. After that comes multiplication and division with addition and subtraction done last. The power operator (^) is said to have a higher priority than multiplication and division; they in turn having a higher priority than addition and subtraction.

So, to calculate the result of the statement above the computer begins by performing

the calculation `3^2` which leaves us with:

```
answer = 12 + 18 / 9 - 6
```

Next the division operation is performed (18/9) giving

```
answer = 12 + 2 - 6
```

The remaining operators, + and -, have the same priority, so the operations are performed on a left-to-right basis meaning that we next calculate 12+2 giving

```
answer = 14 - 6
```

Finally, the last calculation (14 -6) is performed leaving

TABLE-3.2

Variable Range

Variable Type	Range of Values
integer	-2,147,483,648 to + 2,147,483,647
real	±3.4 E ± 38

```
answer = 8
```

and the value 8 stored in the variable *answer*.

Activity 3.9

What is the result of the calculation `12 - 5 * 12 / 10 - 5`

Using Parentheses

If we need to change the order in which calculations within an expression are performed, we can use parentheses. Expressions in parentheses are always done first. Therefore, if we write

```
answer = (12 + 18) / 9 - 6
```

then 12+18 will be calculated first, leaving:

```
answer = 30 / 9 - 6
```

This will continue as follows:

```
answer = 3.3333 - 6
answer = -2.6667
```

An arithmetic expression can contain many sets of parentheses. Normally, the computer calculates the value in the parentheses by starting with the left-most set.

Activity 3.10

Show the steps involved in calculating the result of the expression
`8 * (6-2) / (3-1)`

If sets of parentheses are placed inside one another (this is known as **nested parentheses**) , then the contents of the inner-most set is calculated first. Hence, in the expression

```
12 / (3 * (10 - 6) + 4)
```

the calculation is performed as follows:

```
(10 - 6)   giving  12 / (3*4+4)
3 * 4      giving  12 / (12 + 4)
12 + 4     giving  12 / 16
12 / 16    giving  0.75
```

Activity 3.11

Assuming a program begins with the lines

```
no1 = 12
no2 = 3
no3 = 5
```

what would be the value stored in *answer* as a result of the line

```
answer = no1/(4 + no2 - 1)*5 - no3^2
```

?

Variable Range

When first learning to program, a favourite pastime is to see how large a number the computer can handle, so people write lines such as:

```
no1 = 1234567890
```

They are often disappointed when the program crashes at this point.

There is a limit to the value that can be stored in a variable. That limit is determined by how much memory is allocated to a variable, and that differs from language to language. The range of values that can be stored in DarkBASIC Pro variables is shown in TABLE-3.2.

String Operations

The + operator can also be used on string values to join them together. For example, if we write

```
a$ = “to” + “get”
```

then the value *toget* is stored in variable *a$*. If we then continue with the line

```
b$ = a$ + “her”
```

b$ will contain the value *together*, a result obtained by joining the contents of a$ to the string constant “her”.

Activity 3.12

What value will be stored as a result of the statement

```
term$ = “abc”+"123"+"xyz"
```

The PRINT Statement Again

We’ve already seen that the PRINT command can be used to display values on the

screen using lines such as:

```
PRINT 12
PRINT "Hello"
```

We can also get the PRINT statement to display the answer to a calculation. Hence,

```
PRINT 7+3
```

will display the value 10 on the screen, while the statement

```
PRINT "Hello " + "again"
```

displays *Hello again*.

The PRINT statement can also be used to display the value held within a variable. This means that if we follow the statement

```
number = 23
```

by the line

```
PRINT number
```

our program will display the value 23 on the screen, this being the value held in *number*. Real and string variables can be displayed in the same way. Hence the lines

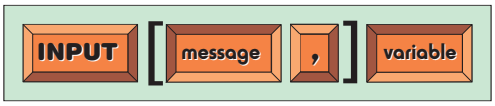
```
name$ = "Charlotte"
weight# = 95.3
PRINT name$
PRINT weight#
```

will produce the output

```
Charlotte
95.3
```

FIG-3.9

The INPUT Statement



Activity 3.13

A program contains the following lines of code:

```
number = 23
PRINT "number"
PRINT number
```

What output will be produced by the two PRINT statements?

Activity 3.14

Type in and test the following program (don't bother to save the program):

```
number = 23
PRINT number
WAIT KEY
END
```

Change the program by removing the first two lines and replacing this with two statements which will assign the value *Jessica McLaren* to a variable called *name$* and then display the contents of *name$* on the screen.

The PRINT statement can display more than one value at a time. For example, we can get it to display the number *12* and the word *Hello* at the same time by writing

```
PRINT 12,"Hello"
```

Each value we want displayed must be separated from the next by a comma. We can use this to display a message alongside the contents of a variable. For example, the lines

```
capital$ = "Washington"
PRINT "The capital of the USA is ", capital$
```

produce the following output on the screen:

```
The capital of the USA is Washington
```

Activity 3.15

Write a program (*name.dbpro*) that sets the contents of the variable *name$* to *Jessica MacLaren* and then uses a PRINT statement that displays the contents of *name$* in such a way that the final message on the screen becomes:

```
Hello, Jessica MacLaren, how are you today?
```

Other Ways to Store a Value in a Variable

The INPUT Statement

There will be many values which we cannot know when we are writing the program. For example, we can't know the name of the player until someone sits down at the computer and begins to play our game. The only way we can get access to that sort of information is to ask the player to type in the information the program requires. This is done with the INPUT statement. In its simplest form the INPUT keyword is followed by the name of the variable where we'd like to store the information the player types in. For example, we might write

```
INPUT name$
```

expecting the person at the keyboard to type in their name and then storing what they type in the variable *name$*. Of course, the player has to be told what sort of information they are expected to enter, so we could precede the INPUT statement with a message telling them what to type in :

```
PRINT “Please enter your name ”
INPUT name$
```

DarkBASIC Pro makes things simpler than this by allowing us to include the message we want displayed as part of the INPUT statement. Hence, we can achieve the same effect as the two statements above using the line:

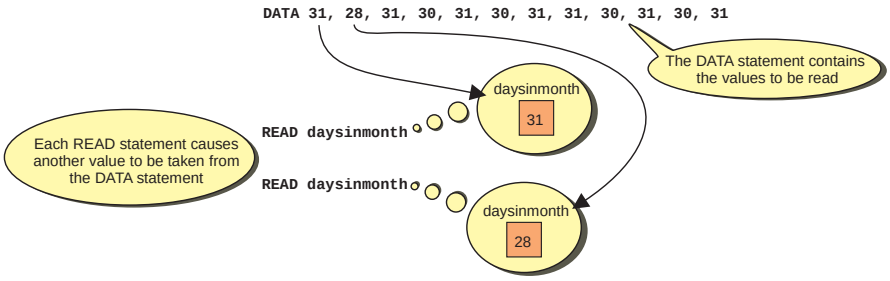
```
INPUT “Please enter your name ”, name$
```

This gives us the final format for the INPUT statement as shown in FIG-3.9.

In the diagram:

FIG-3.10

Using DATA and READ



message	is a string (enclosed in double quotes) which is displayed before any data from the keyboard is accepted.
variable	is a variable name. The value entered by the user at the keyboard will be assigned to this variable. It is the user’s responsibility to enter a value of the correct type.

Activity 3.16

Which of the following are valid INPUT statements?

a) `INPUT age`
b) `INPUT “Enter your height ”, height#`
c) `INPUT “Enter your salary ” salary`

Activity 3.17

Type in and run the following program (*input01.dbpro*):

```
INPUT “Player 1, enter your name :”,name$
PRINT “Hello, ”,name$
WAIT KEY
END
```

We can use the INPUT statement anywhere in our program and as often as necessary.

Activity 3.18

Modify your last program so that it also reads in the age of the player and displays a message of the form:

Hello, {name goes here}, I see you are {age goes here} years old.

The READ and DATA Statements

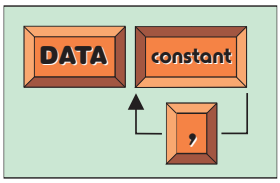
There are times when we want to assign a value to a variable, but we don't want to have to enter that value from the keyboard. For example, let's say a variable, *daysinmonth*, is used to store how many days are in January. The contents of *daysinmonth* is then to be displayed. After this the program stores within *daysinmonth* the number of days in a normal February. Again, the contents of *daysinmonth* is displayed. This continues until every month of the year has been dealt with.

We could start the coding for this as:

```
daysinmonth = 31
```

FIG-3.11

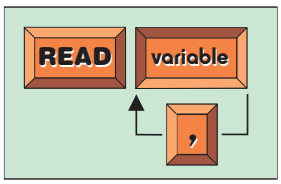
The DATA Statement




```
PRINT daysinmonth
daysinmonth = 28
PRINT daysinmonth
daysinmonth = 31
PRINT daysinmonth
```

FIG-3.12

The READ Statement



Alternatively, we can set up the values we intend to assign to *daysinmonth* in a DATA statement:

```
DATA 31,28,31,30,31,30,31,31,30,31,30,31
```

and then use a READ statement every time we want to assign a value to *daysinmonth.*

```
READ daysinmonth
```

The value given to *daysinmonth* by the READ statement will be the first value listed in the DATA statement. When another READ statement is executed, the second value from the DATA statement will be used. We can therefore rewrite the statements given earlier as:

```
DATA 31,28,31,30,31,30,31,31,30,31,30,31
READ daysinmonth
PRINT daysinmonth
READ daysinmonth
```

```
PRINT daysinmonth
READ daysinmonth
PRINT daysinmonth
```

The operation of these statements is shown in FIG-3.10.

Is this second approach any better than the first? You should have noticed that by using the DATA/READ approach we repeat exactly the same statements over and over again. In a later chapter we will see that this code can be shortened by using a loop statement which would not be possible with the first approach.

