Updated 29 Oct 2011

Please note: This document is an unfinished work in progress and is only provided at this early stage to help newcomers to get started! It will be updated as time goes on.

The text formatting will be reworked/corrected when complete. The PDF file format for work-in-progress releases is only being used in order to preserve the text formatting while writing; other file formats should be possible!

Getting Started with Monkey

What is Monkey?

Monkey is a "games development environment" incorporating a text editor, an easily-learned programming language and a set of standard memorable commands for graphics, audio and more.

At the time of writing, Monkey allows you to create games that run on multiple computer systems such as:

- *Standard Windows PCs, including desktop PCs and laptops;*
- *Apple computers with Intel processors, such as iMacs and MacBooks;*
- *Web browsers, via the Flash and HTML5/Canvas standards;*
- *Mobile phones using the Android, Windows Phone 7 or iOS operating systems;*
- *Xbox 360, via the XNA platform.*

Who is this book for?

This book is aimed primarily at the absolute beginner. If you've never written a line of code in your life, or you've only dabbled in the past, Monkey will soon have you creating your own games, whether it's for your own amusement, to impress your friends or even for cold hard profit.

You don't need any special skills, though it helps if you're good at thinking your way around a problem. Contrary to popular belief, you don't even need to be particularly good at mathematics to write computer games – depending on the type of game, of course.

What do I need?

For simplicity, this tutorial assumes you are testing on a PC or Mac and using the HTML5 target. You will need an up-to-date web browser that supports HTML5 and its associated Canvas element. Any of the following web browsers will be adequate, but *please make sure you visit the relevant web site and download the latest version or you may not have HTML5/Canvas support*:

- *Internet Explorer;*
- *Mozilla Firefox;*
- *Google Chrome;*
- *Opera.*

Note that if you're using the demo version of Monkey, you'll only be able to write games for the HTML5 platform, and only for non-commercial purposes; it's otherwise fully functional.

You have to purchase Monkey in order to target the other platforms (which require some extra manual setup), or to make commercial gain from your HTML5 games. The really cool part about Monkey is that you can develop for the HTML5 platform and simply "rebuild" the same code for the other platforms when you're ready.

This tutorial assumes you're running a Windows PC, but the process of creating folders, saving files and running the Monk program editor is similar on other PC-type platforms, such as the Apple Mac.

Let's go!

Let's take a look at a very simple Monkey program; open up the Monk IDE (Integrated Development Environment) from the main Monkey folder. Depending on your computer settings and operating system, it should be listed as something like *monk* or *monk.exe*.

Run it as you would any other program and you'll be presented with the interface below:

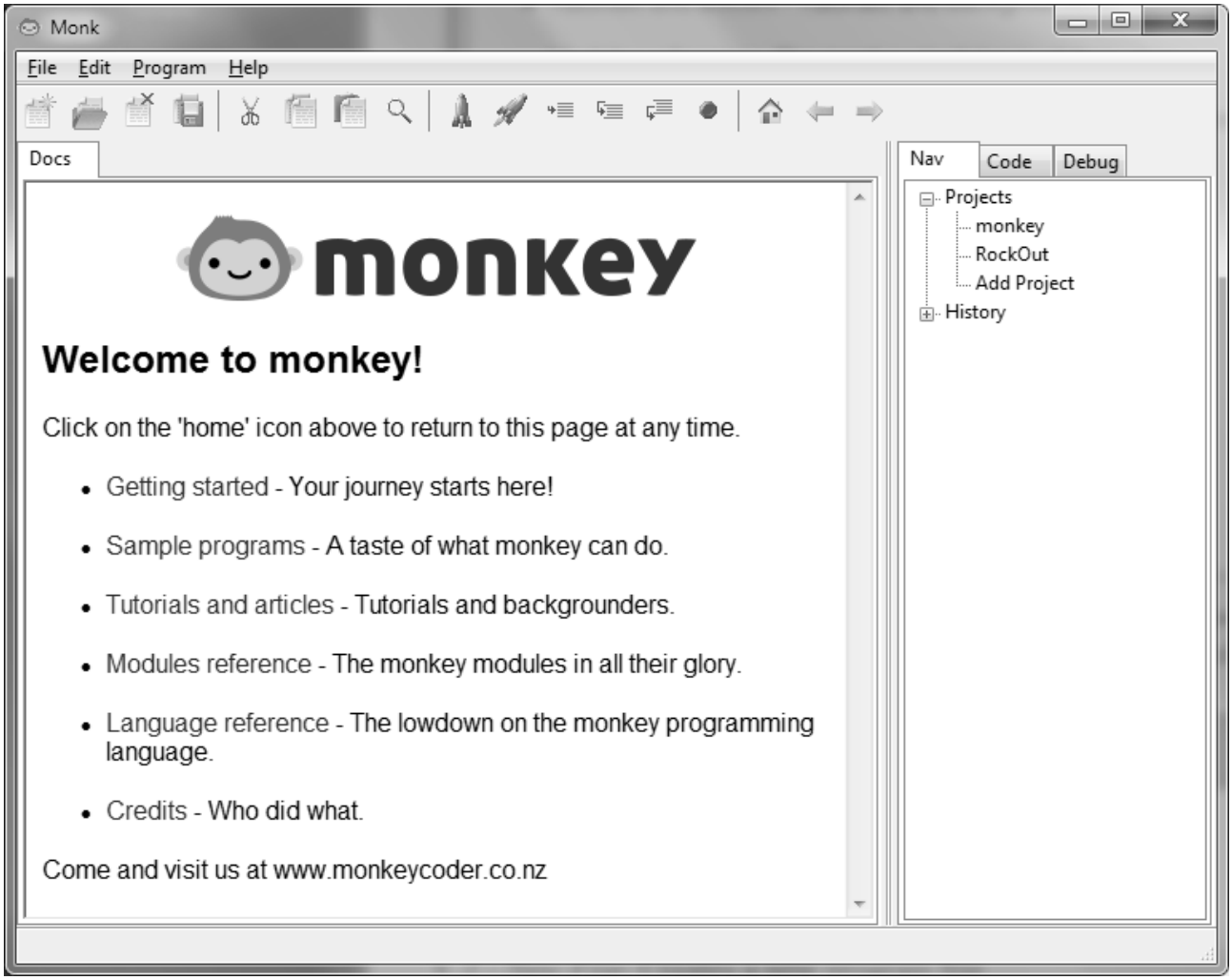

Figure 1: The Monk IDE

The icons at the top are arranged into groups, as below:

File operations

• *New file - Creates a new file*

• *Open file - Opens an existing file*

• *Close file - Closes the currently selected file*

• *Save file - Saves the currently selected file*

Clipboard/text operations

• *Cut - Cuts the selected text and places it on the clipboard*

• *Copy - Copies the selected text to the clipboard*

• *Paste - Paste text from the clipboard to the current editor location*

• *Find - Finds text within the currently selected file*

Compiler/debugger operations

• *Build - Compiles the program into a runnable program for the selected target platform*

• *Build and Run – The same as* Build*, but also runs the program if possible*

• *Step - Covered in later* Debugging *section*

• *Step In - Covered in later* Debugging *section*

• *Step Out - Covered in later* Debugging *section*

• *Stop – Terminate running program or stop build process.*

Help operations

• *Home - Show Monk home page*

• *Back - Go to previous document in history*

• *Forward - Go to next document in history*

All of these operations are accessible from the menus at the top if you prefer to use them.

Creating a new program

Click on the *New* icon to open a new blank tab within the IDE; you should have one called *Docs* and another called *untitled1.monkey* or similar. Each tab you open is effectively a separate text editor (similar to Notepad) that operates on an individual file.

You can switch between tabs (and therefore files) by clicking the named tabs at the top. Try clicking *Docs* to switch to the Monk home page, then click on *untitled1.monkey* to get back to where you were. It should look something like this:

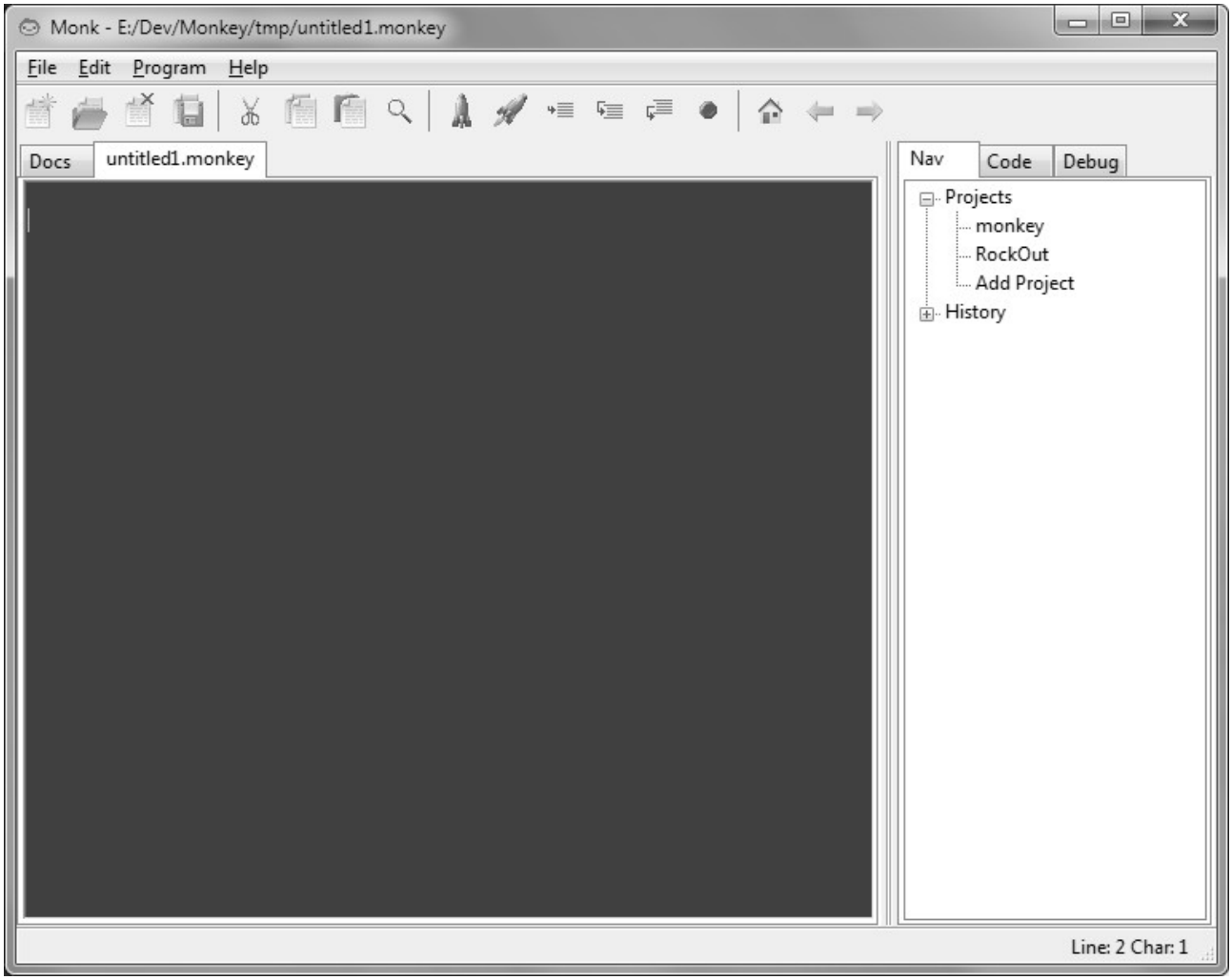

Figure 2: An untitled edit tab

The large blank area with the flashing edit cursor is where you type your program, and the icons and menus at the top affect the currently selected tab's contents in various ways.

The first thing to do is decide where you are going to save your Monkey programs. We'll create a folder on the desktop as an example: click using the right-most mouse button on a blank area of the Windows desktop. (If you can't see it, minimise all open windows so you can see your backdrop picture.) From the resulting pop-up menu, choose *New*, then *Folder*, then type in a name for the new folder: *Monkey Programs*, for example.

Now go to the *File* menu in the Monk IDE and choose *Save As*. In the *Save As* dialog box, navigate to your new folder through the *Desktop* link on the left-hand side, then click in the *File* name field at the bottom, deleting anything that may be in there already.

Type in a name for your program and add *.monkey* on the end (that's *dot* monkey). For example, you could type *invaders.monkey* or *flowerpower.monkey*; however, just type in *hello.monkey* for now. Once you've typed this in, click *Save* to return to the editor. The tab at the top should now show your new file name and you can simply click the *Save* icon to save changes to this file in future.

Creating a simple project: Hello, world!

We'll start with the traditional *Hello, world!* application that introduces almost every programming language in the world: click in the blank text area of your new tab and type in the program below:

```
Function Main ()

        Print “Hello, world!”

End
```

Type in the first line and press *Enter* (or *Return*) on your keyboard. Type the second line and press *Enter*. (To get the indentation (or offset) before the *Print* keyword, first press the *Tab* key on the left edge of the keyboard.) Finally, type the last line and press *Enter*. (You may find the last line is still indented, depending on your Monk settings; if so, move the cursor to the start of the line and delete the blank area.)

Monkey is “case-sensitive”, meaning you have to use capital letters in the right places, so make sure that what you type matches the above *exactly*.

Double-check that everything looks the same, editing it like any other text editor if not (using the cursors and the *Del*/*Backspace* keys where necessary), then finally click the *Build and Run* icon to bring up the *Monkey Build* window; it should all look something like this:

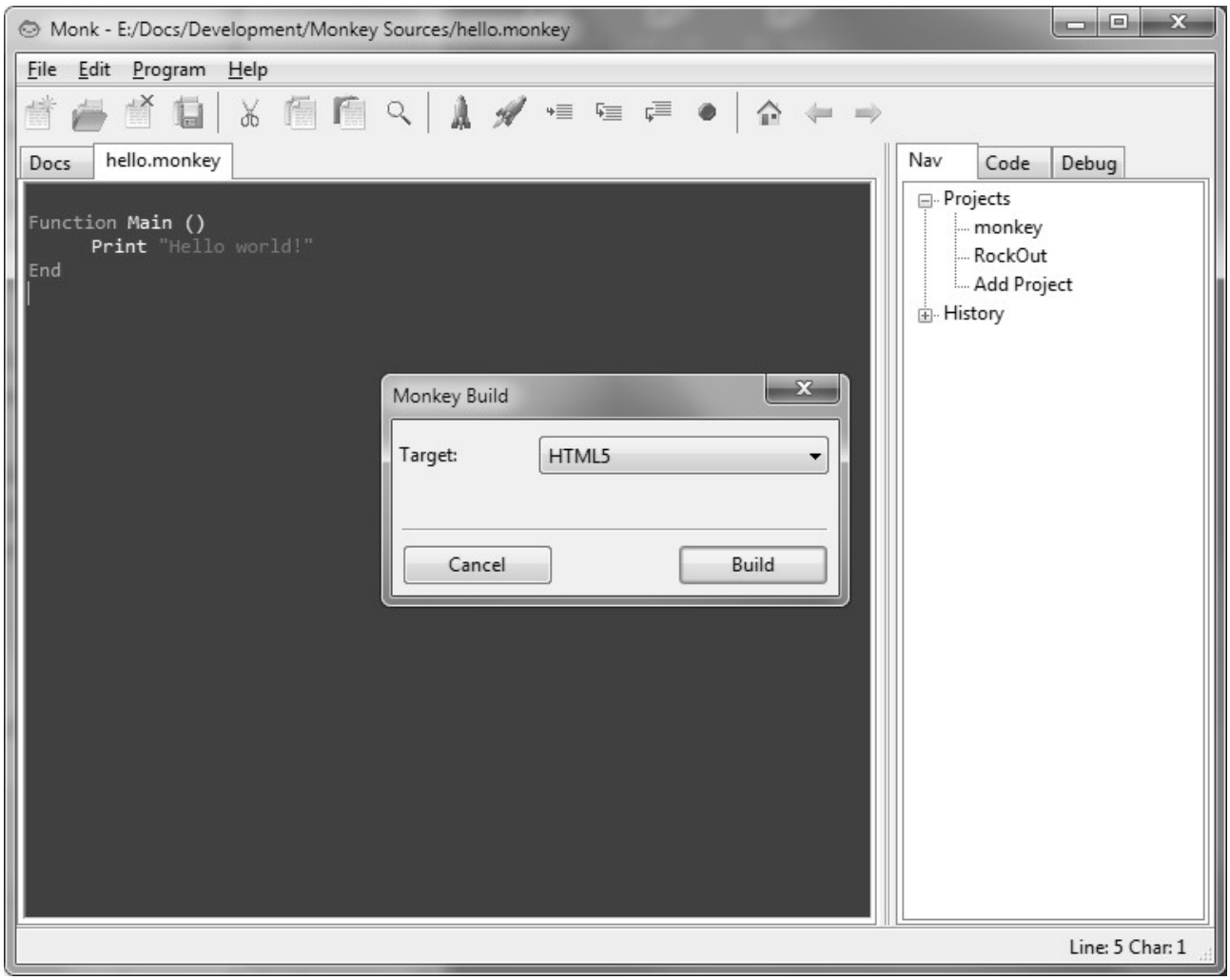

Figure 3: Building the Hello, world! program

Make sure that *HTML5* is the selected target and click *Build*. For HTML5 programs, Monkey will run a tiny program called *MServer* and open your default web browser. MServer will then 'feed' your program to the web browser, and you should see something like this:

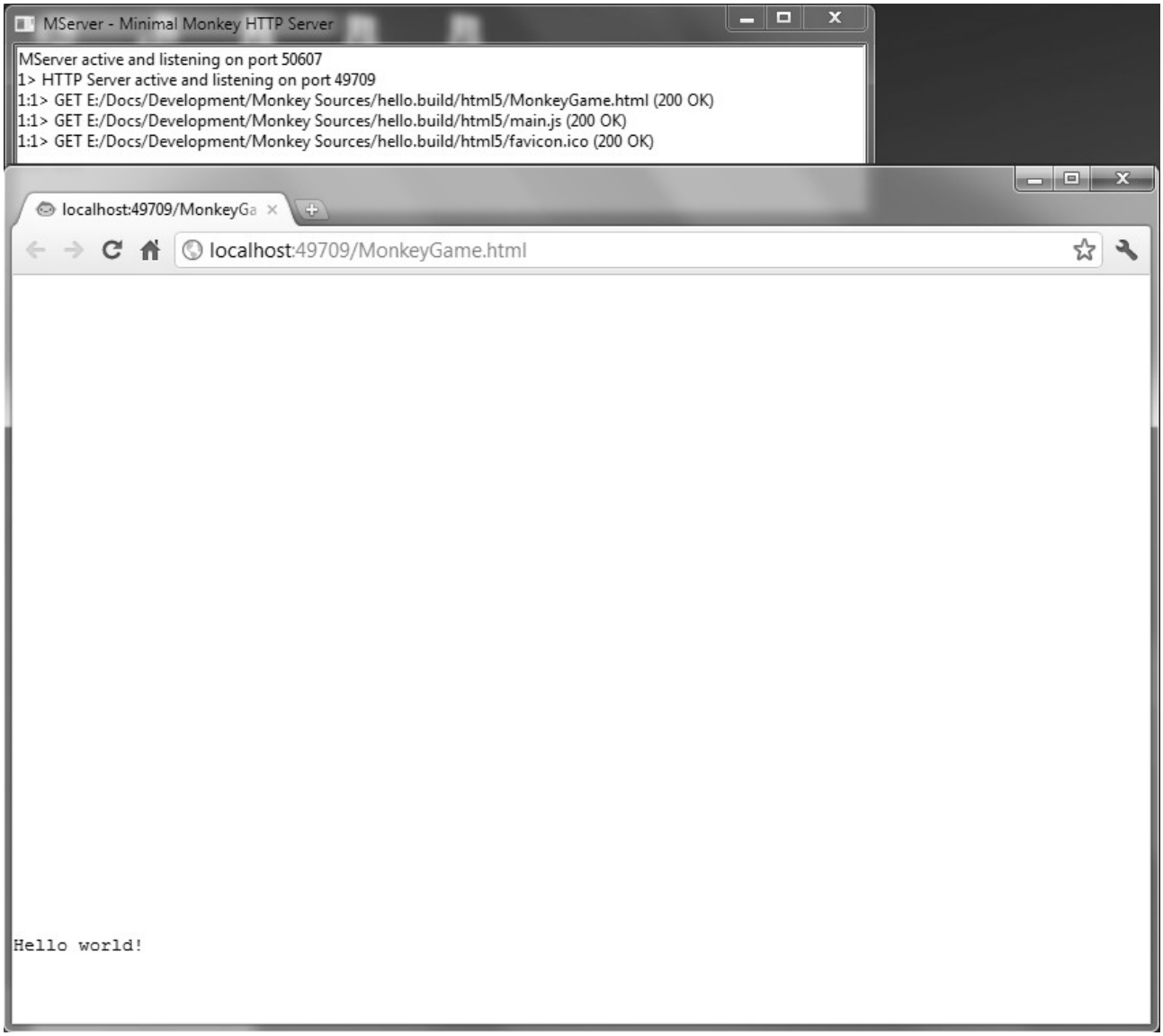

Figure 4: The Hello, world! program in all its glory

(The large blank area above the text is in fact the 'Canvas' where a game would normally appear.)

Now close the web browser and go back to Monk. Try changing the text between the quote marks (make sure you leave them in place) and run it again to see the difference; repeat until you've had enough.

When you're done, click the *Save* icon, then the *Close* icon: *Oh, no! Your precious work! All gone!* Not really; to get back to your previously saved work, run Monk, click the *Open* icon and choose *hello.monkey* from the *Open File* dialog.

Hello, world! explained

Let's have a look at that program again and try to understand what it means:

```
Function Main ()
	Print "Hello, world!"
End
```

The text that makes up a program like this is referred to as *source code*, often abbreviated simply as *code*. First of all, let's strip this down to the smallest Monkey program you can write:

```
Function Main ()
End
```

This is called the *Main function*; it's required by every Monkey program and it's also the starting point for every Monkey program. For now, consider a function to be a 'block' of code (that is, a bunch of related lines of code) to be executed; but we'll cover functions in much more detail later on.

The first line features the special keyword *Function*, followed by the name of the function (*Main* in this case) then a set of empty brackets. The second line, *End*, marks the end point of the function. The code we type in between these two lines will define what the function actually does. In the case of *Hello, world!*, the code to be executed is *Print "Hello, world!"*, which simply 'prints' the given text on the screen.

That's it!

Typing and indentation

Create a new file, as before, but give it a different name (still ending in *.monkey*), then type in these two lines:

```
Function Main ()
End
```

We're doing it this way to ensure right from the start that we have both the opening line and the matching closing line; we'll fill in the rest afterwards.

Now move the edit cursor to the end of the first line (after the brackets) and press *Enter* to insert a blank line between the two existing lines; it should look like this:

```
Function Main ()


End
```

Now, with the text cursor flashing at the start of the blank line, press the *Tab* key, which will shift the cursor to the right. This indents any following text; that is, it shifts everything to the right, as you can see below:

```
Function Main ()

        Print “Hello, world!”
        Print “Hello, solar system!”
        Print “Hello, galaxy!”
        Print “Hello, universe!”

End
```

Fill in a few *Print* lines as above, pressing *Enter* after each one.

Why use indentation?

The indentation is more important than you will probably appreciate at this point, so do make a point of using it right from the start. Indentation is not technically required in order for the program to work, but it really helps to make things readable later on; we can instantly see the indented block of code we've created 'inside' the Main function, and this becomes even more important when we start creating blocks of code inside other blocks of code.

When you build and run the above example, Monkey goes to the start of the Main function and simply executes each line in turn, in the order they are written. This order of execution is what we call *program flow*, and later we'll see how this flow can be altered in order to perform different actions depending on the situation.

Creating a project with external media

The previous examples only required the creation of a single file in order to run. However, games almost always need sound and visuals, and this requires a little extra setup.

Summary of creating a new project

If you're new to managing files and folders, you can follow the full process outlined below for any new project you create, but for those already familiar, the steps can be summarised as follows:

- create a new folder for your project, giving it a unique name of your choosing;
- decide on a 'base' name for your program, eg. *gamename*;
- create a Monkey source code file in the new folder and call it *gamename*.monkey;
- create a folder called *gamename*.data, making sure to use the same base name.

Note that Monkey source files are simple plain text files, so can be created and edited in any program capable of editing such files, e.g. Notepad.

If you're not too familiar with file and folder management, the process outlined below may seem rather involved, but this is mainly to make things as foolproof as possible while you get started; you'll soon pick it up.

We're going to start by creating a folder specifically for our new project, so first minimise all open windows so you can see your desktop background, and locate the folder you created earlier; it will be called *Monkey Programs* if you used the suggestion given. Open it up and you'll probably notice your test programs from earlier.

Right-click (that is, click with the right-most mouse button), on a blank area within this folder and choose *New* from the menu, then choose *Folder* from the next menu. Type in a name for the project folder; let's call it *FirstGame* for now. If you need to rename a file or folder at any point, eg. in case of a typo, right-click its icon and select *Rename* from the menu.

You should be seeing something like this, but don't worry if it doesn't look exactly the same:

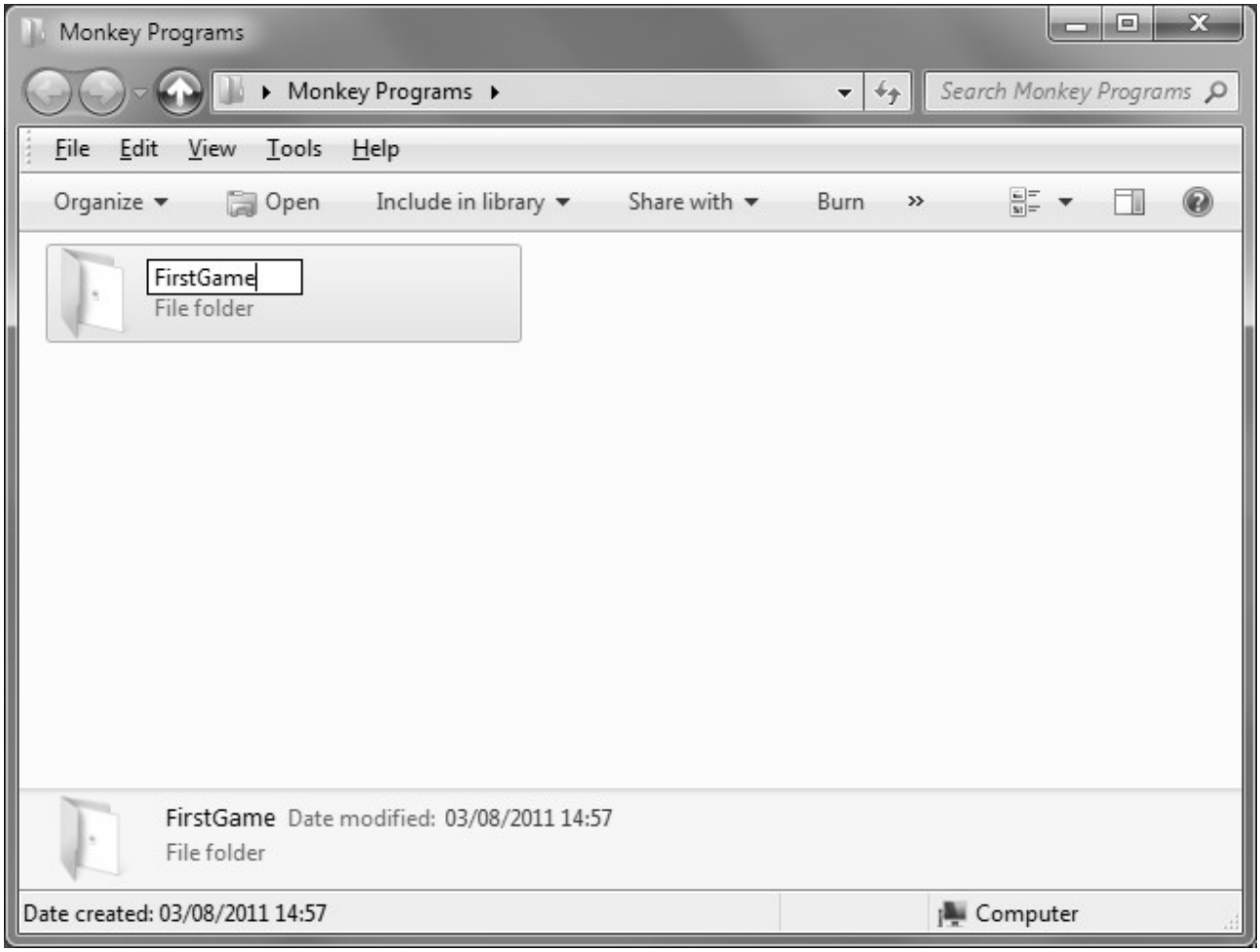

Figure 5: Creating a project folder

Now create a new file via the Monk IDE (as you did for the *Hello, world!* example) and save it into the *FirstGame* folder; call it *firstgame.monkey*. Go back to the desktop, navigate into the folder you just created and you should see that the file has appeared here, as below:

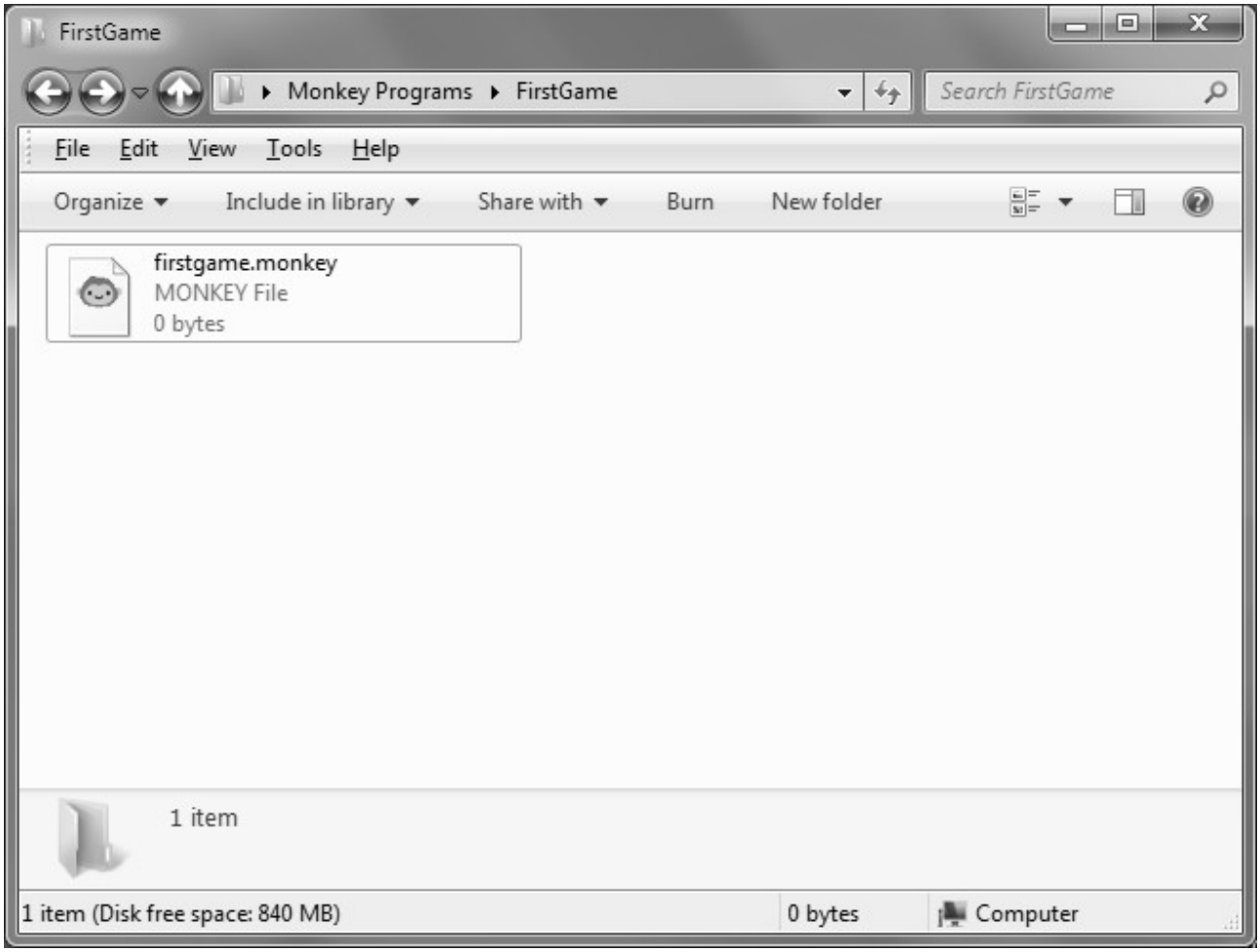

Figure 6: The project's main source file

We've now created a project folder called *FirstGame* and a program file called *firstgame.monkey* within that project folder.

Lastly, we need to create a location for our graphics and any other external media (sounds, for example). Make sure you're looking at the folder containing *firstgame.monkey* (as in the above image) and right-click on a blank area to create another folder here. Name this folder *firstgame.data*.