Several DATA statements may be used by a program, so we might write:

```
DATA 31, 28
DATA 31, 30
```

The computer simply groups the values given in the DATA statements into a single list, so the two DATA statements above have exactly the same effect as:

```
DATA 31,28,31,30
```

The DATA statement can contain values of any type. The next example stores the

FIG-3.13

The RESTORE Statement

names of the first three days of the week:

```
DATA “Sunday”, “Monday”,"Tuesday"
```

Of course, when you read from this DATA statement, the variable being assigned the value must be a string:

```
READ day$
```

The type of values in a DATA statement can even be mixed, containing integer, real or string constants in any order. It is only important that READ statements use the type of variable appropriate to the next value coming from the DATA statement.

We might write

```
DATA 12, 2.7, “Hello”
```

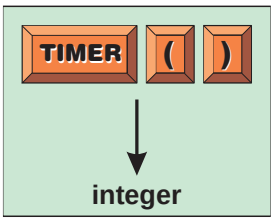
followed by

```
READ no1
READ x#
READ word$
```

and this would be acceptable because variables and values being read are of

FIG-3.14

The TIMER Statement



matching types. That is, the first READ statement would assign the integer value 12 to the integer variable *no1*; the second READ would assign the real value 2.7 to the real variable *x#* and the third READ would assign the string "Hello" to the string variable *word$*. It's also possible to read the value of more than one variable in a single READ statement. Hence, we could reduce the three statements above to the single line:

```
READ no1, x#, word$
```

A DATA statement can be placed anywhere in your program. Often it is placed at the start or end of a program where it can easily be found should the values it holds need to be examined or changed.

The format for the DATA statement is shown in FIG-3.11 and the format of the READ statement is shown in FIG-3.12.

In the diagram:

constant	represents any fixed value. This value can be an integer, real, or string.

In the diagram:

variable	is any variable name. The variable named will be assigned the next available value from the DATA statement.

An error will be reported if your program contains a READ statement but no DATA statement. An error will also occur if a READ statement is executed after all the values in the DATA statement have been used.

Activity 3.19

Write a short program (*days01.dbpro*) which displays the names of the days of the week. Start with Sunday.

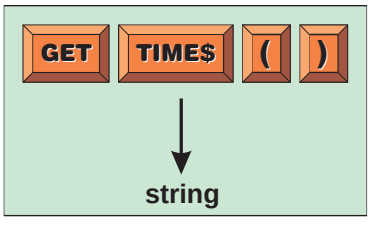
The names should be set up in a DATA statement, then accessed using a series of READ statements.

The RESTORE Statement

DarkBASIC Pro knows which value is to be used next from a DATA statement by keeping a marker which indicates which value in the statement is to be used when the next READ statement is executed.

FIG-3.15

The GET TIME$ Statement



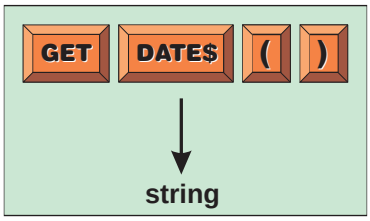
Initially this marker points to the first value in the first DATA statement. After each READ the marker moves on one position. However, it is possible to return the marker to the start of the DATA list by executing the RESTORE statement.

For example, in the code

```
DATA 3,6,9,12
READ no1
READ no2
```

FIG-3.16

The GET DATE$ Statement




```
RESTORE
READ no3
```

the variable *no3* will be assigned the value *3* because the RESTORE statement will have moved the DATA marker back to the first value in the list.

The RESTORE statement has the format shown in FIG-3.13.

Activity 3.20

Modify your last program so that after all the days of the week have been displayed the word *Sunday* is displayed for a second time.

You can achieve this result by adding a RESTORE statement, another READ statement and a PRINT statement to your program.

The Time and Date

The TIMER Statement

DarkBASIC Pro contains a command that lets you find out how long your computer has been switched on. This is the TIMER statement which returns an integer specifying the number of milliseconds that have passed since your machine was last powered up. This information is actually maintained by the operating system and the DarkBASIC Pro statement interrogates the area of computer memory where this data is held.

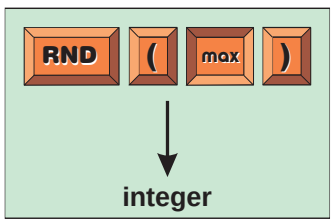
The TIMER statement has the format given in FIG-3.14.

Notice that the parentheses must be included in the statement even though no information is placed within them. DarkBASIC Pro's general syntax demands that any statement that returns a value must always have parentheses.

So the TIMER statement could be used to display how long your machine has been on with the single line

FIG-3.17

The RND Statement



```
PRINT TIMER ()
```

but this would be in milliseconds. Perhaps a better option would be to save the value returned by TIMER and convert that value to seconds, as in the lines:

```
millisecondsPassed = TIMER()
seconds = millisecondsPassed / 1000
PRINT "Your computer has been on for ", seconds, " seconds"
```

Activity 3.21

Create a project (*minutes.dbpro*) which displays how many minutes have passed since your computer was last switched on.

By using TIMER before and after some event we can measure how long that event lasts. For example, we could create a simple reaction time game by seeing how quickly the user can press a key after being told to do so. Such a program requires the following logic:

```
Display "Press any key"
Record the start time
Wait for a key press
Record the finish time
Calculate the duration as finish time minus the start time
Display the duration
```

Activity 3.22

Create a project (*reaction.dbpro*) that implements the logic given above.

The GET TIME$ Statement

If we need to get the actual time of day then we can use the TIME$ statement which returns a string giving the current time (as obtained from the system clock) in the form HH:MM:SS, where HH is the hour (0 to 23), MM is the minutes, and SS the seconds.

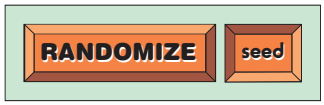
The format for this statement is given in FIG-3.15.

For example, we could display the current time using the line:

```
PRINT GET TIME$()
```

FIG-3.18

The RANDOMIZE Statement



The GET DATE$ Statement

The current date can be returned as a string using the GET DATE$ statement which has the format shown in FIG-3.16.

The string returned is in the American form MM/DD/YY. For example, when run at the time of writing, the statement

```
PRINT GET DATE$()
```

displayed the output 07/16/05.

Generating Random Numbers

Often in a game we need to throw a dice, choose a card or think of a number. All of these are random events. That is to say, we cannot predict what value will be thrown on the dice, what card will be chosen, or what number some other person will think of.

The RND Statement

There is a need to get computer programs to emulate this randomness and this is done using the RND statement. In fact, like RGB, RND is a function. It will generate an integer value within a specified range and return that generated value. For example, if we wanted to display a random number between 0 and 10, we could write

```
PRINT RND(10)
```

LISTING-3.1

Displaying a Random Number

RND has to be supplied with a value enclosed in parentheses. This value lets the command know what range of possible values may be generated. Notice that the lowest value that can be generated is always zero, while the largest value is equal to the number given in the brackets.

Activity 3.23

What expression would we use if we wanted to create a random number in the range 0 to 48?

The format for the RND statement is given m FIG-3.17.

In the diagram:

max	is any positive integer value. The command will return an integer in the range 0 to *max*.

The value given within the parentheses can also be a variable or arithmetic expression, as in the lines:

```
num = 25
PRINT RND (num)        '0 to 25
PRINT RND (num*2-3)    '0 to 47
```

The value returned by RND could be stored in a variable using a statement such as:

```
number = RND(10)
```

If RND(5) generates a number between 0 and 5, how are we going to emulate a dice throw which gives values 1 to 6? Often people suggest writing RND(6), but this gives values in the range 0 to 6, not 1 to 6.

Instead we have to generate a value between 0 and 5 and then add 1 to that number.

We could do this with the line

```
diceThrow = RND(5)+1
```

and follow this with a PRINT statement displaying the contents of *diceThrow*:

```
PRINT "You threw a ",diceThrow
```

The RANDOMIZE Statement

Computers can't really think of a random number all by themselves. Actually, they cheat and use a mathematical formula to calculate an apparently random number. As long as you don't know that formula, you won't be able to predict what number the computer is going to come up with.

But to get the mathematics started correctly, we need to supply it with a start up value or **seed value**. Effectively this seed value determines what numbers the computer is going to generate when RND is used.

The seed value is set up using the RANDOMIZE statement which has the format shown in FIG-3.18.

In the diagram:

seed	is an integer value which is used as a start-up value for the random number generator.

Exactly what seed value you use doesn't really matter, but if you start with the same seed value every time, you'll always get the same set of values from RND. For example, if a program contained the lines

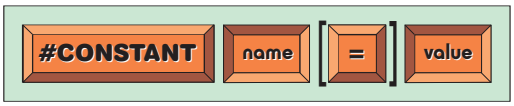
```
RANDOMIZE 12345
PRINT RND(50)
PRINT RND(12)
```

every time that program is executed, the same numbers would be displayed.

To stop this happening we need to make sure that the seed value is different every

FIG-3.19

The #CONSTANT statement



time we run a program. We can achieve this using the TIMER statement. So if we write

```
number = TIMER()
RANDOMIZE number
```

then, since TIMER will return a different value every time it's carried out (remember the time in your computer is being updated 1000 times per second), the seed value for RANDOMIZE will always be different. Actually, we can combine the two statements above into one:

```
RANDOMIZE TIMER()
```

Now we are ready to write a program using random numbers. The program in LISTING-3.1 simulates a dice throw and displays the number generated.

```
REM *** Generate random number ***
RANDOMIZE TIMER()
```

```
number = RND(5)+1

REM *** Display number ***
PRINT number

REM *** End program ***
WAIT KEY
END
```

Activity 3.24

Type in and run the program given above (*random.dbpro*).
Modify the program so that it generates a number between 1 and 49.