Note carefully that the folder should take the same name as the program file, replacing *.monkey* with *.data* (*dot-data*); this is very important! The end result should look something like this:

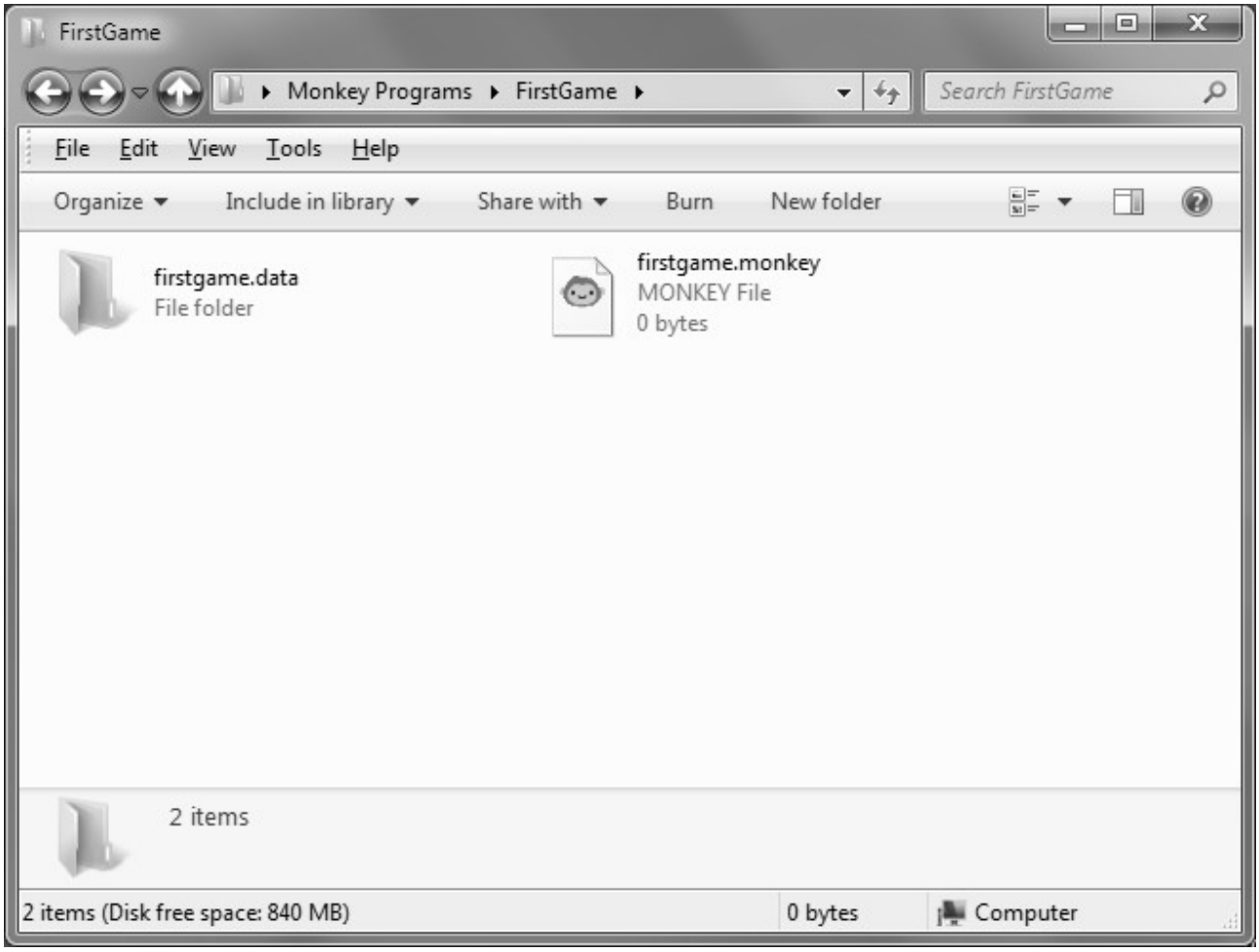

Figure 7: The project's main source file and associated media folder

Don't worry if it doesn't look exactly the same. Just make sure you have a program file and a folder with the same 'base' name (*firstgame* in this case) sitting next to each other, and that the folder's name ends with *.data*.

If your program file appears to be missing the *.monkey* extension in this view, do the following to make it appear:

- click on the *Tools* menu at the top and choose *Folder Options*. If you don't have the *Tools* menu, click on *Organize* instead, then *Folder and search options*;
- in the new window, click on the *View* tab;
- un-check the option marked *Hide extensions for known file types*;
- click *OK* at the bottom.

Being able to see file extensions like this is a very useful option for programmers, as you'll see when you come to using media files.

We're going to need a small image to put on the screen within our game, so download the image located at the address below and place a copy in the *firstgame.data* folder:

```
http://www.hi-toro.com/boing.png
```

Type this address into your web browser and press *Enter*, then right-click the image and save it. In the event that the web site is unavailable, you can use any other image of a similar size (256 x 256 pixels), as long as it is in PNG format (ie. its name ends with *.png*). Make sure that the file you place in the *firstgame.data* folder has the name *boing.png*; if not, right-click the file and choose *Rename* to change it.

Return to Monk and open or select the *firstgame.monkey* file if it isn't already selected, then type in the code below, without worrying too much about how it works:

```
Import mojo

Function Main ()
	New Game
End

Class Game Extends App

	Field player:Image
	Field x:Float
	Field y:Float

	Method OnCreate ()
		player = LoadImage ("boing.png")
		SetUpdateRate 60
	End

	Method OnUpdate ()

		If KeyDown (KEY_LEFT) Then x = x - 4
		If KeyDown (KEY_RIGHT) Then x = x + 4
		If KeyDown (KEY_UP) Then y = y - 4
		If KeyDown (KEY_DOWN) Then y = y + 4

	End

	Method OnRender ()

		Cls 64, 96, 128
		DrawImage player, x, y

	End

End
```

Build and run this program (checking carefully for any typing errors if it doesn't run), and your web browser should open up and the game appear within. You should then be able to control the movement of the 'player' using the up/down/left/right cursor keys. When you're done, close the browser window.

We won't go into too much depth at this point, but we can take a look at the structure and layout of this program in brief. It starts with:

```
Import mojo
```

This special line makes it possible for us to use Monkey's *mojo* module, which is a set of commands that give us the ability to load and display graphics, play sounds, process mouse/keyboard input and so on.

We then have a Main function, which you should hopefully recognise by now. (Don't worry about the meaning of the code within Main, but do remember that the program flow always starts here.)

Following this is a block of code called a 'class' (named *Game* here), which itself contains several blocks of code. Note the three blocks of code that begin with *Method* and close with *End*, and how the code within these blocks is *further* indented to the right. Again, this makes the identification of separate blocks, or 'chunks', of code much easier than if every line were to be aligned to the same left offset.

Also note that the indentation makes it possible to see that the *Class* at the start of the 'class' block matches the final *End* in the program; so we have the outer class block, which then contains three indented 'method' blocks, which each contain their own indented blocks of code. Again, don't worry about what it all means at this point, but see if you can identify these indented blocks of code for yourself.

We also have some blank lines; again, these are optional, and you can have as many blank lines as you like, and they help to further separate different blocks of code for enhanced readability. For comparison, here's the same program without any indentation or spacing:

```
Import mojo
Function Main ()
New Game
End
Class Game Extends App
Field player:Image
Field x:Float
Field y:Float
Method OnCreate ()
player = LoadImage ("boing.png")
SetUpdateRate 60
End
Method OnUpdate ()
If KeyDown (KEY_LEFT) Then x = x - 4
If KeyDown (KEY_RIGHT) Then x = x + 4
If KeyDown (KEY_UP) Then y = y - 4
If KeyDown (KEY_DOWN) Then y = y + 4
End
Method OnRender ()
Cls 64, 96, 128
DrawImage player, x, y
End
End
```

This is just a mess, and it's much harder at a glance to locate the different blocks of code that form the program. In a program containing hundreds or even thousands of lines, this really is a no-go!

Start using indentation and spacing from the very beginning. If you choose to ignore this advice, do be prepared for experienced programmers to complain when you're posting code online for help! You should make it as easy as possible for others to understand your code.

Comments

Finally, we'll add some *comments* to this code. Comments allow us to make notes within the program, which can be used as a reminder as to how a particular piece of code works, or as an explanation to others that we

might share the code with. Comments are denoted using the ' symbol (an apostrophe), and Monkey simply ignores everything that follows this symbol. You can add a comment to the end of an existing line of code or dedicate a whole line to a comment. See if you can locate the comments here:

```
Import mojo ' This gives us graphics, etc!

' This is where the game begins...

Function Main ()
	New Game
End

' Here's the Game class:

Class Game Extends App

	Field player:Image ' Player image
	Field x:Float ' Player's “across” position
	Field y:Float ' Player's “down” position

	Method OnCreate ()
		player = LoadImage ("boing.png")
		SetUpdateRate 60 ' Sixty frames per second!
	End

	Method OnUpdate ()

		' This is where keyboard input is processed
		' and applied to the player's position...

		If KeyDown (KEY_LEFT) Then x = x - 4
		If KeyDown (KEY_RIGHT) Then x = x + 4
		If KeyDown (KEY_UP) Then y = y - 4
		If KeyDown (KEY_DOWN) Then y = y + 4

	End

	Method OnRender ()

		' Here's where we clear the screen and draw the player!

		Cls 64, 96, 128 ' Each number here can be changed from 0-255!
		DrawImage player, x, y

	End

End
```

Comments are incredibly useful, as you *will* find yourself coming back to a program several months later and wondering how on earth a certain section of code works, no matter how well you understood it at the time!

Don't over-use comments, but do use them to explain anything that took you a while to figure out, and write them as if you're explaining things to someone who's never seen your code before.

Disabling code with comments

Comments can be used to disable code, by placing the comment symbol at the start of a line, like this:

```
Function Main ()

        Print “This will be printed...”

        ' Print “This won't!”

End
```

This allows us to “comment out” lines of code, as well as to “un-comment” (by *removing* the ' symbol) in order to reinstate commented-out code, which is very handy for testing purposes.

Block comments

Commenting out one or two lines of code is easy enough, but it you want to comment out a whole block of code, Monkey has a special 'compiler directive' (which gives special instructions to the code-reading part of Monkey), *#rem*, which can be used like this:

```
Function Main ()

        Print “This will be printed...”

        #rem

        Print “This won't be printed!”
        Print “Same here!”
        Print “And here!”

        #end

End
```

Note the special character # (the *hash* symbol, sometimes known as *pound*), which denotes a compiler directive. Anything placed between *#rem* and *#end* will be ignored, which is very handy for quickly disabling huge chunks of code. Just remove *#rem* and *#end* to re-enable the relevant block of code.

Experiment!

You might like to play around with the above program before moving on. Here are few changes you could make:

- change the image used. Copy another image of similar size into the *firstgame.data* folder. Ideally use an image in PNG format (ends with *.png*), but you can also use JPEG files (ending in *.jpg*) too. Make sure you change the file's name in the source code from *boing.png* to whatever your new image is called!
- change the speed at which the image moves. (Hint: the current speed is 4, as found in the block of code beginning with *Method OnUpdate*.)
- anything else you think might work! Don't worry if you 'break' the program – you can always type it in again if you really want to.

Basic programming concepts

Here's where we get serious! We're going to look at some of the core concepts of computer programming, such as *variables*, *program flow* (and the closely related area of *decision making*), then dabble with *object-oriented programming*, which allows for the development of more structured, manageable programs.

Variables

Computer programs need to store and retrieve information in order to function; for example, a game in which the player fights against crazed alien space robots may need to keep track of information such as:

- how many lives the player has left;
- how much ammo remains;
- shield levels;
- player position;
- positions of all crazed alien space robots;
- and many other possibilities.

Information like this is stored in *variables*. In Monkey (as in most programming languages) a variable is a text label – a word – chosen by the developer (that would be you!) and used within the source code of the game to represent a particular piece of information.

Let's say we want to store a person's age so we can make use of it throughout the game. We can use almost any word to represent this information, but it makes sense to use something relevant:

```
age = 25
```

Here we've created a variable called *age* and *assigned* it a value of 25. (You might read the above line as "age equals 25".) There's nothing to stop you storing someone's age in a variable called *sausage*, or *tractor*, or even *brontosaurus*, but those names aren't going to help you remember that they represent someone's age!

Variable naming rules

There are some basic rules regarding the names you can give to variables:

- only letters, numbers and the underscore symbol can be used in a variable name;
- the name must not contain any spaces. For example, *max health* would not be a valid variable name, but *max_health* (using the underscore symbol in place of a space) would be fine;
- a variable name cannot start with a number; the name must begin with either a letter or the underscore symbol. *12gunsound* is therefore invalid, but *_12gunsound* or *gunsound12* are fine;
- variable names are *case-sensitive*, that is, *Capitals Matter*! The variable *Age* is different to the variable *age*.

Declaring variables: an example

To *declare* a variable just means we're telling Monkey we'll be using this particular label to represent some information.

Here's an example you can type in and run (just click the *New File* icon in Monk to open a new editor tab). In this example, we're declaring the variables *age*, *energy* and *ammo*, and immediately assigning values to each one.

There are two details here that we haven't covered yet: note the use of the 'special' Monkey keyword *Local* before each variable name, followed by a colon and the suffix *Int* at the end of each variable name; however, don't worry about their meanings at this point:

```
Function Main ()

        ' Declare age, energy and ammo variables...

        Local age:Int = 25
        Local energy:Int = 100
        Local ammo:Int = 10

        Print age
        Print energy
        Print ammo

End
```

So, to declare a variable, we use the keyword *Local* and add :*Int* to the end of its name. Note that we only use the *Int* suffix to initially declare the variable; it's not needed after that.

If you run the above program you'll just see a list of numbers output on the screen, but you should see that each number matches the values we've specified above. We say that each variable *contains* a value, e.g. the variable *age* contains the value 25. (One way to visualise this is to imagine a small box marked 'age' containing a piece of paper marked 25.)

Try changing the contents of each variable (e.g. changing the value of *age* to 26) and checking the resulting output to confirm that what is printed by the program matches what you've typed in.

Create some variables of your own and add them to this program – remember to use *Local* and :*Int* as in the existing examples – and then add a *Print* line for each new variable. (Note: at this point you should only store whole numbers in your variables, or you'll run into problems, for reasons we'll discover soon!)

Print: an aside

We're using the *Print* command again here, as in the *Hello, world!* example, but what we're asking it to display isn't contained in quotes this time. We use quotes to print specific pieces of text, such as "Hello, world!", but when there are no quotes it means we're either printing the *contents* of a variable or a specific value, eg. *Print 100*.

Try putting double-quotes before and after the variable names to see the difference, e.g. *Print "age"*. This will be interpreted as plain text to be printed, and is nothing to do with the *age* variable.

Variable types

We're going to look at three basic *types* of variable. There are more, but we'll only need these three for now:

- *Integers* store whole numbers, such as 1, 50 and 999;
- *Floats* store fractions, such as 0.5, 33.333 and 1000.95;
- *Strings* store text, such as "Hello, world!", "I am a robot" and "Charlie Chaplin".

Each variable type has an associated *suffix*. In the previous example, the suffix was *Int*, which is used to declare integer variables, that is, variables used to store whole numbers.

In general, to declare any type of variable, you write the keyword *Local*, the name of the variable, and then add the relevant suffix.

You can optionally replace each suffix for these basic types with a shortcut symbol, as documented below; for instance *Local myfloat:Float* can be declared as *Local myfloat#* instead.

For the purposes of this guide, we'll use the longer versions for instant readability, but be aware that these shortcuts exist, as you'll find them in other people's code!

Basic variable types

Integers

Suffix: Int
Shortcut symbol: %
Range: −2,147,483,648 to 2,147,483,647
Example values: 0, 100, 5000, -99
Example declarations: *Local cats:Int*
Local cats%

For our purposes, *integers* are simply whole numbers. When you create an integer variable, you can assign any whole number to it, within the range specified above – not many games will need numbers outside of this range! (However, there are other ways to deal with this if necessary.)

If you don't specify a value for an integer variable when you declare it, Monkey will automatically assign it a value of zero. You can see this by running the demo below:

```
Function Main ()

	Local bullets:Int

	Print bullets
```

```
End
```

Running this demo will simply print 0 (zero), showing that the variable *bullets* has been automatically assigned a default value. Now let's specify an initial value; we do this by adding an *equals* sign followed by a whole number:

```
Function Main ()

	Local bullets:Int = 10

	Print bullets

End
```

Roughly translated to natural language, the first line inside the Main function might be read as: “bullets equals ten.”

Try changing the value, which may be positive (more than zero), negative (less than zero) or even zero itself, then run the program again. Remember, you can use any number from *minus two thousand million* all the way up to *two thousand million*.

Floats

Suffix: Float
Shortcut symbol: #
Range: see text
Example values: 0.1, 0.75, 2000.075, -100.55
Example declarations: *Local weight:Float*
Local weight#

Simply put, floats are numbers containing a decimal point – numbers that *aren't* whole, such as a half (0.5), a quarter (0.25), and so on.

Why “float”?

The word *float* in this context comes from the expression *floating point*. You may be surprised to learn that your computer's processor (it's "brain") only knows how to handle whole numbers – really! *Floating point representation* is the technical term used to describe a way of approximately representing fractions within a system that can only deal with whole numbers.

Your computer's processor has many fast built-in commands, or *functions*, for dealing with floating point numbers – adding, subtracting, dividing, and so on – and these built-in functions are what Monkey makes use of, like most programming languages.

The approximate nature of floating point representation means that floats are not accurate enough for strict scientific or mathematical usage, but certainly good enough for most games, as the errors are tiny.

(Scientific and advanced mathematics programs ignore the processor's built-in functions and work things out the long way, making them slower but more accurate. The Windows Calc program is one such example.)

Here's an example of creating a floating point variable, or float, to store a fraction:

```
Function Main ()

	Local distance:Float

	Print distance

End
```

Note the use of the suffix *Float* instead of *Int* here. As with integers, if you don't assign a value when declaring a float variable, it'll be assigned a default value of zero.

Here's another demonstration, this time assigning a value to each variable as we declare it:

```
Function Main ()

	Local distance:Float = 5.25
	Local time:Float = 10.0

	Print distance
	Print time

End
```

Easy enough; floats are declared and assigned the same way as integers, just using the *Float* suffix instead.

As an aside, here's why floats are needed to store fractions:

```
Function Main ()

	Local distance:Int = 5.25

	Print distance

End
```

If you run this, the output is 5. As a fraction can't be stored in an integer variable like this, the fractional part is simply chopped off!

Monkey needs to tell the computer's processor what kind of variable it's being asked to work on, so that the processor can use the relevant built-in functions to deal with it. If we tell the processor via the *Int* keyword that we intend to store a whole number, we can't then give it a fraction to work with and expect it to get things right; if we do, then the processor simply rounds down the value to a whole number that can be stored as an integer. Change *Int* to *Float* in the above example and it'll work correctly.

Which type do I use, then?

As you can see, you have to decide right from the start whether a variable will be an integer or a float. This may seem limiting, but it turns out that you can almost always decide in advance what you'll need, just by taking the time to think.

Take the number of bullets a player has available, for example: are you going to allow the player to shoot half-bullets? Of course not; if the player starts with 10 bullets and fires a shot, he then has 9 bullets – it's always a whole number.

But what about a program that deals with cake mixtures, for example? Do you want a whole kilogram bag of sugar in every mix, or only a part of that amount? If your program uses a variable to represent sugar in kilograms, you definitely only want part of that amount, so you use a float: one tablespoon of sugar is about 0.02 kg, for example, so you might use something like *Local sugar:Float = 0.02*. An integer *sugar* variable would only allow you to use whole bags of sugar in each cake!

With that said, though, there's no reason you can't go back and change a variable's type if you find you need to, but there can be unexpected knock-on effects, so it's best to try and decide in advance.

Floating point range and accuracy

It's hard to state the range of values that a floating point variable can hold, due to the way floating point representation works. The maximum values (both positive and negative) are greatly reduced depending on how many figures you place before and after the decimal point; the more figures you place *before* the decimal point, the less accurate the numbers *after* the decimal point become, and vice-versa.

It's a very tricky subject to explore in detail, but in practise, for most game uses, you won't really need to worry about it, and can assume a positive and negative range in the thousands of millions.

One thing you should be aware of, though, is that floating point numbers, by their very nature, are inaccurate. Try running this program:

```
Function Main ()

	Local test:Float = 123456789.987654321

	Print test

End
```

The output, when run on the HTML5 target, may look slightly different, and may even vary on different web browsers. (For me, it's 123456789.98765433, demonstrating some rounding at the end.)

When you move on to other target platforms, the results will vary even more; in fact, some platforms won't even be able to represent simple numbers like 0.1 properly! This isn't an error on Monkey's part, or on the part of the target platform, but is again due to the way floating point numbers are stored within different computer systems; they are intentionally inaccurate for speed reasons since computers can only 'understand' whole numbers.

As a simple analogy, in mathematics, a third (1 divided by 3) is represented in decimal notation as:

```
  .
0.3
```

The bold dot above the 3 indicates that the 3 really repeats forever; this would be read as “zero point three recurring” and it means that there should be an infinite number of 3s on the end!

We're never going to type an infinite number of 3s so we have a symbol to represent this concept. Without the symbol, we'd have to decide how far we're willing to go in representing a third:

```
0.3                     ' Not accurate!

0.33
0.333
0.3333

0.333333333333333       ' Still not accurate!
```

At some point, we decide to give up; and as computers have no concept of infinitely recurring decimal places, they also have a limit on how accurately they represent decimal numbers.

You'll find that this usually doesn't matter – what difference does it make if your cake contains 0.021 kg of sugar rather than 0.022? Most things we'll use floats for in games simply don't require perfect accuracy, so don't worry about this; however, it's good to be aware of it.

Strings

Suffix: String
Shortcut symbol: $
Example values: "Hello world!", "The quick brown fox jumps over the lazy dog", "a"
Example declarations: *Local name:String*
Local name$

String variables are used to store text; they can store words, individual letters, punctuation symbols and numbers. (Numbers contained within strings will be treated just as if they were letters; that is, no mathematical operations will be carried out on them).

We've already used strings; we just haven't stored them in variables so far. The example *Print ”Hello, world!”* contains a string, *”Hello, world!”*, in the same way that *Print 10* contains an integer (10) and *Print 0.5* contains a float (0.5).

To store a string in a variable, we do the same as for the other variable types: use *Local* and the relevant suffix, *:String*, to declare it. Here's an example:

```
Function Main ()

        Local name:String = "Billy Bob"
        Print name

End
```

Notice that the string itself is contained within double-quotes. This is important.

If we didn't assign a string to the *name* variable here, it would create a default empty string. You can also specify an empty string manually, simply by writing the two double quotes with nothing in between, as in *Local name:String* = "".

The quotes aren't actually part of the string. They're only there to tell Monkey where the string starts and ends within your code.

You can join strings together with the **+** (*plus*) symbol, so we can create some more interesting output:

```
Function Main ()

	Local name:String = "Billy Bob"

	Print "My name is " + name

End
```

Here, we've joined a hard-coded string (*"My name is "*) with the contents of our *name* variable.

You could assign the *"My name is "* part to a string variable too:

```
Function Main ()


	Local intro:String = "My name is"
	Local name:String = "Billy Bob"

	Print intro + name

End
```

If you run, this, the output will be *"My name isBilly Bob"*, which is not quite right – there's no space between *is* and *Billy*! When joining strings to form sentences, you have to keep an eye out for this.

You could correct this simply by adding a space to the end of the *intro* string, between the word *is* and the closing quote.

Another way to handle it is to use the string joining capabilities of Monkey and insert a hard-coded string, containing just a space, between *intro* and *name*, like so:

```
Function Main ()


	Local intro:String = "My name is"
	Local name:String = "Billy Bob"

	Print intro + " " + name

End
```

Try changing the introductory text and the name. You might want to get adventurous and declare an extra

string to be printed after *name*, so it reads, for example, "My name is Billy Bob and I like rainbows." (Use the + symbol to add the extra string to the *Print* line.)

If you want to get more adventurous still, add yet another string and modify the *Print* line so you can change what Billy Bob likes.

What if you wanted to print this combination of strings over and over without having to type so much? You can just combine the strings.

The example below creates an empty string (*combined:String*) – it's empty because we haven't assigned anything to it while declaring it – then assigns the combined *intro* and *name* strings to it, like so, using the equals sign:

```
Function Main ()


        Local intro:String = "My name is "
        Local name:String = "Billy Bob"
        Local combined:String

        combined = intro + name

        Print combined

End
```

So, this effectively says *combined* equals *intro* plus *name*, or, expanded, *combined* equals "*My name is* " plus "*Billy Bob*". This is ultimately the same as typing *combined = "My name is Billy Bob"*, so when we then print the *combined* string, the output is *My name is Billy Bob*.

A string's contents can be changed outright by simply assigning it a new value:

```
Function Main ()


        Local intro:String = "Hello, world!"

        Print intro

        intro = "Goodbye, world!"

        Print intro

End
```

Although *intro* is initially declared as "*Hello, world!*", we then assign it the value "*Goodbye, world!*", changing its contents like so:

intro = "Hello, world!"
intro = "Goodbye, world!"

Hopefully you'll find strings are fairly straightforward. In summary, you just place your text between double-quotes and assign it to a variable, declared with the *String* type. You can also join them together with the **+** symbol.

Type conversions

Monkey can do some clever work "behind the scenes" with differing types, converting between types on the fly as necessary; for instance, when you try to assign a float value to an integer variable, the float is converted to an integer by chopping off everything after the decimal point, *then* it's assigned to the integer variable:

```
Function Main ()
	Local sugar:Int = 1.5
	Print sugar
End
```

This will print 1, as Monkey has assumed you want to convert this float value to an integer during the assignment.

You can also convert from floats and integers to strings:

```
Function Main ()
	Local flour:Float = 0.1
	Local flour_power:String = flour
	Print flour_power
End
```

At the point where we try to assign the float variable *flour* to the string variable *flour_power*, the float value is read by Monkey, turned into a string behind the scenes, *then* assigned to the *flour_power* string.

This is known as *implicit type conversion*; it's *implied* that you want it to happen. (Assigning a float to a string, for example, implies you want its value to be treated as text.)

You can also *explicitly* convert between certain types. For instance, Monkey doesn't allow you to directly convert the other way, e.g. from a string to an integer or float value, because strings can potentially contain any old text, while integers and floats can only store numbers; however, if you *know* you have a string containing "0.5", for example, you can explicitly convert it to a number like so:

```
Function Main ()
	Local half:String = "0.5"
	Local converted:Float = Float (half)
	Print converted
End
```

We want to assign the string value "0.5" to the float variable *converted*, so when we do the assignment, we state the variable type we want to convert *to* (*Float* in this case), followed by the value to be converted (that's the *half* string variable containing the text "0.5"), in brackets.

Explicit conversion like this is also known as *casting* from one type to another.

(Try adding changing the *half* string above to *"0.5 and some letters"* to see what happens, and see what happens if you use just plain text with no numbers.)

Lastly, here's a type conversion in the middle of a string assignment:

```
Function Main ()

	Local flour:Float = 0.1

	Local instruction:String = "Add " + flour + " kg of flour"

	Print instruction

End
```

There are a few things going on here:

- we declare a float variable, *flour*, and assign it a value of 0.1;
- we declare a string variable, *instruction*, and assign it a "hard-coded" string value, "Add ";
- on the same line, we add the *flour* variable, but as it's being added to a string, Monkey looks at the value of *flour* and converts it into a string, "0.5", then adds it on to the first part;
- again on the same line, we finally add " kg of flour" to the string.

The result is that the *instruction* string contains "Add 0.1 kg of flour".

Note that converting from one type to another like this does *not* change the type of the variable being converted; it only affects *how the variable is interpreted by Monkey* while being assigned or evaluated. In the above example, *flour* is still a float variable containing the numerical value 0.5 at the end.

Literals

Just for reference, when the values of strings and numbers are stated directly in a program, they're called *literals*: string literals, integer literals and float literals:

```
Function Main ()

	Local sticks:Int = 3
	Local stones:Int = 10
	Local sand_buckets:Float = 0.5
	Local insult:String = "Hey, get a load of this!"

End
```

They are often referred to as *hard-coded* strings or values, as you have manually specified their contents in the code, rather than asking the player to enter these values when the program is running, for example, or reading this information from a file on disk.

Variable assignment

Variable assignment is something we've already done a good number of times now; for example:

```
intro = "Hello world"
apples = 100
```

We're taking a string variable, *intro*, and *assigning* it a value, *"Hello world"*; or, put another way, we're assigning the value *100* to the variable *apple*, as in the second example.

Variable assignment can also include performing a *calculation* in place of the value to be assigned:

```
sausages = 100 + 50
```

Here, we're assigning the calculation *100 + 50* to variable *sausages*, that is, *sausages* will now contain the value *150*.

When we perform an assignment like this, Monkey first looks at the part *after* the equals sign, performs any calculations and/or type conversions, *then* places the result into the variable before the equals sign:

```
result = calculation
```

So, don't think of *result = calculation* as "result equals calculation", as this implies that the two are already the same; instead think of it as "*make* the result equal to the calculation".

Here are some examples, with the *result* variable before the equals sign and the *calculation* after it:

result	***= calculation***
bullets	*= 100*
bananas	*= 1 + 9*
energy	*= 50 + 25 + 10*

Translated for readability:

make *result*	**equal *calculation***
make *bullets*	equal *100*
make *bananas*	equal *1 + 9*
make *energy*	equal *50 + 25 + 10*

Here are some practical, though random, examples of variable assignment. Don't run this, as it won't print

anything! (You could add a few Print statements yourself if you want, though.)

```
Function Main ()

	Local apples:Int
	Local oranges:Int

	apples = 5
	oranges = 10 + 20

	Local fruit:Int

	fruit = apples + oranges

	Local produce:Int = fruit + 100

End
```

There are a few things here; in order:

- we create two integer variables, *apples* and *oranges*; we haven't assigned any values to them, so they will contain 0 (*zero*) by default;
- we then assign the value 5 to *apples*;
- we assign the result of *10 + 20* (that is, 30), to *oranges*;
- we create a new variable *fruit*;
- we assign the calculation *apples plus oranges* to *fruit* (total: 35);
- we create a variable *produce* and assign it the calculation *fruit + 100* (total: 135).

You could have declared *produce* and *fruit* at the top with the other variables and assigned values to them later, but sometimes it's just more convenient to do it as you go along. However, it *does* make for better code layout if you declare a bunch of variables together *before* you assign any values to them, as it makes it easier to locate and edit things when your programs become more complex; but code such as the above is still valid (and oh-so-handy!).

Constants

Constants work in a similar way to variables in that they have a given type and they store values, but constants are used to store fixed values; that is, *the values of constants cannot be changed while the program is running*.

Constants are declared in much the same way as variables, except that we use the *Const* keyword to declare them, rather than *Local*; also, their values must be set at the same time as they are declared.

Here's what they look like in practise:

```
Const PISTOL:Int = 1
Const MACHINE_GUN:Int = 2
Const LASER_BLASTER:Int = 3
Const THERMONUCLEAR_CANNON:Int = 4
```

All three basic types – integer, float and string – are supported:

```
Const MAX_LIVES:Int = 3
Const ROCKET_ACCELERATION:Float = 2.3
Const GAME_NAME:String = "Revenge of the Exploding Ninjas"
```

Note also the convention of using *ALL_CAPITALS* for constant names in Monkey; this allows us to instantly tell them apart from ordinary variables. This convention is not mandatory, so you can still name constants any way you like, but it is recommended.

Constants can always be replaced with variables, but they are useful for setting certain values in stone: say you're working with other people on a project; once the constant *LASER_BLASTER* has been defined as 3, any attempt by another programmer to change the value of *LASER_BLASTER* (elsewhere in the code) will cause an error when they try to build the project.

If it's important that this value is never changed throughout the project, making it a constant enforces this design decision. (It also stops *you* from accidentally trying to change a value you didn't mean to change!)

Here's a real example that shows why declaring a value as a constant may be very important: Monkey defines the mathematical value *pi* as a floating point constant in its *monkey.math* module (the code which gives Monkey its mathematical capabilities):

```
Const PI#=3.14159265
```

Note that *PI* has been defined here as a constant (remember, the # symbol is a shortcut for *:Float*).

You may remember from school mathematics classes that *pi* is commonly used to define the properties of a circle ("*circumference = 2 x* pi *x radius*"); it's extremely important that this value doesn't change.

(In reality, *pi* is an infinitely long number, so you might also read the definition of *PI* as a clear example of floating-point inaccuracy.)

Now imagine if *PI* had been defined by Monkey as a variable, and at some point you decide to create a variable representing, say, a *Power Increase* when the player collects a power-up. If you declared that variable using its initials, as *pi* (lower case), but accidentally typed it in upper case at some point (remember, variables are case-sensitive, so *pi* and *PI* are not the same variable), you could end up with a situation like this:

```
Function Main ()

	Local pi:Int ' Power Increase!

	PI = 100
	power = power + pi

End
```

If it wasn't a constant, the value of *PI* would be changed in this example from 3.14159265 – a very important

value – to 100. (A lesser concern here is that the lower case variable *pi* has a value of zero at this point, meaning *power* would in turn contain an expected value.)

Any attempt to perform mathematical operations involving *PI* would be spectacularly wrong from here on! However, because *PI* is in reality defined as a constant, Monkey will quite rightly complain if you try to run the above code.

There's another aspect relating to variables and constants that we need to cover later, called *variable scope*, but for now, you should generally declare constants 'outside' of the *Main* function, like so:

```
Const TONNE_IN_KG:Float = 1000.0

Function Main ()
        Print “One tonne equals “ + TONNE_IN_KG + “ kilograms”
End
```

(A metric tonne is 1000 kilograms; and if you don't do metric, a tonne is... *”about a ton”*!)

The value of a tonne should clearly never be changed, so although it could easily be defined as a variable within the *Main* function, we're declaring it as *Const* to ensure that it can never be changed.

In summary, you can 'read' constant values, as in the above example, but you can't 'write' to them (i.e. change their values).

Right, with that out of the way, let's find out how to do useful things with variables!

Mathematical operations

You can perform mathematical operations, such as addition, multiplication, division, and so on, with numerical variables:

```
Function Main ()


        Local x:Int = 10
        Local y:Int = 2

        Print x + y

End
```

The *Print* command is clever, in that it can convert most variable types (and the results of operations like this) into strings and display them; in this case, it will print *12*.

Change the plus sign above to a minus and the output will of course be *8*, that is, *10 minus 2*. Easy enough!

You can do this with hard-coded numbers as well, of course:

```
Function Main ()
```

```
        Print 1 + 2 + 3 - 1

End
```

The result is 5, as you can no doubt work out on your own. How about assigning all of these values to one variable before printing the result?

```
Function Main ()

        Local a:Int = 1
        Local b:Int = 2
        Local c:Int = 3

        Local result:Int

        result = a + b + c - 1

        Print result

End
```

Here we've declared an integer, *result*, and assigned the combined values of all the other variables to it, then subtracted 1. The *result* variable in the end contains the value 5, the sum of all the other variables minus 1.

This means you can perform a calculation once and simply use the same result over and over again. (Add a few more *Print result* lines to see what this really means – you can calculate once and then print the same result as many times as you like, or perform further calculations using this "pre-calculated" result, meaning you place less of a workload on your computer.)