Activity 3.25

Write a program (*guess01.dbpro*) that performs the following logic:

Computer thinks of a number between 1 and 100
User (you) enters their guess at what the number is
The computer displays both the guess and the original number

Structured English and Programs

When we write a structured English algorithm with the intention of turning that algorithm into a computer program, we always write the algorithm as if we are telling the computer what it has to do. Therefore, the rather long winded algorithm in the Activity above would be better written as:

Generate a random integer between 1 and 100
Get user's guess
Display number and guess

Using Variables to Store Colour Values

We've seen how the value generated by the RND statement can be stored in a variable with a statement such as:

```
number = RND(5)+1
```

Since the RGB statement also returns a value, we can use that same approach there. So rather than write

```
INK RGB(255,0,0),RGB(0,255,0)
```

we could write

```
colour1 = RGB(255,0,0)
colour2 = RGB(0,255,0)
INK colour1, colour2
```

LISTING-3.2

Calculating the Square Root of a Value

Activity 3.26

Write a program (*colours03.dbpro*) that performs the following operations

Assigns the colour red to a variable called *scarlet*;
Assigns the colour blue to a variable called *sky*;
Clears the screen to create red blank screen (use *scarlet*);
Writes the word *Ocean* in blue on the screen (use *sky*)

Named Constants

When a program uses a fixed value which has an important role (for example, perhaps the value 1000 is the score a player must achieve to win a game), then we have the option of assigning a name to that value using the #CONSTANT statement.

The format of this statement is shown in FIG-3.19.

In the diagram:

name is the name to be assigned to the constant value.

value is the constant value being named.

For example, we can name the value 1000 *WinningScore* using the line:

```
#CONSTANT WinningScore = 1000
```

Since the equal sign (=) is optional, it is also valid to write:

```
#CONSTANT WinningScore 1000
```

Real and string constants can also be named, but the names assigned must NOT end with # or $ symbols. Therefore the following lines are valid

```
#CONSTANT Pi = 3.14159265
#CONSTANT Vowels = “aeiou”
```

The value assigned to a name cannot be changed, so having written

```
#CONSTANT WinningScore = 1000
```

it is not valid to try to assign a new value with a line such as:

```
WinningScore = 1900
```

The two main reasons for using named constants in a program are:

1) Aiding the readability of the program. For example, it is easier to understand the meaning of the line

```
IF playerscore >= WinningScore
```

than

```
IF playerscore >= 1000
```

2) If the same constant value is used in several places throughout a program, it is easier to change its value if it is defined as a named constant. For example, if, when writing a second version of a game we decide that the winning score has to be changed from 1000 to 2000, then we need only change the line

```
#CONSTANT WinningScore = 1000
```

to

```
#CONSTANT WinningScore = 2000
```

On the other hand, if we've used lines such as

```
IF playerscore >= 1000
```

throughout our program, every one of those lines will have to be changed so that the value within them is changed from 1000 to 2000.

Testing Sequential Code

The programs in this chapter are very simple ones, with the statements being executed one after the other, starting with the first and ending with the last. In other words, the programs are **sequential** in structure.

Every program we write needs to be tested. For a simple sequential program that involves input, the minimum testing involves thinking of a value to be entered, predicting what result this value should produce, and then running the program to check that we do indeed obtain the expected result.

The program below (see LISTING-3.2) reads in a value from the keyboard and displays the square root of that number.

```
INPUT "Please enter your number : ", number#
squareroot# = number#^0.5
PRINT "The square root of ", number#, " is ",squareroot#
WAIT KEY
END
```

To test this program we might decide to enter the value 16 with the expectation of the result being 4.

Activity 3.27

Type in the program given above (*root.dbpro*) and test it by inputting the value 16.

Perhaps that would seem sufficient to say that the program is functioning correctly. However, a more cautious person might try a few more values just to make sure. But what values should be chosen? Should we try 25 or 9, 3 or 7?

As a general rule it is best to think carefully about what values you choose as test data. A few carefully chosen values may show up problems when many more

randomly chosen values show nothing.

When the test data is numeric, the most obvious choices are to use a typical value (in the case of the above program, 16 falls into this category), a very large value, a negative value and zero. But in each case it is important that you work out the expected result before entering your test data into the program - otherwise you have no way of knowing if the results you are seeing on the screen are correct.

Activity 3.28

What results would you expect from *root.dbpro* if your test data was

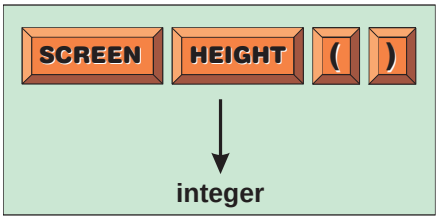
```
401286
     0
    -9
```

Run the program with these test values and check that the expected results are produced.

When entering string test values, an empty string (just press Enter when asked to enter the data), a single character string, and a multicharacter string should do.

FIG-3.20

The SCREEN HEIGHT Statement



These suggestions for creating test data may need to be modified depending on the nature of the program you are testing.

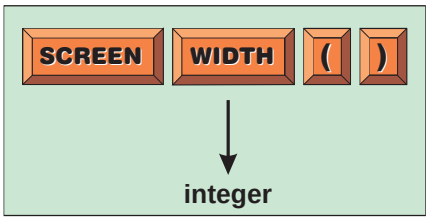
Summary

- The assignment statement takes the form

```
variable = value
```

FIG-3.21

The SCREEN WIDTH Statement



- *value* can be a constant, other variable, or an expression.
- The value assigned should be of the same type as the receiving variable.
- Arithmetic expressions can use the following operators:

```
^ * / + - mod
```

- Calculations are performed on the basis of highest priority operator first and a

LISTING-3.3

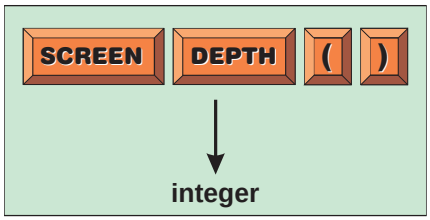
Centring Text

left-to-right basis.

- The power operator has the highest priority; multiplication and division and the mod operator the next highest, followed by addition and subtraction.
- Terms enclosed in parentheses are always performed first.
- The + operator can be used to join strings.
- The INPUT statement reads a value from the keyboard and places that value in a named variable.
- The INPUT statement can display a message designed to inform the user what has to be entered.
- The DATA and READ statements can be used to assign a listed value to a variable.
- The RESTORE statement forces a return to the start of the first DATA statement.
- The TIMER statement returns the time in milliseconds from switch on.
- The GET TIME$ statement returns the current time as a string.
- The GET DATE$ statement returns the current date as a string.
- The RND statement generates a random integer number in the range 0 to a specified maximum.

FIG-3.22

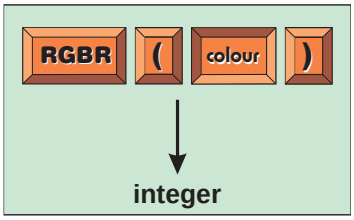
The SCREEN DEPTH Statement



- The RANDOMIZE statement ensures that the numbers created by the RND are truly random.
- The value returned by statements such as RND and RGB can be assigned to a variable.
- A named constant can be created using the #CONSTANT statement.
- The name assigned to a constant must not end with a # or $ symbol.

FIG-3.23

The RGBR Statement



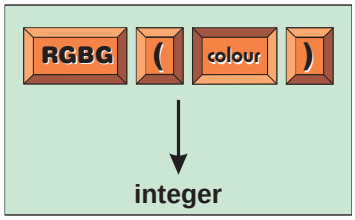
Determining Current Settings

Introduction

Let's say we want to place the title of our new game in the centre of the screen. We know that we can place text at any position using TEXT or CENTER TEXT, but how are we to discover where the centre of the screen is? If we're working in an 800 by 600 display mode, then the centre is at 400,300 - but how can we be sure what display mode is being used? Luckily, DarkBASIC Pro has many statements that allow us to find out this, and other, information. Some of these are given below, others we'll discuss in later chapters.

FIG-3.24

The RGBG Statement



Screen Settings

The SCREEN HEIGHT Statement

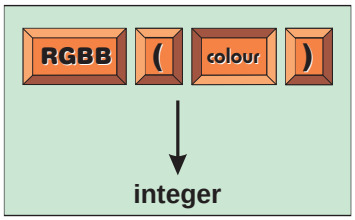
The SCREEN HEIGHT statement returns the height of the output screen in pixels and has the format shown in FIG-3.20.

For example, the statement

```
PRINT SCREEN HEIGHT()
```

FIG-3.25

The RGBB Statement



would display the value 600, assuming the screen resolution was set to 800 by 600.

The SCREEN WIDTH Statement

This statement returns the width of the output screen in pixels. The statement has

the format shown in FIG-3.21.

LISTING-3.4

Colour Component Values

For example, the statement

```
screenwidth = SCREEN WIDTH()
```

would assign the value 800 to the variable *screenwidth*, assuming the screen resolution was set to 800 by 600.

The program in LISTING-3.3 displays the word WELCOME at the centre of the screen.

```
REM *** Find centre of screen ***
centrex = SCREEN WIDTH()/2
centrey = SCREEN HEIGHT()/2

REM *** Display text at centre ***
CENTER TEXT centrex, centrey, "WELCOME"

REM *** End program ***
WAIT KEY
END
```

Activity 3.29

Type in and test the program above (*centred*.dbpro).

Is the text correctly centred both vertically and horizontally?

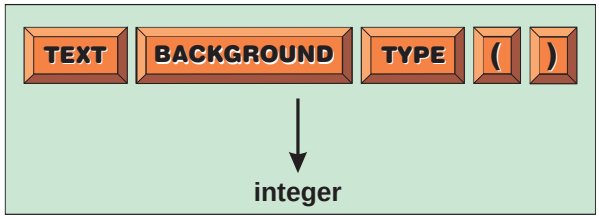
The SCREEN DEPTH Statement

The number of bits used to represent a single pixel on the screen determines the maximum number of colours that can be shown on the screen. For example, if a single bit was used to represent a pixel, that bit could have the value 0 or 1, hence only two colours can be shown. With two bits per pixel, four colours are possible, represented by the bit patterns 00, 01, 10, and 11.

The SCREEN DEPTH statement returns the number of bits used per pixel and has

FIG-3.26

The TEXT BACKGROUND Statement



the format shown in FIG-3.22.