Mathematical operations available in Monkey include addition, subtraction, multiplication, division, exponents (numbers raised *to the power of* another number), and all are represented by special symbols:

Operation	**Symbol**	**Description**
Addition	+	*Plus sign*
Subtraction	-	*Minus sign*
Multiplication	*	*Asterisk ("star")*
Division	/	*Forward slash*
Exponent	^	*Inverted-V symbol*

Here are some examples of usage:

```
Function Main ()

        Local energy:Int = 50
        Local sum:Int
        Local fraction:Float
        Local square:Int

        ' Multiplication...

        energy = energy * 2
        Print energy
```

```
        ' Addition and subtraction...

        sum = 10 + energy - 1
        Print sum

        ' Division...

        fraction = 10.0 / 4.0
        Print fraction

        ' Power of 2...

        square = 3 ^ 2
        Print square

End
```

Notice the use of a *float* for the *fraction* variable here, since we know in advance that's not going to end up as a whole number.

Hard-coded values in calculations

You may have noticed that the hard-coded numbers in the previous example, 10.0 and 4.0, have been specified with *point-zero* (.0) on the end.

Hard-coded values in calculations are treated specially by Monkey. A round number, such as 4, will be treated as an integer. That means that if we write *Local fraction:Float = 10 / 4*, we're asking Monkey to perform the calculation *10 / 4* using two whole numbers, *ten* and *four*.

Behind the scenes, Monkey stores the result of any calculation in a sort of temporary variable, *then* copies this result into the variable we've specified (which is *fraction* in this case).

If both of the hard-coded values are integers, Monkey uses an integer for its temporary 'behind-the-scenes' variable. The real result of *ten divided by four* is 2.5, but if Monkey is using an integer to store the temporary result, the value stored here will be 2. Integers can only store whole numbers, so the *point-5* is simply lopped off.

Now, if any of the values used in the calculation is a float, Monkey knows that the *result* will most likely be a float, so it instead uses a temporary float variable to store the result.

Therefore, even if only one of the values *10* or *4* is specified as *10.0* or *4.0*, the temporary result of the calculation will be stored correctly as a float (2.5) before being assigned to our *fraction* variable. (In the previous example, we've specified *both* numbers with *point-zero* just for consistency and easy recognition as floating point values.)

Modifying variables

So far, we've declared variables and assigned values to them, but variables are called variables for a reason – their contents can be changed; they're *variable*!

Consider an integer variable, *apples*, which contains the value 5, and imagine that we want to change its value to 6:

```
Function Main ()
```

```
        Local apples:Int = 5
        Print apples

        apples = 6
        Print apples

End
```

Giving it a new value like this is fine, but we're hard-coding the values here. What if we want to perform a calculation upon *apples* (such as adding 1 to its value) and then update *apples* to store the new value?

```
Function Main ()

        Local apples:Int = 5
        Print apples

        apples = apples + 1
        Print apples

End
```

In this example, we first set *apples* to 5, but then... what's this?

apples = apples + 1

If you read this as *apples equals apples plus 1*, it makes no sense, but if you read it in this form:

result = calculation

… it should make more sense. As you know from earlier, the *calculation* is performed first, *then* assigned to the result.

As *apples* starts out with a value of 5, the calculation is therefore *5 plus 1*:

apples = 5 + 1

The result of this calculation is *then* assigned to *apples*, with the result that *apples* now contains the value 6.

So, to increase the value of a variable by 1, we say:

variable = variable + 1

To decrease by 2, we say:

variable = variable - 2

You can of course use any of the mathematical operators; for example the multiplication operator * to multiply a value by 2:

*variable = variable * 2*

("Make *variable* equal to *variable times two*.")

You can even multiply a variable by itself:

*variable = variable * variable*

So, if *variable* starts out as 5 here, Monkey will perform the calculation 5 x 5, then assign the result back to the variable, like so:

- *variable = variable * variable*
- *variable = 5 * 5*
- *variable = 25*

A common in-game use might be to give the player a "power-up" providing a new life:

lives = lives + 1

If the player had 3 lives, the result of this calculation would mean that the player now has 4 lives.

Try covering up the left side of the table below so that you can only see the calculation on the right. You should be able to see the logic as you go down the table: *lives + 1* is effectively the same as *3 + 1*, which is the same as 4:

Result	**=**	**Calculation**
lives	=	*lives* + *1*
		⇩
lives	=	*3* + *1*
lives	=	*4*

After performing the calculation, the result 4 is stored in the *lives* variable. The calculation *lives = lives + 1* therefore increases the value of *lives* by one.

Here's that power-up in action:

```
Function Main ()
	Local lives:Int = 3
	Print "Lives: " + lives
	' Got power up!
	lives = lives + 1
	Print "Lives: " + lives
End
```

Again, as you can perform any mathematical operation, in the case of the player losing a life, you subtract a value:

lives = lives – 1

If *lives* again starts out with a value of 3, the end result of this calculation will see a result of 2 being stored in the *lives* variable. You lost a life!

The same simple method can be used in many other practical ways:

- energy boost:

  energy = energy + 20

- position change:

  x = x + 10
  y = y – 20

- slowing down:

  speed = speed – 0.1

Order of mathematical operations

The order in which mathematical operations are carried out can be very important. Try running this program, for instance:

```
Function Main ()
        Print 10 * 5 + 10 * 2 + 1
End
```

If you were to work this out from left to right, the answer is 121:

Reading from left to right, step-by-step:

10 * 5 + 10 * 2 + 1 = **50** + 10 * 2 + 1

10 * 5 + 10 * 2 + 1 = **60** * 2 + 1

10 * 5 + 10 * 2 + 1 = **120** + 1

10 * 5 + 10 * 2 + 1 = **121**

However, Monkey tells us the result is 71. What's going on? Well, in mathematics, operations are carried out in a specific order (known as the *order of operations*).

Unfortunately, the 'correct' order of operations is different depending on where you learn and who teaches you! Here are just a few variations:

Orders of operations

BODMAS: *Brackets Of Division, Multiplication, Addition, and Subtraction*

BODMAS (alternative): *Brackets, Ordinals, Division, Multiplication, Addition, and Subtraction*

BOMDAS: *Brackets Of Multiplication, Division, Addition and Subtraction*

BOMDAS (alternative): *Brackets, Ordinals, Multiplication, Division, Addition and Subtraction*

BIDMAS: *Brackets, Indices, Division, Multiplication, Addition and Subtraction*

BEDMAS: *Brackets, Exponential, Division, Multiplication, Addition and Subtraction*

PEMDAS: *Parentheses, Exponents, Multiplication, Division, Addition and Subtraction*

Which one is 'correct' is the subject of many a heated debate. If you ever accidentally drop one of these into an online programming or mathematics-based forum, don't be surprised if World War Three breaks out. (The effect is similar to that of the *"Y is not a vowel"* discussion.)

However, Monkey uses *BODMAS*, so mathematical expressions are evaluated in the order *B-O-D-M-A-S*. If you can remember that term ("bod-mass") and that it stands for "Brackets Of Division, Multiplication, Addition and Subtraction" you'll be fine for most purposes!

Feel free to skip this section and treat it as a reference; just remain aware that the order in which you carry out mathematical operations can affect the outcome.

If you choose to read on, don't worry if it starts to seem tough going; you can always come back here to check the rules if you need them!

Order of operations, Monkey-style

BODMAS: **B**rackets **O**f **D**ivision, **M**ultiplication, **A**ddition, and **S**ubtraction

Taking the basic mathematical operators first, **D**ivisions are evaluated, then **M**ultiplications, then **A**dditions, then **S**ubtractions.

Any mathematical expression placed inside **B**rackets is also evaluated according to these rules, *with expressions inside brackets considered before anything else*.

So, looking at the previous program again, let's see how Monkey works it out according to BODMAS to end up with a result of 71:

```
Function Main ()
        Print 10 * 5 + 10 * 2 + 1
End
```

First, the multiplications are carried out (*Multiplication* comes before *Addition* in BODMAS):

10 * 5 + **10 * 2** + 1

This is evaluated down to:

50 + 20 + 1

As the resulting calculations are all additions, the order is now irrelevant, so we end up with:

70 + 1 = 71

BODMAS: The B is for Brackets!

You can force mathematical expressions to be evaluated in a different order via the use of brackets (also known as parentheses). Here's a calculation performed first of all *without* any brackets to force the order of evaluation:

```
Function Main ()
        Print 100 + 2 * 5
End
```

The multiplication is evaluated first, as you would expect according to BODMAS, then the addition is carried out, so we get:

Multiplication: 100 + **2 * 5**
Addition: 100 + **10**
Result: 110

Now let's say we *want* Monkey to evaluate the addition part first:

```
Function Main ()
        Print (100 + 2) * 5
End
```

Because the *B* in BODMAS stands for brackets, Monkey looks for brackets before anything else. Therefore any calculation placed *within* brackets is performed first. Now we have:

Brackets: **(100 + 2)** * 5
Multiplication: **102** * 5
Result: 510

The BODMAS rules are further applied to any calculations *inside* brackets:

```
Function Main ()
        Print (5 * 10 + 20) + 50
End
```

Everything inside the brackets will be evaluated first, using BODMAS, so in this case, *inside the brackets*, the

multiplication happens, then the addition. After that, *outside the brackets*, the outer addition is performed.

Let's see this step-by-step:

Brackets first:	**(5 * 10 + 20)** + 50
Multiplication inside brackets:	(**5 * 10** + 20) + 50
Addition, still inside brackets:	(**50** + 20) + 50
Addition outside brackets:	(**70**) + 50
Result:	120

Nested brackets in BODMAS

Brackets can also be *nested*; that is, you can place sets of brackets *inside* other sets of brackets in order to control the order of evaluation:

```
Function Main ()
        Print (500 * (2 + 1)) - 5
End
```

The innermost brackets are always evaluated first, which means the expression *2 + 1* is solved first, then the result is multiplied by 500. Finally, 5 is subtracted from that result.

Brackets can be nested as deeply as you like, but they can start to look pretty confusing, as in this example:

```
Function Main ()
        Print 500 * (2 + (1 * (10 - 6))) - 5
End
```

If you find you're nesting calculations too deeply like this, you might want to consider performing the innermost calculations on a previous line for the sake of readability; for instance, the above might become:

```
Function Main ()
        Local inner:Int = 1 * (10 - 6)   ' This was (1 * (10 - 6)) but we
                                         ' can drop the outer brackets!
        Print 500 * (2 + inner) - 5
End
```

In closing this particular subject, you might find it helpful to use brackets even where they're not needed; for instance, this example is already in BODMAS order:

```
Function Main ()

        Print 500 * 2 + 1

End
```

However, once you know how brackets affect evaluations, you might find it easier to determine at a glance what's going on by doing something like this, even though it's not technically necessary:

```
Function Main ()

        Print (500 * 2) + 1

End
```

Arrays

Arrays allow you to create *collections* of variables. Say you're writing a game where you're a bartender keeping track of a patron's drink:

```
Local glass:Float = 1.0  ' Full glass!

glass = 0.5              ' Half full. (Or half empty... )

glass = 0.0              ' Empty glass!
```

Easy enough to manage; but what if there are two or three customers?

```
Local glass1:Float = 1.0
Local glass2:Float = 1.0
Local glass3:Float = 1.0

glass1 = 0.5
glass2 = 0.25
glass3 = 0.75 ' Slow drinker!

glass1 = 0.0
glass2 = 0.0
glass3 = 0.0
```

Well, that's still manageable, but how about a really busy bar?

```
Local glass1:Float = 1.0
Local glass2:Float = 1.0
Local glass3:Float = 1.0
Local glass4:Float = 1.0
Local glass5:Float = 1.0
Local glass6:Float = 1.0
Local glass7:Float = 1.0
Local glass8:Float = 1.0
```

```
Local glass9:Float = 1.0
Local glass10:Float = 1.0

' Random drinkin' speeds...

glass1 = 0.5
glass2 = 0.25
glass3 = 0.35
glass4 = 0.5
glass5 = 0.5
glass6 = 0.75
glass7 = 0.5
glass8 = 0.45
glass9 = 0.65
glass10 = 0.15

' Wow, they all finished at the same time!

glass1 = 0.0
glass2 = 0.0
glass3 = 0.0
glass4 = 0.0
glass5 = 0.0
glass6 = 0.0
glass7 = 0.0
glass8 = 0.0
glass9 = 0.0
glass10 = 0.0
```

This is getting silly, isn't it? Imagine if you had a hundred people in the bar!

These variables all represent the same thing – how full (or empty) a patron's glass is – so let's stop creating individual *glass* variables for each patron:

```
Local glass:Float [10]
```

If you cover up the square brackets, you have the simple declaration of a floating-point variable, *glass*, as in the first example.

The brackets, and the number within, effectively tell Monkey to create ten such variables in one go. After the declaration, we can access any of these ten variables by using the name *glass* and an index number in brackets:

```
' Create ten glasses...

Local glass:Float [10]

' Let's fill glass number 5...

glass [5] = 1.0
```

At this point, you need to be aware of a very important point: computers, unlike people, start counting from *zero*, not from one:

People:
1, 2, 3, 4, 5, etc.

Computers:
0, 1, 2, 3, 4, 5... etc.

This is very important when accessing arrays: filling the *first* glass in our example would look like this:

```
glass [0] = 1.0
```

Since computers include zero when counting, we count from 0 to 9 to access all ten array entries. Don't bother running this, since you won't see anything, but look at how we access each entry in the array:

```
Function Main ()

	Local glass:Float [10]

	glass [0] = 1.0
	glass [1] = 1.0
	glass [2] = 1.0
	glass [3] = 1.0
	glass [4] = 1.0
	glass [5] = 1.0
	glass [6] = 1.0
	glass [7] = 1.0
	glass [8] = 1.0
	glass [9] = 1.0

End
```

It's a little unintuitive to us, but... count 'em! We have entries from one to nine, plus that extra zero at the start, making for ten entries in total.

(It may be easier to grasp the numbering if you focus on the numbers within the square brackets and read your way down, putting up a new finger for each one.)

So, although you've declared an array of ten items, if you try to do this, you'll cause an error:

```
glass [10] = 1.0
```

Try adding this line to the previous example and running it. You should receive an error message something like this:

```
Array index out of range
```

That's because *glass [10]* would in fact be the *eleventh* array entry, and our array can only hold ten entries, which are strictly numbered from 0 to 9.

We'll cover loops later on, so don't worry too much about the *For/Next* stuff you see here; all you need to understand is that the *index* variable is increased from 0 through to 9 within the *For/Next* block:

```

```

```
Function Main ()

	Local glass:Float [10]			' Our array of ten glasses

	Local index:Int				' We'll be increasing this value from 0 to 9

	For index = 0 To 9
		glass [index] = 1.0		' Filling each glass!
	Next

End
```

So, the code inside the *For/Next* loop is executed ten times, with the *index* variable starting at zero, then increasing to one, then two, and so on, up to nine.

This in effect means we're accessing each array entry, from *glass [0]*, *glass [1]*, *glass [2]*, all the way up to *glass [9]*.

Compare to the length our original ten-glass version, which would look like this:

```
Function Main ()

	Local glass1:Float = 1.0
	Local glass2:Float = 1.0
	Local glass3:Float = 1.0
	Local glass4:Float = 1.0
	Local glass5:Float = 1.0
	Local glass6:Float = 1.0
	Local glass7:Float = 1.0
	Local glass8:Float = 1.0
	Local glass9:Float = 1.0
	Local glass10:Float = 1.0

End
```

And if you still think it's not that much of a saving... imagine we have *100* patrons with glasses to keep topped up! The code would be ten times as long as this!

Depending on what your array represents, you might easily be handling hundreds or even thousands of array entries: bullets and explosion particles; citizens or their houses in a city management game, etc.

Changing the array version to deal with 100 glasses means changing the initial array size from 10 to 100, and bumping up the *index* value from 9 to 99:

```
Function Main ()

	Local glass:Float [100]

	Local index:Int

	' Ten full glasses!

	For index = 0 To 99
		glass [index] = 1.0
		Print glass [index]
	Next

End
```

Why not invent a future “mega-bar” and update this to handle 1000 patrons. (Make sure you fill up *all* of their glasses, not just the first 100, or you won't be very popular!)

You might also like to try adding two more *For/Next* loops, one to set each glass to half-full (0.5) and another to empty them completely (0.0). Just copy the existing *For/Next* loop and alter the value from 1.0 as required.

Statements and expressions

Before we continue, let's look at a couple of points of *syntax* (“sin-tax”), or the rules of a language. In particular, we need to have a basic understanding of *statements* and *expressions*.

Statements

In programming, a *statement* is roughly equivalent to a sentence, or an instruction. For example, these three examples are all individual statements:

```
1) Print “Hello world”

2) If a = 10 Then Print “Hello world”

3) DrawImage player, x, y
```

We have only briefly seen points *2)* and *3)* in action, while setting up the *firstgame.monkey* project at the very beginning, but they are all effectively complete 'sentences' in Monkey language.

Take point *2)*, for example, and chop off the ending:

```
If a = 10 Then
```

This isn't a complete sentence – it's missing the required action to be taken should *a* equal ten. It's an invalid statement. You can read it out loud and it won't make sense, where the previous example will. A statement generally sounds like a complete sentence, or command, as in example *1)*, and can stand by itself.

Statements can also take the form of certain blocks of code, such as *If/Endif*. We'll be covering *If/Endif* next, but it looks something like this:

```
If a = 10
        Print “Hello world”
        Print “Something else”
        Print “Anything”
Endif
```

Everything from *If* to *Endif* is considered a statement, but there are also three statements within the *If*/*Endif* statement – the *Print* lines. (If you're struggling here, come back and have a look at this after reading about *If*/*Endif*.)

It's quite normal to have statements within statements. Here is the same line printed twice; the underlined portions of this line are both considered statements:

```
If a = 1 Then Print "True"
        ...
If a = 1 Then Print "True"
```

Expressions

An *expression* is the part of a program statement that performs a calculation or comparison, which may involve variables, literals (hard-coded values) or return values from function calls. Take the previous example:

```
If a = 1 Then Print "True"
```

In this statement, the *expression* part is *a = 1*. Here it is in a more generic form:

```
If [EXPRESSION] Then Print "True"
```

Decisions, decisions

So far, we've really done little more than assign values to variables and print them out. A real program needs to make decisions! Some examples:

- Has the player been killed too many times?
- Have all of the enemies been killed?
- Has the player clicked on a playing card? Which kind?
- Did the player pick up the Magical Potion of Awesomeness? What happens if he did?

In computer logic, a decision generally involves checking a variable (or some other piece of information) and taking an action (such as modifying another variable or calling a function) based on its value.

For example:

```
If lives = 0 Then Print “Game Over”
```

If you read this out loud you can probably tell how it works even without any explanation – it's almost plain English:

“If lives equals zero then print Game Over!” (Bill exclaimed.)

In this example, we're looking at the variable *lives* and taking an action based on its value: *If* the player has no lives, *then* print *Game Over*. Let's see it in practise:

```
Function Main ()

        Local lives:Int = 1
        Print "Lives: " + lives

        lives = lives - 1               ' Stepped on a grenade! D'oh!
        Print "Lives: " + lives

        If lives = 0 Then Print "Game Over!"

End
```

Here, *lives* starts out as one, and is then reduced by one, giving a result of zero. At the *If* check, the program determines that *lives* is indeed zero and prints *Game Over*.

What happens if *lives* is *not* zero? Well, try it: change the initial value of *lives* to two and run the program. Of course, nothing is printed. Starting with two lives and taking one away leaves us with one; when the program comes to check the value of *lives* and find it's 1, the action following *Then* is simply ignored.

If/Then

The *If/Then* statement takes the form:

***If** something is true **Then** take this action*

What if you want to take an alternative action should the *something* in question **not** be true? You use the *Else* keyword:

***If** something is true **Then** take this action **Else** take this action instead*

Taking the previous example:

```
Function Main ()

        Local lives:Int = 1

        Print "Lives: " + lives

        lives = lives - 1               ' Stepped on a grenade again! Comedy gold!
```

```
        Print "Lives: " + lives

        If lives = 0 Then Print "Game Over!" Else Print "Still Alive!"

End
```

Run the program and it will print *Game Over!* as before.

Now change the initial value of *lives* to two and run it again. Since *lives* is 1 when the program checks it, the action following *Else* is carried out instead.

If/Endif

The *If/Then* statement is useful for quick tests like this, but it's very limiting if you need to take multiple actions based on a given result.

First of all, Monkey lets you string together multiple lines by using the semi-colon character **;** to separate them:

```
Function Main ()

        ' This...

        Print "Hello"; Print "Hello again"; Print "Hello again, again"

        ' … is the same as this...

        Print "Hello"
        Print "Hello again"
        Print "Hello again, again"

End
```

We could therefore do something like this in an *If/Then* test:

```
If lives = 0 Then money = money – 1000; ammo = 0; Print “Game Over!”; Print “Try Again!”
```

… but it's really too complex to read easily, or to quickly locate a point of interest to edit later on. For anything more complicated than a single action, we use an *If/EndIf* block:

```
Function Main ()

        Local lives:Int = 0

        If lives = 0

                money = money – 100
                ammo = 0

                Print “Game Over!”
                Print “Try Again!”

        EndIf
```

```
End
```

Now it's much easier to see the individual actions being carried out. *All* of the actions between *If* and *Endif* are carried out if *lives* equals zero.

Notice that the actions are now enclosed as a block of code, indented for readability, within the *If/EndIf* keywords, just like other code blocks such as *Function/End*, *Class/End*, etc.

Now change the initial value of *lives* to one (or any other value) and run it, and you'll notice... that nothing happens! The *whole block of code* within the *If/EndIf* block is being skipped because *lives* is no longer zero.

Let's add some actions to be taken if the value of *lives isn't* zero. Just as we can use the *Else* keyword to perform an alternative action in the single-line *If/Then* statement, we can also use it in the block form:

```
Function Main ()

	Local lives:Int = 3

	Local money:Int = 5000
	Local ammo:Int = 100

	If lives = 0

		money = money - 100
		ammo = 0

		Print "Game Over!"
		Print "Try Again!"

	Else

		money = money + 10	' Still alive? Have some money!

		Print "Still Alive!"
		Print "Woo!"

	EndIf

End
```

Before you run this code, try and determine what will be printed on screen. The *money* and *ammo* variables can be safely ignored; they're just here as examples of actions to be taken. Look at the value of *lives* and trace through the program to see which block of code will be executed.

Hopefully you got that right! As a further exercise, make a change to the program that would cause the *other* block of code to be executed and make sure you understand why that is.

Finally, you can 'nest' *If/Endif* tests in order to eliminate unnecessary code execution. Let's say you'd really like to brag heartily about your forthcoming fruit intake to anyone who will listen, but *only* if you have two apples *and* two oranges together:

```

```

```
Function Main ()

        Local apples:Int = 1
        Local oranges:Int = 2

        If apples = 2

                If oranges = 2
                        Print “Got two apples AND two oranges!”
                Endif

        Endif

End
```

Here, because *apples* does not equal two, the outermost *apples* test fails and so the program skips to the matching *Endif*. (Notice that the indentation immediately shows which *Endif* matches which *If*.)

This means that the program doesn't even reach the innermost *oranges* test; that code block is ignored completely.

If you change apples to two, you'll see that the innermost code *is* then executed; the point is that it doesn't have to be executed if it's not needed; in this scenario, we don't need to check the number of oranges because we already know we don't have two apples.

An effective optimisation

This is a great way to *optimise*; that is, to avoid having to execute unnecessary code. Fast-paced arcade games don't want to be slowed down because they're executing code that doesn't even need to be run.

Say you want to test whether or not a bullet has hit an on-screen enemy; if there are a hundred enemies then you're going to have to check the position of every bullet against the position of every enemy, potentially a very hefty calculation that could slow your game to a crawl, particularly on slower devices.

If a bullet has gone outside the screen area, though, you don't need to check for collisions with on-screen enemies.

Here's the test in 'pseudo-code', or human-readable code that shows the logic but doesn't actually run:

```
If [bullet hits alien]
        [explode alien]
Endif
```

If you were testing ten rapid-fire bullets against a hundred aliens, you'd be performing this test a thousand times. The test itself could potentially involve some quite complicated arithmetic, which takes time, and multiplying that time by a thousand really could have an impact on how smoothly your game runs.

Why not eliminate as many bullets (or aliens) as possible from the test? If they're off-screen, let's ignore them! The above pseudo-code is shown here in bold:

```
If [bullet on screen]

        If [bullet hits alien]
                [explode alien]
        Endif

Endif
```

Now the potentially slow, complicated test of 'bullet-hitting-alien' (the part in bold) won't even be executed if the bullet is off-screen. We've eliminated a block of code that doesn't always need to be run, and that's good!

Comparisons

Our decisions have so far been based upon whether or not a given condition is true, that is, whether or not one value equals another:

If height equals 100 *then do something*

We also need to be able to check for the exactly opposite case in many situations. For example, what if we want to perform an action when one value *isn't* equal to another value?

If height does not equal 100 *then do something*

How about if one value is less than another?

If height is less than 100 *then do something*

Or greater than another?

If height is greater than 100 *then do something*

To make things more complicated, we might even need to know if a value is less than, *or equal to*, another!

If height is less than 100 or equals 100 *then do something*

(The same applies to *greater than or equal to*.)

That's a lot of comparison possibilities! Fortunately, for most comparisons, there are only three symbols you need to remember:

Symbol	***Meaning***
=	*Equals*
<	*Less than*

> *Greater than*

Let's see them in action:

```
Function Main ()

	Local a:Int = 1
	Local b:Int = 2

	' Comparisons...

	If a = b Then Print "a equals b!"

	If a < b Then Print "a is less than b!"

	If a > b Then Print "a is greater than b!"

End
```

Run the program, then play with the values of *a* and *b* at the top (e.g. swap their values around) to see the different results. Check the code to see which line has been executed, and therefore which comparison was true, and make sure you understand why.

That covers *equals*, *less than* and *greater than*, but how about *doesn't equal*? Well, we've already said you only need to know three symbols. In Monkey, you use a combination of the *less than* and *greater than* symbols together, < >, to mean *not equal*:

```
Function Main ()

	Local a:Int = 1
	Local b:Int = 2

	If a = b Then Print "a equals b!"
	If a <> b Then Print "a does not equal b!"

End
```

So, you can read the above *If* tests, in order, as:

"If a equals b"

… and...

"If a *does not* equal b"

Try making the initial values the same to see the difference.

Finally, you can test for the case where a value is less than, *or equal to*, another value, by combining the less than and equals symbols, like so:

```
Function Main ()

	Local dave:Int = 25
	Local bill:Int = 50
```

```
        If dave <= bill Then Print "Dave is younger than, or the same age as, Bill"
        If dave >= bill Then Print "Dave is older than, or the same age as, Bill"

        If dave = bill Then Print "Dave is the same age as Bill"

End
```

(Note that the same method applies to *the greater than or equal to* case.)

Run the program, then try playing with Dave and Bill's ages. You might have to think a little about the output you're seeing: if the ages are the same, you'll see three lines of output! (They're all technically true if you think about it.)

In summary, the valid combinations of these symbols are:

Symbol	*Meaning*
<>	*Does not equal*
<=	*Less than or equal to*
>=	*Greater than or equal to*

Multiple comparisons

Lastly, you can run a block of code based on the outcome of multiple comparisons:

```
If bullets > 0 Or laser_bolts > 0
        Print “Still got ammo!”
Endif
```

Notice the *Or* keyword between the two tests, `bullets > 0` and `laser_bolts > 0`. If *either* of these two tests is true, the code in-between will be executed.

For readability, it can be easier to place each test within brackets. This is the same code:

```
        If (bullets > 0) Or (laser_bolts > 0)
                Print “Still got ammo!”
        Endif
```

Here's that code in runnable form:

```
Function Main ()

        Local bullets:Int = 10
        Local laser_bolts:Int = 100

        If (bullets > 0) Or (laser_bolts > 0)
                Print “Still got ammo!”
        Endif

End
```

Try changing the values of *bullets* and *laser_bolts* so that one or both of them are zero.

You can also execute a block of code when *both* tests are true:

```
Function Main ()

	Local bullets:Int = 10
	Local laser_bolts:Int = 100

	If (bullets > 0) And (laser_bolts > 0)
		Print “Still got both kinds of ammo!”
	Else
		Print “Still got one kind of ammo anyway...”
	Endif

End
```

If you set either *bullets* or *laser_bolts* to zero and re-run the program, the *If* test will be false, so the *Else* block of code will be executed. *Both* values must be greater than zero for this test to be true.

```
Function Main ()

	Local fuel:Int = 100
	Local wheels:Int = 4

	Repeat

		fuel = fuel – 1
		Print “Fuel left: “ + fuel

		If fuel = 0 Or wheels = 0
			Print “Ran out of fuel, or wheels fell off! Game Over!”
			Exit
		Endif

	Forever
End
```

Boolean evaluation

These comparisons are all 'evaluated' by Monkey as *True* or *False*; that is, the result of the test will either turn out to be true, or it will turn out to be false. This 'true or false' evaluation of expressions is known as *Boolean logic*.

The code following the comparison will only be executed when the result is true:

```
Local a:Int = 1

If a = 1 Then Print “True”
```

In this example, because *a* does equal one, the comparison is true and the code after the test is executed. Change the value of *a* to anything else and the comparison is false, so nothing is printed.

Too many decisions!

After our necessary little detour into the various comparisons you can make, let's get back to basic decision making.

If/Endif is fine for simple *one-way-or-the-other* decisions, but what if you want to test for a number of different results and perform actions for each separate outcome?

Select/Case

In Monkey, we use the *Select* keyword to choose a value and the *Case* keyword to take an action depending on that value. Sounds complicated, but it's fairly simple; let's assume that *bananas* is an integer variable which is holding a value of 2:

```
Select bananas

    Case 1
        Print "You have one banana!"

    Case 2
        Print "You have two bananas!"

    Case 3
        Print "You have three bananas!"

End
```

Translating this to English, we're asking Monkey to *select* the appropriate *case* for the value of *bananas* and run the relevant code. In the case where *bananas* equals 1, it will run the code directly after *Case 1*; if *bananas* equals 2, it will run the code after *Case 2*; and so on.

Note the indentation to show that this is a separate block of code *and* to keep the separate cases clear.

In the case where the value is 2 (as it is here), the program jumps straight to *Case 2* and executes the block of code that prints *You have two bananas!*

Importantly, after executing the code for the given case, it then leaves the entire *Select/End* block and

continues onwards, ignoring all the other *Case* statements. *Only the code after the relevant* Case *statement is executed.*

Run this example and then change the value of bananas to 2, then to 3:

```
Function Main ()

	Local bananas:Int = 1

	Select bananas

		Case 1
			Print "You have one banana!"
			Print "Nobody needs more than one banana!"

		Case 2
			Print "You have two bananas!"
			Print "Two-handed banana bandit!"

		Case 3
			Print "You have three bananas!"
			Print "Look, that's really too many bananas..."

	End

	Print "OK, that's enough banana advice for now."

End
```

As you can see, the program jumps to the final *Print* statement, outside of the *Select*/*End* block, regardless of which *Case* block is executed, ignoring any previous or following *Case* blocks.

A little exercise: add a case for the value of *bananas* being zero. Print something funny relating to an unfortunate lack of bananas.

Now try changing *bananas* to a value *not* covered by any of the cases. Run the program, and as you might have guessed, it skips the entire *Select* block – since none of the cases matches – and just prints the banana advice line.

What if we wanted to carry out an action in this situation where *none* of the *Case* values matches *bananas*? Luckily, Monkey provides the *Default* keyword for just this eventuality:

```
Function Main ()

	Local bananas:Int = 100

	Select bananas

		Case 0
			Print "It is a lonely individual who has no bananas."
			Print "I cast thee out."

		Case 1
			Print "You have one banana!"
			Print "Nobody needs more than one banana!"

		Case 2
			Print "You have two bananas!"
			Print "Two-handed banana bandit!"
```

```
                        Case 3
                                Print "You have three bananas!"
                                Print "Look, that's really too many bananas..."

                        Default
                                Print "Why do you have so many bananas?"
                                Print "I could only dream of having so many bananas. Sigh..."

                End

                Print "OK, that's enough banana advice for now."

        End
```

There's no *Case 100*, so the program skips to the *Default* code block and prints an appropriate message. *Default* is therefore a sort of "catch-all" *Case*.

(You can add a case to this program specifically for the value 100 if you like – the *Case* values don't have to be sequential numbers, or in any kind of order.)

Loops

The programs we've written so far are very simple: they carry out a few simple operations involving variables, perhaps perform a few tests on them and very quickly come to an end.

Most real-world programs operate in a loop, repeating the same block (or blocks) of code over and over, like this:

Do this stuff...

Print "Hello"

Over and over.

Such a program would simply keep printing out the word *Hello* until you manually end the program. Here's what a real-world Monkey version would look like, but please ***don't run it***:

```
        ' WARNING: Do NOT run this program!

        Function Main ()

                Repeat
                        Print "Hello"
                Forever

        End
```

This program would cause the Monk IDE to freeze, for reasons that will be covered later.

Repeat/Forever

In a *Repeat/Forever* loop like this, any code between *Repeat* and *Forever* is, simply put, repeated... forever! This is known as an *infinite loop*. In most cases you need a way to be able to break out of such a loop, which will otherwise run until the end of time (or at least until you turn off your computer).

One way to "escape" from a loop like this is with the *Exit* keyword, called when a specified condition is met:

```
Function Main ()

	Local counter:Int

	Repeat

		counter = counter + 1
		Print counter

		If counter = 100 Then Exit

	Forever

	Print "Exited from loop!"

End
```

This loop will keep increasing the value of *counter*, check whether or not *counter* equals 100, then exit the loop if it does. You'll notice that program flow continues after the *Forever* line, which is the end of the loop block, telling us it's exited from the loop.

Repeat/Until

Another way to exit from a loop is by using a *Repeat/Until* block, which works in exactly the same way, but effectively moves the *If* check to the *Until* line, becoming a core part of the loop rather than a check somewhere in the middle:

```
Function Main ()

	Local counter:Int

	Repeat

		counter = counter + 1
		Print counter

	Until counter = 100

	Print "Exited from loop!"

End
```

If you just read out loud the *Repeat* and *Until* lines from this program, you should be able to see that this loop will, quote, "repeat until *counter* equals 100". It starts out with a value of zero, then on each *iteration* of the loop, *counter* is increased by one. When *counter* equals 100, the program exits the loop. (To *iterate* through a loop means to repeat it with a small change each time, such as increasing or decreasing a variable.)