If a call to this statement returns the value 16, then the number of colours that can be shown is calculated as 2^{16}. The code required to perform this calculation is:

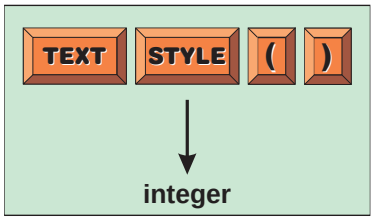
```
noofcolours = 2^SCREEN DEPTH()
```

Colour Components

If we were to generate a random colour with the lines

FIG-3.27

The TEXT STYLE Statement




```
RANDOMIZE TIMER()
colour = RGB(RND(255),RND(255),RND(255))
```

we could find out the settings of the red, green and blue components of that colour

FIG-3.28

The TEXT SIZE Statement

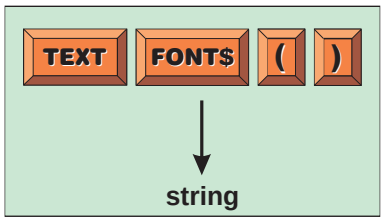
using the following statements.

The RGBR Statement

The RGBR statement returns an integer specifying the red component of a specified

FIG-3.29

The TEXT FONT$ Statement



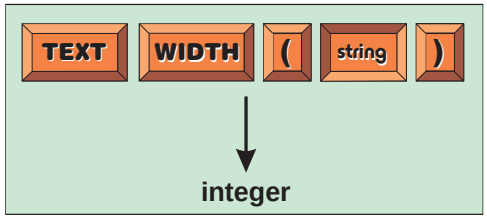
colour. The statement has the format shown in FIG-3.23.

In the diagram:

colour	is an integer value representing a colour. This value will probably have been generated using the RGB statement.

FIG-3.30

The TEXT WIDTH Statement



Hence, assuming the variable colour had been set using the line given earlier, we could extract the red component of that colour with the line

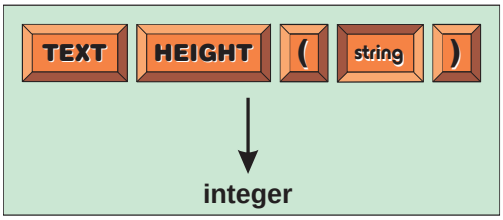
```
redvalue = RGBR(colour)
```

The RGBG Statement

The RGBG statement returns an integer specifying the green component of a specified colour. The statement has the format shown in FIG-3.24.

FIG-3.31

The TEXT HEIGHT Statement



In the diagram:

colour	is an integer value representing a colour. This value will probably have been generated using the RGB statement.

LISTING 3.5

Display Text Characteristics

The RGBB Statement

The RGBB statement returns an integer specifying the blue component of a specified colour. The statement has the format shown in FIG-3.25.

In the diagram:

colour	is an integer value representing a colour. This value will probably have been generated using the RGB statement.

The three statements are used in LISTING-3.4 to display the component values of a randomly generated colour.

```
REM *** Create random colour ***
RANDOMIZE TIMER()
colour = RGB(RND(255),RND(255),RND(255))

REM *** Extract components of this colour ***
red = RGBR(colour)
green = RGBG(colour)
blue = RGBB(colour)

REM *** Use the new colour ***
INK colour,0

REM *** Display the colour details ***
PRINT “The generated colour has the following settings”
PRINT “Red component : ”,red
PRINT “Green component : ”,green
PRINT “Blue component : ”,blue

REM *** End program ***
WAIT KEY
END
```

Activity 3.30

Type in and test the program (*colours03.dbpro*) in LISTING-3.4.

Text Settings

Details of the text font, size and style currently being used by a program can be retrieved using the following statements.

The TEXT BACKGROUND TYPE Statement

We can discover the current text background mode (opaque or transparent) using the TEXT BACKGROUND TYPE statement which has the format shown in FIG-3.26.

The statement returns the value zero if a transparent background is being used; 1 is returned when the background setting is opaque.

The TEXT STYLE Statement

The style of font, (bold, italic, etc.) can be determined using the TEXT STYLE statement which has the format shown in FIG-3.27.

The integer value returned lies between 0 and 3 (0 - normal; 1 - italic; 2 - bold; 3 - bold italic).

The TEXT SIZE Statement

The TEXT STYLE statement returns the current text size setting in points. This statement has the format shown in FIG-3.28.

The TEXT FONT$ Statement

The TEXT FONT$ statement returns a string giving the name of the font currently being used. For example, it would return the string "Arial", assuming this font had been selected earlier, using the SET TEXT FONT statement. The TEXT FONT$ statement has the format shown in FIG-3.29.

The TEXT WIDTH Statement

When placing text on the screen it can be very useful to know in advance just how many pixels wide that piece of text is going to be. The exact width of the text will obviously depend on the text itself, *goodbye* being wider than *hello*, but text font, style and size settings are also going to effect the width of the text. We can find out the exact width of any text to be displayed using the TEXT WIDTH statement. This has the format shown in FIG-3.30.

In the diagram:

string	is the string whose width is to be determined.

The TEXT HEIGHT Statement

The number of pixels from the lowest point on a piece of text (typically at the bottom of letters such as *g* and *y*) to the highest point (on letters such as *t* and *l*) can be found

Activity 3.1

a) Integer
b) String
c) Integer
d) Real
e) String
f) Integer
g) Real
h) String
i) String
j) Real

Activity 3.2

"Mary is" - string
12 - integer
" years old" - string

Activity 3.3

a) Valid
b) Invalid. Integer variable will store 12
c) Invalid. Hello should be enclosed in double quotes("Hello")
d) Valid
e) Invalid. Must be double quotes, not single quotes
f) Valid

Activity 3.4

a) Valid
b) Invalid. Must start with a letter
c) Invalid. Names cannot be within quotes.
d) Valid
e) Invalid. Spaces are not allowed in a name
f) Valid
g) Invalid, **end** is a DarkBASIC Pro keyword
h) Valid

Activity 3.5

```
1.   desc$="tall"
2.   result#= 12.34
```

Activity 3.6

a) Valid
b) Invalid. Fraction part lost
c) Invalid. A string cannot be copied to an integer variable
d) Valid
e) Invalid. A real cannot be copied to a string variable
f) Invalid. A string cannot be copied to a real variable

Activity 3.7

a) 2
b) -1
c) 5
d) -4

Activity 3.8

a) no2 is 16
b) x# is 82.18
c) no3 is zero
d) no4 is 9
e) m# is 0.4
f) v2# is 40.99

Activity 3.9

The result is 1

The expression is calculated as follows:

12-5* 12/10-5
12-60/10-5
12-6-5
6-5

Activity 3.10

Steps

8*(6-2)/(3-1)
8*4/(3-1)
8*4/2
32/2
16

Activity 3.11

```
answer = no1 / (4 + no2 - 1) * 5 - no3 ^ 2
answer = 12  / (4 + 3   - 1) * 5 - 5   ^ 2
answer = 12  / (7       - 1) * 5 - 5   ^ 2
answer = 12  /  6            * 5 - 5   ^ 2
answer = 12  /  6            * 5 - 25
answer = 2                   * 5 - 25
answer = 10                      - 25
answer = -15
```

Activity 3.12

term$ will hold the string *abcl23xyz*

Activity 3.13

Output:

number
23

Activity 3.14

The final version of the program should read:

```
name$ = "Jessica McLaren"
PRINT name$
WAIT KEY
END
```

Activity 3.15

```
REM *** Assign name to variable & display
it ***
name$ = “Jessica McLaren”
PRINT “Hello, ”,name$,", how are you today?"
REM *** End program ***
WAIT KEY
END
```

Activity 3.16

a) Valid
b) Valid
c) Invalid. The comma is missing after the message.

Activity 3.17

No solution required.

Activity 3.18

```
REM *** Get name ***
INPUT “Player 1, enter your name ”, name$
INPUT “Enter your age ”, age
PRINT “Hello, ”, name$, “, I see you are
”,age," years old"
REM *** End program ***
WAIT KEY
END
```

Activity 3.19

```
REM *** Set up names of days of the week ***
DATA
“Sunday”,“Monday”,“Tuesday”,“Wednesday”,
“Thursday”,“Friday”,“Saturday”
REM *** Read and display each day ***
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
REM *** End program ***
WAIT KEY
END
```

Activity 3.20

Existing lines are in grey.

```
REM *** Set up names of days of the week ***
DATA
“Sunday”,“Monday”,“Tuesday”,“Wednesday”,
“Thursday”,“Friday”,“Saturday”
REM *** Read and display each day ***
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$

READ day$
PRINT day$
READ day$
PRINT day$
READ day$
PRINT day$
REM *** Go back to the start of the data ***
RESTORE
REM *** Read and display the first day ***
READ day$
PRINT day$
REM *** End program ***
WAIT KEY
END
```

Activity 3.21

```
millisecondsPassed = TIMER()
seconds = millisecondsPassed / 1000
minutes = seconds / 60
PRINT “Your computer has been on for ”
,minutes, “ minutes”
WAIT KEY
END
```

This could be reduced to just

```
minutes = TIMER()/60000
PRINT “Your computer has been on for ”
,minutes, “ minutes”
WAIT KEY
END
```

Activity 3.22

```
REM *** Display message ***
PRINT “Press any key”
REM *** Record start time ***
start = TIMER()
REM *** Wait for key press ***
WAIT KEY
REM *** Record finish time ***
finish = TIMER()
REM *** Calculate and display duration ***
duration = finish - start
PRINT “You took ”, duration, “ milliseconds”
REM *** End program ***
WAIT KEY
END
```

Activity 3.23

```
RND(48)
```

Activity 3.24

The RND line needs to be changed to read:

```
RND(48) + 1
```

Activity 3.25

```
REM *** Generate random value ***
RANDOMIZE TIMER()
number = RND(99)+1
REM *** Guess the number ***
INPUT “Enter your guess (1 to 100) : ”
,guess
REM *** Display both values
PRINT “Number was ”, number," Guess was “
,guess
```

```
REM *** End program ***
WAIT KEY
END
```

Activity 3.26

```
REM *** Assign colours ***
scarlet = RGB(255,0,0)
sky = RGB(0,0,255)
CLS scarlet
INK sky, scarlet
PRINT "Ocean"
REM *** End program ***
WAIT KEY
END
```

Activity 3.27

No solution required.