A similar *loop construct* in Monkey is the *While/Wend* loop:

```
Function Main ()

	Local counter:Int

	While counter < 10

		counter = counter + 1
		Print counter

	Wend

	Print "Exited from loop!"

End
```

This also repeats a section of code, but places the test right at the start of the loop, unlike *Repeat/Until*, which places it at the end of the loop.

The difference is subtle but can have a significant effect; in a *Repeat/Until* loop, the code in-between will always be run at least once, *then* the test will be carried out to determine whether the loop should repeat.

On the other hand, in a *While* loop, if the *While* test isn't true, the code in-between won't be run and the loop will exit; this means that if the *While* test isn't true when you first enter the *While/Wend* loop, it will do nothing at all.

In the above code example, we're effectively saying "As long as *counter* is smaller than ten, execute this code and then iterate again".

The Game Loop

Games almost always run in loops: if you think of a game like Namco's famous *Pacman*, the program behind the scenes is constantly checking the player's position and comparing it with the positions of the ghosts and the pills, in order to determine would should happen – usually a score increase or instant death!

A *game loop* like this will typically test for player input from the keyboard, mouse or other controller; update the player's position on screen based on this input; update enemy positions on screen; check for collisions between the player (or the player's bullets) and the enemies; and draw everything on screen, many times per second. (Most arcade-style games aim to do all this, and draw the results, 60 times per second.)

You may recall a brief discussion covering the *OnUpdate* and *OnRender* 'methods' from the very start of this tutorial (but don't worry if not; we'll come to this later anyway). These two methods generally perform the following functions:

- **OnUpdate**:

  Runs the code, defined by you, that takes player input, updates the player's position, updates enemy positions, checks for collisions, and so on.

- **OnRender**:

Runs the code defined by you that draws everything on screen.

Monkey hides its low-level workings from you, but behind the scenes Monkey performs something like this (very simplified) loop:

```
Repeat

        OnUpdate        ' Your OnUpdate code
        OnRender        ' Your OnRender code

Forever
```

As you can see, this is an *infinite loop* that repeats the update code and the drawing code over and over. That's really how most games operate while you're playing. (The complexity comes in the actual game update and drawing code, of course.)

We'll come back to game loops later on, but first we need to know how to bundle frequently-used lines of code into *functions* that we can treat as single commands. This will also help us greatly when it comes to learning *methods*.

A very simple game

Time for a break! We've now covered enough to create a very simple game, so let's put some of this theory together.

You don't need to understand all of this right now; it's just a demonstration of what a simple game needs in order to run and some real-world usage. This will be possibly the dullest game you've ever seen – a rectangle moved via the keyboard – but will show you how a real game works and provide a basis for you to experiment with.

Monkey games are written using the *mojo* module, which, as we learned earlier, is a set of commands that let us load and display graphics, play sounds, process mouse/keyboard input, and so on.

Before all else, we need to tell Monkey we'll be using *mojo*, via the *Import* keyword:

```
Import mojo
```

We need to create an *App class*, which we'll cover in detail later, but for now we only need to know that it's necessary for the operation of our program:

```
Class Game Extends App

        Method OnCreate ()
        End

        Method OnUpdate ()
        End
```

```
        Method OnRender ()
        End

End
```

And we need a Main function, which creates a new game for us:

```
Function Main ()
        New Game
End
```

Again, you don't need to know what's going on at this point, but you might notice that *Game* is the name of the class we created, and *Main* makes reference to the *Game* class.

This is the basic structure of any Monkey game, and, put together, it looks like this:

```
' Basic Monkey game structure:

Import mojo

Class Game Extends App

        Method OnCreate ()
        End

        Method OnUpdate ()
        End

        Method OnRender ()
        End

End

Function Main ()
        New Game
End
```

In summary, we have:

- the *mojo* import;
- the *App* class;
- the Main function launching the game.

When the *Main* function creates the game, the program flow is handled behind the scenes by the *App* class and effectively works something like this:

```
OnCreate                ' Your OnCreate code, called once

Repeat

        OnUpdate        ' Your OnUpdate code, called over and over
        OnRender        ' Your OnRender code, called over and over

Forever
```

If you run the "Basic Monkey game structure" example, nothing interesting will happen. That's because *we* have to fill in the *OnCreate*, *OnUpdate* and *OnRender* methods with actual code!

Let's look at each of these methods and what they're used for:

- *OnCreate*: startup code, called once;
- *OnUpdate*: code called while the program is running; we typically read and store the player's keyboard, mouse or other input; apply any movements to the player; check for collisions between player and enemies, or between enemies and bullets, and so on;
- *OnRender*: we draw the result of these calculations.

Game loops typically update and draw the on-screen action many times per second; in general, arcade games aim for 60 frames per second. Just as a movie or cartoon is a series of still images played quickly, so is a game.

One thing we must do for any Monkey game to work is set the update rate; we do this in the *OnCreate* method, using the *mojo* module's *SetUpdateRate* command:

```
Method OnCreate ()
        SetUpdateRate 60
End
```

(For a small game, you might well load in images and sounds here too.)

We're going to fill in the *OnUpdate* method next, but we're going to need a way to track the player's position. We need some *fields* for this purpose, which are just a special kind of variable related to classes, and they're used in much the same way as the variables you already know. These will go just inside the class, here:

```
Class Game Extends App

        Field x:Int
        Field y:Int

        Method OnCreate ()
        End

        ...
```

2D positioning explained

Two-dimensional computer games refer to the on-screen position of gameplay elements (the player, the bullets, the aliens) using *x and y offsets*.

You may know that a computer display is a grid formed from thousands of tiny dots called *pixels*. A Monkey game, by default, will create a display that's 640 pixels wide and 480 pixels high, a common PC monitor resolution; this effectively divides the game's display area into a 640 by 480 pixel grid. (We refer to this display size as "*640 x 480*".)

To specify a position on the display, we state the number of pixels across and the number of pixels down. The top-left of the display is considered to be zero pixels across the screen and zero pixels down, which we state as co-ordinates *(0, 0)*. The first number in such a pair is usually referred to as the *x*-position and

the second number as the *y*-position *(x, y)*.

Pixel positions:

x is across, *y* is down

So the middle position of a 640 x 480 display is *(320, 240)*; that is, *x = 320* and *y = 240*.

This positioning system is what our *x* and *y* fields refer to, in reference to the player's position on the screen. We'll change the values of *x* and *y* to appear to move the player around the screen.

With that covered, we can define the OnUpdate method!

```
Method OnUpdate ()

        ' Check the cursor keys (arrows) and adjust the x and y
        ' positions according to which keys are being held:

        If KeyDown (KEY_LEFT) Then x = x - 2
        If KeyDown (KEY_RIGHT) Then x = x + 2

        If KeyDown (KEY_UP) Then y = y - 2
        If KeyDown (KEY_DOWN) Then y = y + 2

End
```

That'll do for now! Let's fill in OnRender and see what this baby can do!

```
Method OnRender ()

        ' Cls clears the screen:

        Cls

        ' DrawRect draws a rectangle at our x and y co-ordinates, which
        ' were calculated in the OnUpdate method. The rectangle will
        ' be 8 pixels wide and 8 pixels high:

        DrawRect x, y, 8, 8

End
```

So, putting it all together in its final runnable form, we get:

```
' Basic Monkey game structure:

Import mojo

Class Game Extends App

        Field x:Int
        Field y:Int

        Method OnCreate ()
                SetUpdateRate 60
```

```
		End

		Method OnUpdate ()

			' Check the cursor keys (arrows) and adjust the x and y
			' positions according to which keys are being held:

			If KeyDown (KEY_LEFT) Then x = x - 4
			If KeyDown (KEY_RIGHT) Then x = x + 4

			If KeyDown (KEY_UP) Then y = y - 4
			If KeyDown (KEY_DOWN) Then y = y + 4

		End

		Method OnRender ()

			' Cls clears the screen:

			Cls 0, 0, 0

			' DrawRect draws a rectangle at our x and y co-ordinates, which
			' were calculated in the OnUpdate method. The rectangle will
			' be 8 pixels wide and 8 pixels high:

			DrawRect x, y, 8, 8

		End


End

Function Main ()
		New Game
End
```

Monkey calls *OnCreate*, which sets the update rate (60 frames per second), then calls *OnUpdate* and *OnRender* in an infinitely-repeating loop.

Technically, the program calls OnUpdate *60 times per second, as per the update rate we set (unless your computer can't keep up), and calls* OnRender *"whenever it can", which usually means "about the same number of times". For simplicity's sake, though, we'll just assume they're both called 60 times per second!*

Build and run this game, press the cursor keys to move the rectangle and marvel at its power and simplicity! All Monkey games using *mojo* are built from a similar foundation to this.

We'll cover the details of classes and *mojo* in more depth later, but here are some changes you can make to the program. First of all, let's set *x* and *y* to the middle of the 640 x 480 display on startup, by modifying *OnCreate* with the lines in bold here:

```
Method OnCreate ()

		x = 320
		y = 240

		SetUpdateRate 60

End
```

Run this and the player will start in the middle of the screen.

You might have noticed that it's possible to move outside of the screen area – try it if not – so let's force the x and y positions to remain within the display. Modify *OnUpdate* by adding the bold lines below:

```
Method OnUpdate ()

        ' Check the cursor keys (arrows) and adjust the x and y
        ' positions according to which keys are being held:

        If KeyDown (KEY_LEFT) Then x = x - 2
        If KeyDown (KEY_RIGHT) Then x = x + 2
        If KeyDown (KEY_UP) Then y = y - 2
        If KeyDown (KEY_DOWN) Then y = y + 2

        ' Don't let x or y go outside the 640 x 480 display area:

        If x < 0 Then x = 0
        If x > 639 Then x = 639

        If y < 0 Then y = 0
        If y > 479 Then y = 479

End
```

You'll now find you can't move outside the display area any more. We've done this by limiting the values of the *x* and *y* fields using *less than* and *greater than* checks.

Looking at *x*, we check to see if it's less than zero (zero is the left edge of the screen); if it is less than zero, we force it to zero.

We then check to see if it's greater than 639: the display is 640 pixels wide, but the numbering starts from zero, so the rightmost pixel is in fact 639 pixels across. If *x* is greater than 639, we force it back to 639.

(We don't see the player 'jump' awkwardly when we force these values, as it all happens before we get to *OnRender*!)

The same conditions are then applied to the value of *y*, so it's similarly limited to the display area.

Modifying the game

Here are some tweaks you can apply to this code:

- lower the update rate to see how it slows down the movement of the whole game. Put it back to 60 when you're done;

- change the numbers after *Cls*. They're currently 0, 0, 0, which are red, green and blue colour values respectively; we can mix these values together to form any colour imaginable.

  Using zero for all three means there is no red, no green and no blue in the colour, which is why we have a black screen! You can give each number a value from 0 to 255. For example, *Cls 255, 0, 0* will give you a bright red screen, *Cls 0, 255, 0* a green screen and *Cls 0, 0, 255* a blue screen; *Cls 255, 255, 255* is pure white.

Mix and match all three values as you see fit: *Cls 96, 128, 255* gives a pale blue, for example.

- change the movement speed; it's currently 2; try making it 1 for all movement directions. Any ideas as to how you might change the speed to a non-whole number; 2.5, for example? Hint: look at the type of the *x* and *y* fields!

- invert the up/down controls; you can either change the keycodes used in *OnUpdate* (KEY_UP and KEY_DOWN) *or* the plus/minus signs used to change the value of *y*.

  If you're feeling adventurous, refer to the Monkey docs (in Monk, go the the *Docs* tab, click *Module Reference*, then *mojo.input*), you'll find a list of 'key codes'. You can change the key codes used in OnUpdate and thereby change which keys the game uses.

  The key code in the excerpt below is *KEY_LEFT*, so that's the part you would replace with a different key code chosen from the docs:

```
If KeyDown (KEY_LEFT) ...
```

Play with the game code until you either get bored or break it beyond repair!

For now, it's time to get back to the theory, and learn about...

Functions

Functions, at their simplest, make it easy to bundle frequently-used lines of code into a single 'command'. Take this example program:

```
Function Main ()

        Print "Hello!"
        Print "Hello again!"
        Print "Hello, for the last time!"

End
```

Let's say you wanted to execute those three *Print* lines many times throughout your program – an unlikely scenario, but nice and simple to grasp!

You could type out (or copy/paste) those lines each time you need them:

```
Function Main ()

        Local a:Int = 100

        If a = 100
                Print "Hello!"
                Print "Hello again!"
                Print "Hello, for the last time!"
```

```
        Endif

        If a = 101
                Print "Hello!"
                Print "Hello again!"
                Print "Hello, for the last time!"
        Endif

        If a = 102
                Print "Hello!"
                Print "Hello again!"
                Print "Hello, for the last time!"
        Endif

End
```

Pretty tedious – and messy! How about this?

```
Function Main ()

        Local a:Int = 100

        If a = 100 Then PrintStuff ()
        If a = 101 Then PrintStuff ()
        If a = 102 Then PrintStuff ()

End

Function PrintStuff ()

        Print "Hello!"
        Print "Hello again!"
        Print "Hello, for the last time!"

End
```

(Note that we can now use the single-line *If/Then* since we don't need to execute multiple lines each time. It's much neater and the repeated code has been moved out of the way, into its own function.)

Notice that I've placed the *PrintStuff* function below the *Main* function in this example, but there's no reason it couldn't be placed above it instead. (Cut and paste it if you want to try.)

No matter what order you declare your functions in, the *Main* function will *always* be called first; the other functions are only called when *you* decide they should be called.

The program won't just run through each function in turn; at the end of the Main function, this program will simply exit; *PrintStuff* is only called at the point where we ask for it to be called.

We know that *a* is 100 in this example, so *PrintStuff* will be called when this value is checked. The program flow will "jump into" the *PrintStuff* function and execute the code within.

At the end of the *PrintStuff* function, program flow returns back to the *Main* function, to the statement just after we called *PrintStuff*. Therefore, after PrintStuff has run, the next line in the above excerpt will be *If a = 101*.

As the repeated code has been placed into the *PrintStuff* function, each time we want to run those three lines

of code, instead of typing them out in full, we just type *PrintStuff ()*.

As with variables, the name you give to a particular function is up to you, and the same naming rules apply, so *PrintStuff* could just as easily be called *SayHello* or *Squirrels*. As long as that's the name you use when you call the function, it doesn't matter, but you should make function names as descriptive as possible, while still being easy to type.

(The Monkey convention is to type function names with upper-case initials for each "word", as in *FireMachineGun*, *FireCannon*, *CheckPosition*, etc, though it's not mandatory.)

To declare a simple function, you type the keyword *Function*, its name (chosen by you) and add opening and closing brackets; you then terminate it with a closing *End*, like so:

```
Function MyFunction ()

End
```

The code to be executed by your function is then placed in-between (and indented), just as you do with the Main function:

```
Function MyFunction ()
	Print "Hello"
End
```

Functions can often be thought of as miniature standalone programs to be executed whenever you want (this one prints "Hello"), and, if written well, can be copied and pasted to be re-used in future projects.

Function parameters

Functions can take *parameters*, that is, values that modify how they work. Let's say that instead of the word "Hello" each time you call *PrintStuff*, you want it to print "Goodbye" (or, indeed, anything else) in certain situations. We can *pass* a string value as a parameter, like this:

```
Function PrintStuff (message:String)
	Print message
End
```

This code won't run (there's no *Main* function to call it from), but let's take a look at its construction.

We've added a string-type parameter, *message*, in between the brackets. (They're more correctly called parentheses, but most people call them brackets, so we will too – common usage prevails!)

A parameter is effectively a variable that *only the code within the function can access*. The point of a parameter,

though, is that it can accept a value which can be different each time the function is called:

```
Function PrintStuff (message:String)
	Print message
End

Function Main ()
	PrintStuff ("Hello")
End
```

I've placed the *PrintStuff* function at the top of the code here, so you can see the *function definition* to start with, and the *call* later on, inside *Main*.

Remember, the program starts at *Main*, where it:

- jumps to *PrintStuff*;
- runs the *Print message* line, using the "Hello" string passed as a parameter;
- returns to *Main*, where it runs out of code and exits.

(Always look for the *Main* function first when reading a Monkey program; you can then trace its flow from there.)

Run the program, then change the value being passed to *PrintStuff*, i.e. change the word *Hello* to something else.

Let's see the original example with modifiable *PrintStuff* output:

```
Function Main ()

	Local a:Int = 100

	If a = 100 Then PrintStuff ("Hello")
	If a = 101 Then PrintStuff ("Hello")
	If a = 102 Then PrintStuff ("Goodbye")

End

Function PrintStuff (message:String)

	Print message

	Print message + "!"
	Print message + " again!"
	Print message + ", for the last time!"

End
```

If you change the original value of *a* to 102, the messages displayed by *PrintStuff* will change. You can therefore see that it's possible to change the effect of a function call by passing different values.

Now let's pass an *integer* value to a function. We'll use that value to print different messages depending on what value is passed:

```

```

```
Function DoStuff (action:Int)

	Select action

		Case 1
			Print "Gone to the shops."

		Case 2
			Print "Gone to a party. Woo."

		Case 3
			Print "Gone hunt'n'. (H'yuk!)"

		Default
			Print "Don't know what that action is!"

	End

End

Function Main ()

	DoStuff (1)

End
```

So, we pass a value of 1 when calling the *DoStuff* function, and then, inside the *DoStuff* function, the *action* parameter *receives* that value. We can therefore treat *action* as a variable containing the value 1.