Activity 3.28

Test Value	Expected Result
401286	633.471
0	0
-9	Undefined

Activity 3.29

The text is not centred vertically since the CENTER TEXT statement positions the top of the text at the y-ordinate specified. To be correctly centred, the middle of the text would have to positioned at this y-ordinate.

Activity 3.30

No solution required

Activity 3.31

No solution required.

using the TEXT HEIGHT statement, which has the format shown in FIG-3.31.

In the diagram:

string is the string whose l

determined.

The program in LISTING-3.5 demonstrates the use of the statements in this section.

```
REM *** Set text characteristics ***
SET TEXT FONT "Arial"
SET TEXT TO BOLD
SET TEXT SIZE 20
SET TEXT OPAQUE

REM *** Read in text ***
INPUT "Enter text : ", text$

REM *** Display details ***
PRINT "Font used is ",TEXT FONT$()
PRINT "Font style is ",TEXT STYLE()," 0
- normal, 1 - italic, 2 ↳- bold, 3 -
bold italic"
PRINT "Font size is ", TEXT SIZE()," 
points"
PRINT "Text background ",TEXT BACKGROUND
TYPE()," 0 - transparent ↳1 - opaque"
PRINT text$," is ",TEXT WIDTH(text$),"
pixels wide"
PRINT text$," is ",TEXT HEIGHT(text$),"
pixels high"

REM *** End program ***
WAIT KEY
END
```

Activity 3.31

Type in and test the program in LISTING-3.5 (*textdetails.dbpro*).

Summary

- Use SCREEN WIDTH to find the current screen width setting.
- Use SCREEN HEIGHT to find the current screen height setting.
- Use SCREEN DEPTH to find how many bits are used to represent one screen pixel.
- Use RGBR to find the value of the red component in a specified colour.
- Use RGBG to find the value of the green component in a specified colour.
- Use RGBB to find the value of the blue component in a specified colour.

- Use TEXT BACKGROUND TYPE to determine if transparent or opaque backgrounds are being used with text output.
- Use TEXT STYLE to determine the current text style setting.
- Use TEXT SIZE to determine the current text size setting.
- Use TEXT FONT$ to determine the current text font name.
- Use TEXT WIDTH to determine the width of a specified piece of text.
- Use TEXT HEIGHT to determine the height of a specified piece of text.

Solutions

Selection

AND, OR and NOT Operators

Boolean Conditions

IF..ENDIF Statement

IF..THEN Statement

Nested IF Statements

Relational Operators

SELECT Statement

Testing Selective Structures

Binary Selection

Introduction

As we saw in structured English, many algorithms need to perform an action only when a specified condition is met. The general form for this statement was:

```
IF condition THEN
      action
ENDIF
```

Hence, in our guessing game we described the response to a correct guess as:

```
IF guess = number THEN
      Say "Correct"
ENDIF
```

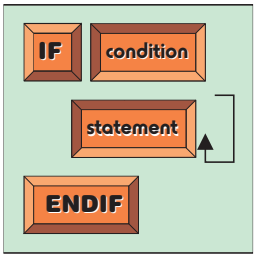
As we'll see, DarkBASIC Pro also makes use of an IF statement to handle such situations.

The IF Statement

In its simplest form the IF statement in DarkBASIC Pro takes the format shown in FIG-4.1.

FIG-4.1

The Simple IF Statement



Notice that DarkBASIC Pro's IF statement does not contain the word THEN

In the diagram:

condition	is any term which can be reduced to a true or false value.
statement	is any executable DarkBASIC Pro statement.

If condition evaluates to true, then the set of statements between the IF and ENDIF terms are executed; if condition evaluates to false, then the set of statements are ignored and execution moves on to the statements following the ENDIF term.

An unlimited number of statements may be placed between the IF and ENDIF terms.

Condition

Generally, the condition will be an expression in which the relationship between two quantities is compared. For example, the condition

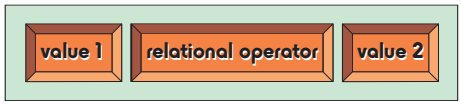
```
no < 0
```

will be true if the content of the variable *no* is less than zero (i.e. negative).

A condition is sometimes referred to as a **Boolean expression** and has the general format given in FIG-4.2.

FIG-4.2

A Boolean Expression



In the diagram:

value1 and *value2* may be constants, variables, or expressions

relational operator is one of the symbols given in TABLE-4.1.

TABLE-4.1

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	<>

The values being compared should be of the same type, but it is acceptable to mix integer and real numeric values as in the conditions:

```
v > x#
t# < 12
```

However, numeric and string values cannot be compared. Therefore, conditions such as

```
name$ = 34
no1 <> "16"
```

are invalid.

Activity 4.1

Which of the following are not valid Boolean expressions?

a) `no1 < 0`
b) `name$ = "Fred"`
c) `no1 * 3 >= no2 - 6`
d) `v# => 12.0`
e) `total <> "0"`
f) `address$ = 14 High Street`

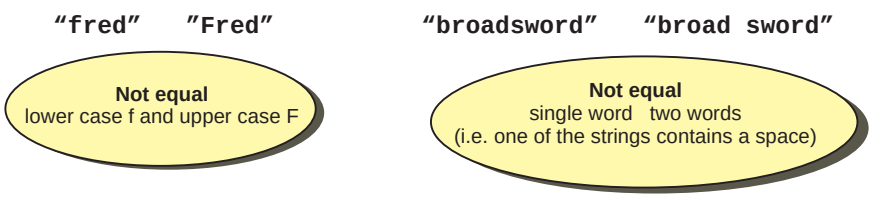
When two strings are checked for equality as in the condition

```
IF name$ = "Fred"
```

the condition will only be considered true if the match is an exact one (see FIG-4.3), even the slightest difference between the two strings will return a false result.

FIG-4.3

Comparing Strings



Not only is it valid to test if two string values are equal, or not, as in the conditions

```
IF name$ = "Fred"
IF village$ <> "Turok"
```

it is also valid to test if one string value is greater or less than another. For example, it is true that

```
"B" > "A"
```

Such a condition is considered true not because B comes after A in the alphabet, but because the coding used within the computer to store a "B" has a greater numeric value than the code used to store "A".

The method of coding characters is known as **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). This coding system is given in Appendix A at the back of the book.

If you are comparing strings which only contain letters, then one string is less than another if that first string would appear first in an alphabetically ordered list. Hence,

"Aardvark" is less than "Abolish"

But watch out for upper and lower case letters. All upper case letters are less than all lower case letters. Hence, the condition

```
"A" < "a"
```

is true.

If two strings differ in length, with the shorter matching the first part of the longer as

```
"abc" < "abcd"
```

then the shorter string is considered to be less than the longer string. Also, because the computer compares strings using their internal codes, it can make sense of a condition such as

```
"$" < "?"
```

which is also considered true since the $ sign has a smaller value than the ? character in the ASCII coding system.

Activity 4.2

Determine the result of each of the following conditions (true or false). You may have to examine the ASCII coding at the end of the book for part f).

a) `"wxy" = "w xy"`
b) `"def" < "defg"`
c) `"AB" < "BA"`
d) `"cat" = "cat."`
e) `"dog" = "Dog"`
f) `"*" > "&"`

TABLE-4.2 shows some Structured English IF statements and the DarkBASIC Pro equivalents.

TABLE-4.2

Examples of Simple IF Statements

Structured English	DarkBASIC Pro Code

IF *no* is negative THEN
  make *no* positive
ENDIF
	IF no < 0
	  no = -no
	ENDIF

IF *day* is zero THEN
  Display “Sunday”
ENDIF
	IF day = 0
	  PRINT “Sunday”
	ENDIF

IF *value* is even THEN
  Subtract 1 from *value*
ENDIF
	IF value mod 2 = 0
	  value = value - 1
	ENDIF

The program in LISTING-4.1 reads in two numbers and displays a message if the numbers are equal. The program employs the following logic:

```
Get values for no1 and no2
IF no1 = no2 THEN
      Display “Numbers are equal”
ENDIF
```

LISTING-4.1

Using a Simple IF Statement

```
REM *** Read in two numbers ***
INPUT “Enter first value : ”,no1
INPUT “Enter second value ”,no2

REM *** IF both numbers are the same THEN Display message ***
IF no1 = no2
   PRINT “Numbers are equal”
ENDIF

REM *** End program ***
WAIT KEY
END
```

Notice the use of indentation in the program listings. DarkBASIC Pro does not demand that this be done, but indentation makes a program easier to read - this is particularly true when more complex programs are written.

Activity 4.3

Type in and test the program in LISTING-4.1 (Call the project *same.dbpro*)

Modify the program you created for project *guess.dbpro*, so that, after the player has typed in his guess, the program displays the word *Correct* if the guess and number are equal.

In the next program (see LISTING-4.2) a real value representing the radius of a circle is read from the keyboard. As long as a valid value has been entered (i.e. a value greater than zero) then the area of the circle is calculated and displayed.

Notice that this time we have more than one statement within the IF structure.

LISTING-4.2

Placing Several Statements within the IF..ENDIF Structure

```
REM *** Read radius of circle ***
INPUT “Enter radius : ”, radius#

REM *** IF valid radius THEN ***
IF radius# > 0
   REM *** Calculate and display area ***
   area# = 3.14159 * radius# * radius#
   PRINT “Area of circle is ”,area#
ENDIF
```

continued on next page

LISTING-4.2
(continued)

Placing Several Statements within the IF..ENDIF Structure

```
REM *** End program ***
WAIT KEY
END
```

Activity 4.5

Write separate DarkBASIC Pro programs for each of the following tasks: (Name the projects *act4_5_1.dbpro*, *act4_5_2.dbpro*, etc.)

1. Read in an integer number (*no1*) and display the message “Negative value” if the number is less than zero.

2. Read in a real number representing the width and height of a square. If the number is greater than zero, calculate and display the area of the square.

3. Read in a word. If the word is “yes”, display the message “Access granted".

4. Read in an integer value and display the word “Even” if it is an even number (HINT: an even number gives no remainder when divided by 2).

Compound Conditions - the AND and OR Operators

Two or more simple conditions (like those given earlier) can be combined using either the term AND or the term OR (just as we did in structured English in Chapter 1).

The term AND should be used when we need two conditions to be true before an action should be carried out. For example, if a game requires you to throw two sixes to win, this could be written as:

```
RANDOMIZE TIMER ()
dice1 = RND(5) + 1
dice2 = RND(5) + 1
IF dice1 = 6 AND dice2 = 6
    PRINT “You win!”
ENDIF
```

The statement **`PRINT “You win!”`** will only be executed if both conditions, *dice1* = *6* and *dice2* = *6*, are true.

Activity 4.6

Using the code given above, if *dice1* = 6 and *dice2* = 5, will the statement **`PRINT “You win!”`** be carried out?

You may recall from Chapter 1 that there are four possible combinations for an IF statement containing two simple expressions. Because these two conditions are linked by the AND operator, the overall result will only be true when both conditions are true. These combinations are shown in TABLE-4.3.

TABLE-4.3

The AND Operator

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

We link conditions using the OR operator when we require only one of the conditions given to be true. For example, if a dice game produces a win when the total of two dice is either 7 or 11, we could write the code for this as:

```
RANDOMIZE TIMER ()
dice1 = RND(5) + 1
dice2 = RND(5) + 1
total = dice1 + dice2
IF total = 7 OR total = 11
    PRINT "You win!"
ENDIF
```

Again, the computer reduces the individual Boolean expressions to either true or false. If at least one of the individual conditions is true, then the overall result is also true. This time the four possible combinations give the results shown in

TABLE-4.4

The OR Operator

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

TABLE-4.4

Activity 4.7

If *no1* =10 and *no2* = 7, which of the following IF statements will evaluate to true?

a) `IF no1 < no2 OR no2 = 8`
b) `IF no1 + no2 > 15 OR no1 < 9`
c) `IF no2 - no1 > 0 OR no1 / no2 > 1`
d) `IF no1 >= 10 OR no2 <= 10`

There is no limit to the number of conditions that can be linked using AND and OR. For example, a statement of the form

```
IF condition1 AND condition2 AND condition3
```

means that all three conditions must be true, while the statement

```
IF condition1 OR condition2 OR condition3
```

means that at least one of the conditions must be true.

Activity 4.8

A game requires 3 dice to be thrown. If at least two dice show the same value, the player has won.

Write a program (*dice.dbpro*) which contains the following logic:

```
Throw all three dice
IF any two dice match THEN
    Display "You win!"
ENDIF
Display the value of each dice
```

Activity 4.9

Modify your previous project *Act4_5_3* so that the message "Access granted" is displayed if the word input is either "yes" or "YES".

Once we start to create conditions containing both AND and OR operators, we must remember that the AND operator takes precedence over the OR operator. Therefore, the statement

```
IF dice = 5 OR dice = 2 AND card$ = "Ace"
```

means that throwing a dice value of 5 is sufficient to give us an overall result of true and it does not matter what value *card$* is. However, it we don't throw a 5, then we must throw a 2 and *card$* must be equal to "Ace" to achieve an overall true result.

The normal rule of performing the AND operation before OR can be modified by the use of parentheses. Expressions within parentheses are always evaluated first. Hence, if we write

```
IF (dice = 5 OR dice = 2) AND card$ = "Ace"
```

the expression will be calculated as follows:

```
    (true OR false)     AND     false
=       true            AND     false
=   false
```

Activity 4.10

What is the overall result of the Boolean expression

```
(score > 20 OR lives > 2) AND (weaponpower < 1 OR ammunition >= 200)
```

when *score* =15, *lives* = 3, *weaponpower* = 1, and *ammunition* = 250

The NOT Operator

DarkBASIC Pro's NOT operator works in exactly the same way as that described in Chapter 1. It is used to negate the final result of a Boolean expression.

If we assume *dice* = 4, then the line

```
IF NOT (dice = 5 OR dice = 2)
```

will evaluate as

```
      NOT (false OR false)
=     NOT        false
=     true
```

Activity 4.11

When *money* =100 and *cards* =21, what is the result of the condition:

```
NOT (money > 80 AND cards > 20)
```

ELSE - Creating Two Alternative Actions

In its present form the IF statement allows us to perform an action when a given condition is met. But sometimes we need to perform an action only when the condition is not met. For example, when the user has to guess the number generated by the computer, we use an IF statement to display the word "Correct" when the user guesses the number correctly:

```
IF guess = number
    PRINT "Correct"
ENDIF
```

However, shouldn't we display an alternative message when the player is wrong?

One way to do this is to follow the first IF statement with another testing the opposite condition:

```
IF guess = number
    PRINT "Correct"
ENDIF
IF NOT guess = number
    PRINT "Wrong"
ENDIF
```

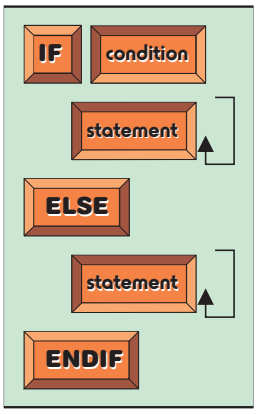
Although this will work, it's not very efficient since we always have to test both conditions - and the second condition can't be true if the first one is!

As an alternative, we can add the word ELSE to our IF statement and follow this by the action we wish to have carried out when the stated condition is false:

```
IF guess = number
    PRINT "Correct"
```

FIG-4.4

The IF..ELSE Statement



```
ELSE
    PRINT "Wrong"
ENDIF
```

Activity 4.12

Modify *act4_5_1.dbpro* to display the phrase "Positive number" if the variable *no1* is greater than or equal to zero and displays the phrase "Negative number" if *no1* contains a value less than zero.

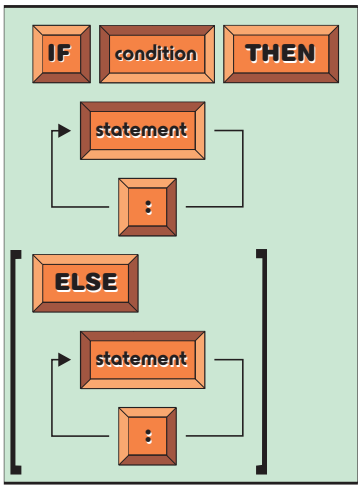
This gives us the longer version of the IF statement format as shown in FIG-4.4.

Activity 4.13

Modify your *guess.dbpro* project so that the message "Wrong" appears if the player guesses the wrong number.

FIG-4.5

The Alternative IF Statement



Although the syntax diagram shows the IF statement spread over several lines, this statement must be entered as a single line in your program.

Activity 4.14

Create a project (*smaller.dbpro*) which reads in two numbers from the keyboard and displays the smaller of the two values.

Activity 4.15

Modify project *act4_5_4.dbpro* so that the program displays the word "Odd" if an odd value is entered.

The Other IF Statement

DarkBASIC Pro actually offers a second version of the IF statement which has the format shown in FIG-4.5.

As you can see from the diagram, this version uses the word THEN but omits the ENDIF term. You can have as many statements as you need in each section (after THEN and ELSE) but these must be separated by colons.

A major restriction when using this version of the IF statement is that the keyword ELSE, if used, must appear on the same line as the term IF. Hence, it is invalid to write:

```
IF no1 < 0 THEN
    PRINT "Negative"
ELSE
    PRINT "Positive"
```

Instead you must write

```
IF no1 < 0 THEN PRINT "Negative" ELSE PRINT "Positive"
```

Activity 4.16

Rewrite the IF statement you created in Activity 4.12 to use this alternative version of the IF statement.

It is probably best to avoid this version of the IF statement, since the requirement to place the IF and ELSE terms on the same line does not allow a good layout for the program code.

Activity 4.17

1. What is a Boolean expression?
2. How many relation operators are there?
3. If a condition contains both AND and OR operators, which will be performed first?

Summary

- Conditional statements are created using the IF statement.
- A Boolean expression is one which gives a result of either true or false.
- Conditions linked by the AND operator must all be true for the overall result to be true.
- Only one of the conditions linked by the OR operator needs to be true for the overall result to be true.
- When the NOT operation is applied to a condition, it reverses the overall result.
- The statements following a condition are only executed if that condition is true.
- Statements following the term ELSE are only executed if the condition is false.
- A second version of the IF statement is available in DarkBASIC Pro in which IF and ELSE must appear on the same line.

Multi-Way Selection

Introduction

A single IF statement is fine if all we want to do is perform one of two alternative actions, but what if we need to perform one action from many possible actions? For example, what if we need to select from three or more alternative actions? How can we create code to deal with such a situation? In structured English we use a modified IF statement of the form:

```
IF
    condition 1:
        action1
    condition 2:
        action 2
    ELSE
        action 3
ENDIF
```

However, this structure is not available in DarkBASIC Pro and hence we must find some other way to implement multi-way selection.

Nested IF Statements

One method is to use nested IF statements - where one IF statement is placed within another. For example, let's assume in our number guessing game that we want to display one of three messages: *Correct, Your guess is too high*, or *Your guess is too low*. Our previous solution allowed for two alternative messages: *Correct* or *Wrong* and was coded as:

```
IF guess = number
    PRINT "Correct"
ELSE
    PRINT "Wrong"
ENDIF
```

LISTING-4.3

The Number Guessing Game Again

In this new problem the **`PRINT "Wrong"`** statement needs to be replaced by the two alternatives: *Your guess is too high*, or *Your guess is too low*. But we already know how to deal with two alternatives - use an IF statement. In this case, our IF statement

```
IF guess > number
    PRINT "Your guess is too high"
ELSE
    PRINT "Your guess is too low"
ENDIF
```

If we now remove the **`PRINT "Wrong"`** statement from our earlier code and substitute the four lines given above, we get:

```
IF guess = number
    PRINT "Correct"
ELSE
    IF guess > number
        PRINT "Your guess is too high"
    ELSE
        PRINT "Your guess is too low"
    ENDIF
ENDIF
```

Activity 4.18

Modify your *guess.dbpro* project so that the game will respond with one of three messages as shown in the code given above.

Activity 4.19

In *act4_5_1.dbpro* we created an IF statement which displayed one of two messages: *Positive Number* or *Negative number*.

Technically, the number zero is neither positive nor negative, hence we should really produce a third message: *Zero* when *number* = 0.

Modify your earlier solution to this previous task to achieve this requirement.

There is no limit to the number of IF statements that can be nested. Hence, if we required four alternative actions, we might use three nested IF statements, while four nested IF statements could handle five alternative actions. To demonstrate this we'll take our number guessing game a stage further and display the message *Your guess is slightly too high* if the guess is no more than 5 above the original number; the message *Your guess is slightly too low* will be displayed if the guess is no more than 5 below the original number.

We'll start by working out the difference between our guess and the computer's number using the line

```
difference = guess - number
```

Now, if we've guessed the number correctly, then *difference* will be zero. However, if we've gone too high, then *difference* will be a positive number. On the other hand, a low guess will result in difference being negative. When difference is a small value (either positive or negative) then guess must be close to number. The complete program is given in LISTING-4.3.

```
REM *** Generate number ***
RANDOMIZE TIMER()
number = RND(99)+1
REM *** Get guess ***
INPUT "Enter your guess (1 - 100) ", guess
REM *** Calculate difference between the two values ***
difference = guess - number
REM *** Display appropriate message ***
IF difference = 0
       PRINT "Correct"
ELSE
   IF difference > 0
       IF difference <= 5
           PRINT "Your guess is slightly too high"
       ELSE
           PRINT "Your guess is too high"
       ENDIF
   ELSE
       IF difference >= -5
           PRINT "Your guess is slightly too low"
       ELSE
           PRINT "Your guess is too low"
       ENDIF
   ENDIF
ENDIF
```

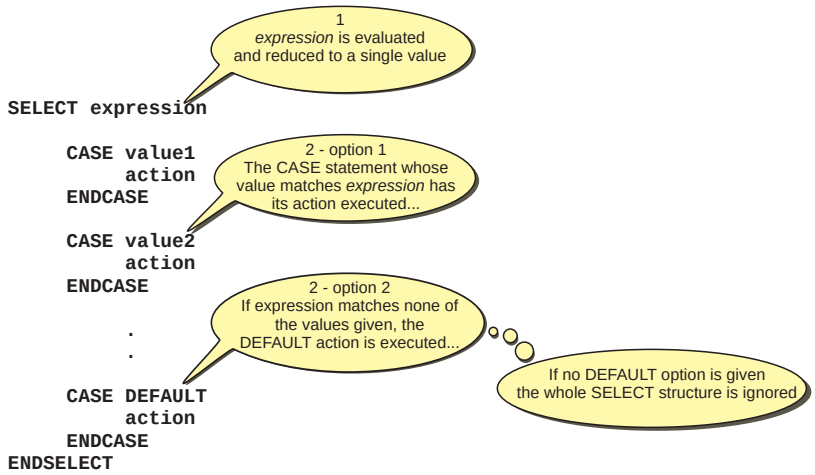
Activity 4.20

Modify your *guess.dbpro* program to match the code given in LISTING-4.3.

Test the program to check that it operates as expected.

FIG-4.6

How the SELECT statement operates



Activity 4.21

In a game a player's character carries the following items: a sword, a wand, a bag of dragon's teeth and a water skin.

Create a new project (*items.dbpro*) which reads in a number from the keyboard and displays the name of the corresponding item. Hence, if 1 is entered, the phrase *A sword* is displayed, if 2 is entered, *A wand* is displayed, etc. If an invalid value is entered, the phrase *Unknown item* is displayed.

The SELECT Statement

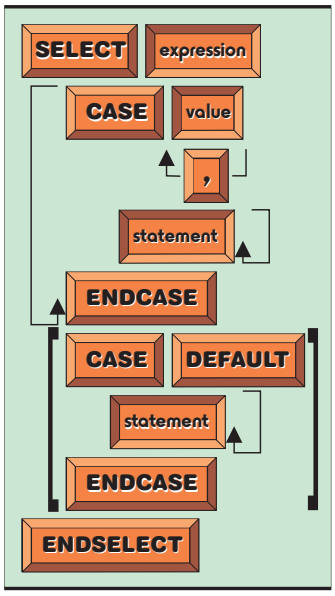
An alternative, and often clearer, way to deal with choosing one action from many is to employ the SELECT statement. The simplest way to explain the operation of the SELECT statement is simply to give you an example. In the code snippet given below we display the name of the day of week corresponding to the number entered. For example, entering 1 results in the word *Sunday* being displayed.

```
INPUT "Enter a number between 1 and 7 ", day
SELECT day
    CASE 1
        PRINT "Sunday"
    ENDCASE
    CASE 2
        PRINT "Monday"
    ENDCASE
    CASE 3
```

```
        PRINT "Tuesday"
    ENDCASE
    CASE 4
        PRINT "Wednesday"
    ENDCASE
    CASE 5
        PRINT "Thursday"
    ENDCASE
    CASE 6
```

FIG-4.7

The SELECT Statement




```
        PRINT "Friday"
    ENDCASE
    CASE 7
        PRINT "Saturday"
    ENDCASE
ENDSELECT
```

Once a value for *day* has been entered, the SELECT statement chooses the CASE statement that matches that value and executes the code given within that section. All other CASE statements are ignored and the instruction following the END SELECT statement (not shown above) is the next to be executed. For example, if *day* = 3, then the statement given beside CASE 3 will be executed (i.e. **`PRINT "Tuesday"`**). If *day* were to be assigned a value not given in any of the CASE statements (i.e. a value outside the range 1 to 7), the whole SELECT statement would be ignored and no part of it executed.

Optionally, a special CASE statement can be added at the end of the SELECT statement. This is the CASE DEFAULT option which is used to catch all other values which have not been mentioned in previous CASE statements. For example, if we modified our SELECT statement above to end with the code

```
    CASE 7
        PRINT "Saturday"
    ENDCASE
    CASE DEFAULT
        PRINT "Invalid day"
    ENDCASE
ENDSELECT
```

then, if a value outside the range 1 to 7 is entered, this last CASE statement will be executed. FIG-4.6 shows how the SELECT statement is executed.

Several values can be specified for each CASE option. If the SELECT value matches any of the values listed, then that CASE option will be executed. For example, using the lines

```
INPUT “Enter a number ” , num
SELECT num
    CASE 1, 3, 5, 7, 9
        PRINT “Odd”
    ENDCASE
    CASE 2,4,6,8,10
        PRINT “Even”
    ENDCASE
ENDSELECT
```

the word *Odd* would be displayed if any odd number between 1 and 9 was entered.

The values given beside the CASE keyword may also be a string as in the example below:

```
INPUT “Enter your name ”, name$ ‘
SELECT name$
    CASE “Liz”,"John"
        PRINT “Hello friend”
    ENDCASE
    CASE DEFAULT
        PRINT “I do not know your name”
    ENDCASE
ENDSELECT
```

Although the value may also be a real value as in the line

```
CASE 1.52
```

it is a bad idea to use these since the machine cannot store real values accurately. If a real variable contained the value 1.52000001 it would not match with the CASE value given above.

The general format of the SELECT statement is given in FIG-4.7.

In the diagram:

expression	is a variable or expression which reduces to a single integer, real or string value.
value	is a constant of any type (integer, real or string).
statement	is any valid DarkBASIC Pro statement (even another SELECT statement!).

TABLE-4.5

Testing Complex Conditions

dice1 = 6	dice2 = 6	dice1 = 6 AND dice2 = 6
false	false	false
false	true	false
true	false	false
true	true	true

Activity 4.22

Rewrite the *Items* program so that it uses a SELECT structure when determining which message is to be displayed.

Activity 4.23

Write a project (*Grading*) which accepts a score from the keyboard and displays the grade assigned according to the following rules:

Score 0-99	grade: Pathetic
Score 100-199	grade: Beginner
Score 200 - 299	grade: Apprentice
Score 300-399	grade: Competent
Score 400-499	grade: Master
Score 500-599	grade: Grand Master
Other values	Invalid score

TABLE-4.6

Dealing with Impossible Combinations

dice = 5	dice = 2	card$="Ace"	dice = 5 OR dice = 2 AND card$="Ace"
false	false	false	false
false	false	true	false
false	true	false	false
false	true	true	true
true	false	false	true
true	false	true	true
true	true	false	* true
true	true	true	* true

Testing Selective Code

When a program contains IF or SELECT structures, our test strategy has to change to cope with these structures. In the case of an IF statement, we must create two test values: one which results in the IF statement being true, the other in the IF statement being false. For example, if a program contained the lines

```
INPUT no
IF no <= 0
    PRINT "This is a negative number"
ENDIF
```

then we need to have a test value for *no* which is less than zero and another which is not less than zero. Perhaps the values -8 and 3.

Another important test is to find out what happens when the variable's value is exactly equal to the value against which it is being tested. In the above case that would mean testing the code with *no* set to 0. Very often this is the only value which will highlight a problem in the code.

Activity 4.24

If *no* is zero, will the message "*This is a negative number*" be displayed by the code given above?

Since zero is not a negative number we have discovered an error in our code. The line

```
IF no <= 0
```

should actually read

```
IF no < 0
```

We would not have detected this error if we hadn't used zero as our test value.

When an IF statement contains more than one condition linked with AND or OR operators, testing needs to check each possible combinations of true and false settings. For example, if a program contained the line

```
IF dice1 = 6 AND dice2 = 6
```

then our tests should include all possible combinations for the two conditions as shown in TABLE-4.5.

So our test values, chosen to meet these combinations, might be

dice1 = 3 dice2 = 5
dice1 = 4 dice2 = 6
dice1 = 6 dice2 = 1
dice1 = 6 dice2 = 6

If the dice values are randomly generated in the program we would have to change lines such as

```
dice1 = RND(5) +1
```

to

```
INPUT "Enter value for dice 1 : ",dice1
```

to allow the test to take place. Once the tests have been completed, the INPUT lines would be replaced by the original code.

In a complex condition it is sometimes not possible to create every theoretical combination of true and false combinations. For example, if a program contains the line

```
IF dice = 5 OR dice = 2 AND card$ = "Ace"
```

then the combinations of true and false are shown in TABLE-4.6.

But the last two combinations in the table are impossible to achieve since the variable *dice* cannot contain the values 5 and 2 at the same time. So our test data will have test values which create only the remaining 6 combinations.

When testing nested IF statements, as in the lines

```
IF guess = number
    PRINT "Correct"
ELSE
    IF guess > number
        PRINT "Your guess is too high"
    ELSE
        PRINT "Your guess is too low"
    ENDIF
ENDIF
```

then each path through the structure must be tested. For the above code this means that we must test for the following conditions being true:

Activity 4.1

a) Valid
b) Valid
c) Valid
d) Invalid. => is not a relational operator (should be >=)
e) Invalid. Integer variable compared with string.
f) Invalid. 14 High Street should be in double quotes.

Activity 4.2

a) False. Only the second string contains a space.
b) True. "def is shorter and matches the first three characters of "defg".
c) True A comes before B.
d) False Only the second string contains a full stop.
e) False Only the second string contains a capital D.
f) True. * has a greater ASCII coding than &

Activity 4.3

No solution required.

Activity 4.4

```
REM *** Generate random value ***
RANDOMIZE TIMER ()
number = RND (99)+1
REM *** Guess the number ***
INPUT "Enter your guess (1 to 100) : "
,guess
REM *** IF the guess is correct THEN ***
REM *** Display "Correct" ***
IF guess = number
  PRINT "Correct"
ENDIF
REM *** Display both values ***
PRINT "Number was ", number," Guess was "
,guess
REM *** End program ***
WAIT KEY
END
```

Activity 4.5

1.

```
REM *** Read in integer ***
INPUT "Enter a number : ", no1
REM *** IF neg THEN Display message ***
IF no1 < 0
  PRINT "Negative value"
ENDIF
REM *** End program ***
WAIT KEY
END
```

2.

```
REM *** Read in real ***
INPUT "Enter length of side : ", side#
REM *** IF greater than zero THEN ***
IF side# > 0
  REM *** Calculate and display area ***
  area# = side# * side#
  PRINT "Area of square is ",area#
ENDIF
REM *** End program ***
WAIT KEY
END
```

3.

```
REM *** Read in word ***
INPUT "Enter a word : ",word$
IF word$ = "yes"
  PRINT "Access allowed"
ENDIF
REM *** End program ***
WAIT KEY
END
```

4.

```
REM *** Read in an integer ***
INPUT "Enter a number : ",no1
REM *** IF an even number THEN Display
"Even" ***
IF no1 mod 2=0
  PRINT "Even"
ENDIF
REM *** End program ***
WAIT KEY
END
```

Activity 4.6

No, the PRINT statement is not executed.
The condition
    dice1 = 6 AND dice2 = 6
reduces to
    true AND false
which further reduces to
    false

Activity 4.7

a) false OR false = false
b) true OR false = true
c) false OR true = true
d) true OR true = true

Activity 4.8

```
REM *** Throw dice ***
RANDOMIZE TIMER ()
dice1 = RND(5)+1
dice2 = RND(5)+1
dice3 = RND(5)+1
REM *** IF at least two dice match THEN
display message ***
IF dice1 = dice2 OR dice1 = dice3 OR dice2
= dice3
  PRINT "You win"
ENDIF
REM *** Display dice values ***
PRINT "Dice 1 was ", dice1
PRINT "Dice 2 was ", dice2
PRINT "Dice 3 was ", dice3
REM *** End program ***
WAIT KEY
END
```

uess = number
guess > number
guess < number

To test a SELECT structure, then every value mentioned in every CASE option must be tested. Hence, the lines

```
INPUT "Enter a number " , num
SELECT num
    CASE 1, 3, 5, 7, 9
        PRINT "Odd"
    ENDCASE
    CASE 2,4,6,8,10
        PRINT "Even"
    ENDCASE
ENDSELECT
```

need to be tested using the values 1, 2, 3, 4, 5, 6, 7, 8, and 9. In addition, at least one test should specify a value not given in any of the CASE statements. This will check that the DEFAULT option is executed as expected (assuming there is a DEFAULT option), or that the whole SELECT structure is bypassed as expected.

Summary

- The term nested IF statements refers to the construct where one IF statement is placed within the structure of another IF statement.
- Multi-way selection can be achieved using either nested IF statements or the SELECT statement.
- The SELECT statement can be based on integer, real or string values.
- The CASE line can have any number of values, each separated by a comma.
- The CASE DEFAULT option is executed when the value being searched for matches none of those given in the CASE statements.
- Testing a simple IF statement should ensure that both true and false results are tested.
- Where a specific value is mentioned in a condition (as in no < 0) , that value should be part of the test data.
- When a condition contains AND or OR operators, every possible combination of results should be tested.
- Nested IF statements should be tested by ensuring that every possible path through the structure is executed by the combination of test data.
- SELECT structures should be tested by using every value specified in the CASE statements.
- SELECT should also be tested using a value that does not appear in any of the CASE statements.

Solutions

Activity 4.9

```
REM *** Read in word ***
INPUT “Enter a word : ”,word$
IF word$ = “yes” OR word$ = “YES”
  PRINT “Access granted”
ENDIF
REM *** End program ***
WAIT KEY
END
```

Activity 4.10

Substituting true and false we get:

```
(false OR true) AND (false OR true)
= true AND true
       =true
```

Activity 4.11

Substituting true and false we get:

```
  NOT (true AND true)
= NOT true
= false
```

Activity 4.12

```
IF no1 >= 0
  PRINT* “Positive number”
ELSE
  PRINT “Negative number”
ENDIF
```

Activity 4.13

```
REM *** Generate random value ***
RANDOMIZE TIMER ()
number = RND(99)+1
REM *** Guess the number ***
INPUT “Enter your guess (1 to 100) :
”,guess
REM *** IF the guess is correct THEN
Display

“Correct” ***
IF guess = number
  PRINT “Correct”
ELSE
  PRINT “Wrong”
ENDIF
REM *** Display both values
PRINT “Number was ”, number," Guess was
“,guess
REM *** End program ***
WAIT KEY
END
```

Activity 4.14

```
REM *** Read in two numbers ***
INPUT “Enter first number : ”, no1
INPUT “Enter second number : ”, no2
IF no1 < no2
  PRINT “Smallest number is ”, no1
ELSE
  PRINT “Smallest number is ”,no2
ENDIF
REM *** End program ***
WAIT KEY
END
```

Activity 4.15

```
REM *** Read in an integer ***
INPUT “Enter a number : ”,no1
REM *** IF even THEN Display “Even” ***
IF no1 mod 2 = 0
  PRINT “Even”
ELSE
  PRINT “Odd”
ENDIF
REM *** End program ***
WAIT KEY
END
```

Activity 4.16

```
IF no1 >= 0 THEN PRINT “Positive number”
ELSE PRINT “Negative number”
```

(this is entered in a single line)

Activity 4.17

1. A Boolean expression is an expression which reduces to either true or false.

2. Six (<, <=, >, >=, =, <>)

3. AND is always performed first unless the OR is enclosed in parentheses.

Activity 4.18

```
REM *** Generate random value ***
RANDOMIZE TIMER()
number = RND(99)+1
REM *** Guess the number ***
INPUT “Enter your guess (1 to 100) : ”,guess
REM *** Respond to guess ***
IF guess = number
  PRINT “Correct”
ELSE
  IF guess > number
     PRINT “Your guess is too high”
  ELSE
     PRINT “Your guess is too low”
  ENDIF
ENDIF
REM *** Display both values ***
PRINT “Number was ”, number," Guess was
“,guess
REM *** End program ***
WAIT KEY
END
```

Activity 4.19

```
REM *** Read in a number ***
INPUT “Enter number ”, no1
REM *** Display appropriate message ***
IF no1 > 0
  PRINT “Positive number”
ELSE
  IF no1 = 0
     PRINT “Zero”
  ELSE
     PRINT “Negative number”
  ENDIF
ENDIF
REM *** End program ***
WAIT KEY
END
```

Activity 4.20

No solution required.

Activity 4.21

```
REM *** Get number ***
INPUT “Enter item number (1 - 4) : ”, no
IF no = 1
  PRINT “A sword”
ELSE
  IF no = 2
     PRINT “A wand”
  ELSE
     IF no = 3
        PRINT “A bag of dragon’s teeth”
     ELSE
        IF no = 4
           PRINT “A water skin”
        ELSE
           PRINT “Unknown item”
        ENDIF
     ENDIF
  ENDIF
ENDIF
```

Activity 4.22

```
REM *** Get number ***
INPUT “Enter item number (1 - 4) : ”, no
REM *** Display appropriate message ***
SELECT no
  CASE 1
     PRINT “A sword”
  ENDCASE
  CASE 2
     PRINT “A wand”
  ENDCASE
  CASE 3
     PRINT “A bag of dragon’s teeth”
  ENDCASE
  CASE 4
     PRINT “A water skin”
  ENDCASE
  CASE DEFAULT
     PRINT “Unknown item”
  ENDCASE
ENDSELECT
REM *** End program ***
WAIT KEY
END
```

Activity 4.23

```
REM *** Get numberRead score
INPUT “Enter your score : ”, score
REM *** Display appropriate message ***
SELECT score / 100
  CASE 0
     PRINT “Pathetic”
  ENDCASE
  CASE 1
     PRINT “Beginner”
  ENDCASE
  CASE 2
     PRINT “Apprentice”
  ENDCASE
  CASE 3
     PRINT “Competent
  ENDCASE
  CASE 4
     PRINT “Master”
  ENDCASE
  CASE 5
     PRINT “Grand master”
  ENDCASE
  CASE DEFAULT
     PRINT “Invalid score”
  ENDCASE
ENDSELECT
REM *** End program ***
WAIT KEY
END
```

Activity 4.24

Yes. The condition no <= 0 is true and hence the PRINT statement is executed.