The *Select* test inside *DoStuff* looks at the value of *action* and prints the appropriate message.

> *Variable scope: a brief aside*
>
> Note that *action* (and any other variables that might be created within the *DoStuff* function) can *only* be accessed from within the *DoStuff* function it belongs to; if you try accessing it from Main you'll receive an error; try it: add a *Print action* line to the Main function. This is an example of *variable scope*, which we'll cover very shortly.

Try changing the value passed to *DoStuff* and checking the output message to understand how it works.

Here's how such a function might be used in a real game:

```
Function Main ()

	Fire (1)

End

Function Fire (shot_type:Int)

	Select shot_type

		Case 1
			Print "Machine gun fired!"

		Case 2
			Print "Rocket fired!"

		Case 3
			Print "Thermonuclear missile fired!"
```

```
                    Default
                            Print "Phwup..." ' Unknown shot type!

            End

    End
```

Rather than printing simple text, you might use each *Case* to select a different image to be drawn when the player fires a shot, as well as to decide how much damage will be inflicted upon the unlucky recipient.

In this example, the program simply calls *Fire* once and then exits, but in a real game loop you might call *Fire* every time the player presses the *Space* key, for example.

Multiple parameters

You can also pass multiple parameters to functions. Here's an example where we pass two values, one for *apples* and one for *oranges*:

```
    Function Main ()

            PrintFruit (2, 4)

    End

    Function PrintFruit (apples:Int, oranges:Int)

            Print "Apples: " + apples
            Print "Oranges: " + oranges

            ' Or, to put it another way...

            Print "You have " + apples + " apples and " + oranges + " oranges!"

    End
```

The parameters can be of mixed types, such as one integer and one float, and you can use as many parameters as you like, though it's best to keep things as minimal as possible.

The parameters are interpreted by the function according to the order in which they are passed: in this case, the first value passed, 2, will be received as the *apples* parameter of *PrintFruit*, and the second value, 4, will be received as *oranges*.

You *must* pass the correct number of parameters required by a function. However, it is possible for a function to have *optional parameters*, which can be skipped...

Optional parameters

You can specify default values for some or all parameters in a function, which means that those parameters can be omitted when calling the function, i.e. they become optional. Here's a simple example:

```
Function Main ()
        CountBullets ()
End
Function CountBullets (ammo:Int = 100)
        Print "Bullets: " + ammo
End
```

Run this program, and notice that the output from *CountBullets* tells us we have 100 bullets, even though we passed no parameters when calling it from *Main*. The default value is defined in the *ammo* parameter of *CountBullets*, so if our function has a default parameter, *ammo*, and we choose to pass no value for *ammo*, a default value of 100 will be used.

Try passing a specific value in the *CountBullets* call, within *Main,* to see the difference, e.g. *CountBullets (99)*.

You can have multiple parameters with optional values, but you should note that if you mix normal 'required' parameters with optional parameters, the optional parameters *must* come last in the function definition.

What does this mean? Well, this is *not* valid:

Function CountBullets (ammo:Int = 100, gun_type:Int)

… because the optional parameter comes first, but this *is* valid:

Function CountBullets (gun_type:Int , ammo:Int = 100)

Note that the parameters are the same, just defined in a different order. Any parameters with default values *must* be defined *after* those that require values to be passed. Hence, *gun_type* in this instance must come before *ammo*.

Here's a more complicated version; the meanings don't matter here, but notice how the multiple parameters that require you to pass values are defined first, and those with optional/default values are defined last:

Function CountBullets (gun_type:Int , gun_ready:Int, ammo:Int = 100, shot_speed:Int = 10)

In this case, all of these calls are valid:

CountBullets (1, 1)
CountBullets (1, 1, 99)
CountBullets (1, 1, 99, 0)

As a minimum, in this case you *must* pass at least the first two parameters since they have no default values.

Functions, like all of the examples so far, which simply execute a block of code, are often called commands: we tell them to go and do something, and they do it.

They can be so much more useful, though, when they go off and do something, then tell us the result of what they did!

Returning values

This is where functions become really powerful.

You can perform calculations within a function, then *return* the result of those calculations to the point in the code from where you called the function.

What does this mean? Well, let's take a really simple example: how old will someone be in 10 years' time? What we need to do, of course, is take the player's current age and add 10:

```
Function Main ()

	Local current_age:Int = 20

	PrintAgeInTenYears (current_age)

End

Function PrintAgeInTenYears (age:Int)

	age = age + 10
	Print age

End
```

That's easy enough; the *PrintAgeInTenYears* function takes the value you pass to it, adds 10 and prints the result.

But what if you want to do something else with that result instead of just printing it right away? You really just want the function to carry out the calculation – adding 10 to the value passed – and then tell you the result:

```
Function Main ()

	Local current_age:Int = 20

	Local result:Int = GetAgeInTenYears (current_age)

	Print result

End

Function GetAgeInTenYears:Int (age:Int)

	age = age + 10

	Return age

End
```

The point here is that the *result* variable in *Main* will ultimately contain the value *returned* by the *GetAgeInTenYears* function.

Let's look at this in detail: in the *Main* function we create an *Int* variable, *current_age*, containing the player's current age, 20.

We're looking for the player's current age, plus ten years, so we've created a new variable called *result* to store it in. (You could call this variable anything you want, of course.)

As you already know, you can assign a value to a variable as soon as you create it (e.g. *Local result:Int = 5*), and that's really all we're doing here. The only difference is that instead of assigning a hard-coded value to *result*, or the contents of an existing variable (as in *Local result:Int = ammo*), we're assigning a value obtained from a function call (*Local result:Int = GetAgeInTenYears ()*).

Let's take a closer look at the *GetAgeInTenYears* function; in fact, let's go line-by-line:

```
Function GetAgeInTenYears:Int (age:Int)
        age = age + 10
        Return age
End
```

The first line looks like the previous function examples, except we now have an *Int* suffix *after the function name*. This function is intended to perform a calculation and return a result, and just as a variable holds a value of a given type – *integer*, *float*, *string*, and so on – *a function can only return a result of a given type*.

So, in this case, the *GetAgeInTenYears* function returns an integer value – this is what the *Int* suffix after the function name tells us.

The *age = age + 10* line simply takes the value passed to the function via the *age* parameter (we're passing *current_age*) and adds ten to it.

The final line inside the function block, *Return age*, is the key: it *returns* the value of *age* to the line that called the function. Go back and look at the function call in *Main*:

```
Local result:Int = GetAgeInTenYears (current_age)
```

The *result* variable here will receive the value that was calculated within *GetAgeInTenYears*.

Variable scope

All this time we've been declaring variables using the *Local* keyword, but why *Local*?

Declaring a variable using *Local* means that it is local to the function it's declared in – it belongs to that function, just as *you* might declare that you 'belong 'to a particular area; you are local to that town, block,

street or whatever.

When we declare a variable as *Local* within a function, we are stating that *the variable can only be used within that function*. Try this:

```
Function Example ()
	Local something:Int = 100
	Print something
End

Function Main ()
	Example ()
	Print something
End
```

When you run this, Monkey will complain that the variable *something* hasn't been declared. ("Identifier 'something' not found.")

We can see that it *has* been declared, but it is local to the *Example* function. *Main* doesn't even know it exists! A function cannot 'see' another function's local variables.

We could declare another *something* variable within *Main*, but it's important to understand that it would be completely unrelated to the *something* variable in *Example:*

```
Function Example ()
	Local something:Int = 100
	Print something
End

Function Main ()
	Local something:Int
	Example ()
	Print something
End
```

This will print 100 followed by 0; can you see why?

The *something* variable declared inside *Main* contains 0 by default, and has nothing to do with the *something* variable inside *Example*, which contains 100.

This is called *variable scope*, which defines which parts of the program can access a given variable . A variable with *Local* scope can only be accessed from the function in which it is declared.

What if you *do* want a variable to be accessible to all functions?

Global variables

To allow a variable to be accessed from any part of a program, we declare it outside of all functions, using the *Global* keyword instead of *Local*.

In every other respect, global variables work just like the local variables you already know: you give them a type, you assign values to them, you perform calculations and tests on them; but you can access them anywhere:

```
Global Something:Int = 100

Function Example ()
	Print Something
End

Function Main ()
	Print Something
	Example ()
End
```

Here, *Main* will print the value of *Something*, then call *Example* (which also prints the value of *Something*).

Notice that this is the same code as in the previous example, but we've moved the *Something* variable 'outside' of all functions and declared it using *Global* rather than *Local*.

You might also notice that we've declared the global variable with a capital letter *S*: starting globals with a capital letter is another Monkey naming convention for ease of recognition. As usual, you don't *have* to do it, but following this convention helps you instantly to tell the scope of a given variable when reading through your code.

The *Something* variable now has *global scope*, so all functions can access and alter its contents. The result of this change is that both functions now print the same value, 100.

As with locals, you don't have to assign the value at the start:

```
Global Something:Int

Function Example ()
	Something = 100
	Print Something
End

Function Main ()
	Print Something
	Example ()
	Print Something
End
```

Trace through this example to work out why it prints 0, 100 and 100. As long as you remember that all program flow starts at *Main*, you should have no problems!

That last statement might seem given this example, since Monkey clearly reads the value of *Something* before it gets to *Main*.

However at the top of your code, outside of all functions, all you can do is make certain declarations, such as imports (as you did with *Import mojo* in the earlier game example), and the declaration of constants and globals; you can't do anything else, such as calling *Print* or other functions, using *If/Then*, loops, etc; so there's no real program flow until *Main*.

Why use Local?

It might seem a no-brainer to just make everything accessible to all functions and use *Global* for all variables. *Don't do it!* Only use *Global* where you really need to.

While modern PCs have huge amounts of memory, devices such as smart-phones are a lot more limited, and programs that use too much memory can slow down the whole system. (Even on PCs, you should aim to use as little memory as possible, to be "system-friendly".)

When you call a function, Monkey will allocate the memory needed to store its local variables; when the program exits that function, this memory can be freed.

On exiting a function, its variables are no longer accessible and are said to be *out of scope*. When a variable goes out of scope, it is destroyed – its contents are made void and its memory is freed up. (It'll be reallocated from scratch next time you call the function.)

Because global variables are *always* accessible from anywhere in the program, they are always *in scope*, and their memory can never be freed. (Their contents have to stay in memory to be accessible all the time.)

This scope/memory issue is one of the reasons why globals should not be used by default, but other important reasons include:

- *speed*: for technical reasons we won't cover here, the computer's processor can usually work much faster with local variables than globals;

- *code organisation*: declaring a variable within a function means that you can *see* and refer to the variable while you're writing the function; if it was declared along with a hundred other variables somewhere at the top of your program, you'd have to keep scrolling up and down within the program (potentially thousands of lines long) to double-check variable names, type, etc;

- *limited access to information*: this is particularly important when you get to object-oriented programming, as we're about to do, where an important part of the concept is to limit access to information in order to reduce the potential for error.

  In short, the different parts of a program should know as little as possible about each other. This way, you know that a given variable can *only* be modified by a given function or method, which helps greatly in tracking down bugs. If all variables are global, any part of the program could potentially be causing the error.

(Variable scope also hugely affects objects, but we'll discuss that separately.)

At this point, you might like to try writing some simple programs of your own, or building upon the earlier game example and put your new-found knowledge of variables, loops, program flow and all the rest to some practical use.

If you choose to just plough straight on, I would recommend at least stopping for a cup of tea (or your beverage of choice)! Our next topic will be object-oriented programming, a concept that *does* require a decent understanding of all the topics we've covered so far.

Object-oriented programming

"Object-oriented programming, eh? Sounds confusing!"

Well, it can *get* confusing, but the basic concepts are actually quite simple.

So far, we've just created a bunch of unrelated variables and toyed with them: you may remember that we represented a kilogram bag of sugar as a float value and we briefly did the same for flour. But if we're baking a cake, why not represent a cake as the collection of ingredients that it really is?

```
' Cake is made up of these variables:

Local sugar:Float
Local flour:Float
Local butter:Float
Local eggs:Int
```

If we could bundle all of these variables together into a 'cake object', it might look something like this:

```
Cake Object Contains

        Local sugar:Float
        Local flour:Float
        Local butter:Float
        Local eggs:Int

End
```

(Don't try to run this – it's not real Monkey code!)

So, if we were to create a cake object and assign a value to each of its ingredient variables, we'd then have a single entity – a cake – that can be manipulated and passed around the program, with all of these ingredient variables bundled together inside it.

Instead of passing a bunch of ingredient variables to a *BakeCake* function, for example, we can simply pass a single *Cake* object; the *BakeCake* function could then access the *Cake* object's fields as necessary:

```
Function BakeCake (Cake Object)

        Print Cake Object's sugar variable
        Print Cake Object's flour variable
        Print Cake Object's butter variable
        Print Cake Object's eggs variable

End
```

(Again, this is not real code.)

In Monkey, we declare the contents of our “Cake” object using the *Class* keyword, followed by the name of the class:

```
Class Cake

	Field sugar:Float
	Field flour:Float
	Field butter:Float
	Field eggs:Int

End
```

You'll also notice that we declare the variables using the *Field* keyword instead of *Local*, but otherwise they work just as you'd expect.

So, a *cake* is an object, just as a *car* is an object; a *person* is an object; a *cat* is an object; and so is a dog. The player's spaceship in a game is an object; its bullets are objects too. Anything can be considered an object!

Rather than trying to make a bunch of unconnected variables interact with each other, we make whole objects interact with each other, as in the real world.

As you've seen, we define objects (such as the aforementioned cakes, cars, people, cats, dogs, spaceships and bullets) using the *Class* keyword. We're creating a new type, or *class*, of object, which is a list of *features* of the object; or at least the features we're interested in.

Take a car, for example: in a driving game we might be interested in knowing its colour (for display purposes), its top speed, its current speed and its fuel.

Let's build a car!

Classes and objects

A very basic class is simply a bunch of variables bundled together. We could therefore create a *Car* class describing the features of a car, which might be made up of the following variables:

```
Variable		Suggested type

colour			A string, eg “Red”
top_speed		We'll use an integer for this, since it can be a fixed amount
current_speed		This can vary by fractional amounts, so we'll use a float
fuel			Ditto!
```

A class's variables are called *fields*, so a basic Monkey class describing a car might look like this:

```
Class Car
```

```
        Field colour:String
        Field top_speed:Int
        Field current_speed:Float
        Field fuel:Float

End
```

This code does nothing as it stands; it just describes the particular features of a car that we're interested in for our game. We can think of this class as a "blueprint" for a car; the design plans.

It's just a *description* of a Car object, though, and that means we have to actually *build* a car from this description for it to be of any use!

Constructing an object

To build a car in the game world, we create a Car object. We do this in much the same way that we've declared variables all along:

```
Local auto:Car = New Car
```

Here, *auto* is the name of the variable, and *Car* is the name of the class. After this line is run, we can consider *auto* to be a Car object.

Let's look at this in two parts, before and after the equals sign. Here's the variable we're creating, *auto*:

```
Local auto:Car
```

Rather than being of an integer, float or string type, it's using our new variable type: *Car*. Other than that, it's declared like any other variable, using the *Local* keyword.

Then, after the equals sign, we have:

```
New Car
```

This is the part that "reads" the blueprint (the Car class) and produces an actual Car object for us. The resulting object is then assigned to the *auto* variable. Put back together, we have:

```
Local auto:Car = New Car
```

Once we have a Car object assigned to the *auto* variable, we can access the object's fields. Fields are used just like ordinary variables, and you can do exactly the same things with them, such as *If/Then* checks, addition, subtraction, string joining, etc.

One important difference is that you refer to them in a special way, by specifying the name of the variable, followed by a dot, followed by the field name. If *auto* is a Car object and we want to set its *fuel* field to 100,

we type:

auto.fuel = 100

It works just like an ordinary variable; take *fuel*, for example:

```
fuel = 100
```

We just put *auto.* in front of it to indicate that we mean the *auto* object's *fuel* variable:

```
auto.fuel = 100
```

So, in a more generalised form, it's:

```
object.field = 100
```

Don't try to run the code below, but read it and apply the above principles to see if you understand what's happening. In particular, read the *Local* declaration and assignment in two separate parts, before and after the equals sign:

```
' Create a Car-type variable and assign a new Car object to it:

Local auto:Car = New Car

' Set the Car object's variables, or fields:

auto.colour = “Red”
auto.top_speed = 180
auto.current_speed = 0
auto.fuel = 75
```

Let's see this class in action; you can run this code:

```
' The blueprint for a Car object:

Class Car

        Field colour:String
        Field top_speed:Int
        Field current_speed:Float
        Field fuel:Float

End

Function Main ()

        ' The creation of a Car object:

        Local auto:Car = New Car

        ' The Car object, auto, contains all of the fields in the Car blueprint:
```

```
        auto.colour = “Red”
        auto.top_speed = 180
        auto.current_speed = 0
        auto.fuel = 75

        Print “The colour of this car is: “ + auto.colour
        Print auto.fuel

End
```

Now, let's say we want to reduce the amount of fuel in the car as it drives along; we know that the *Car* class contains a field called *fuel*, and we know that this is just a normal float-type variable.

If we want the car's fuel level to drop, we just subtract as we would with 'normal' variables, like so:

```
auto.fuel = auto.fuel - 1
Print auto.fuel
```

We can also test these fields using *If/Endif*, just as we do with variables:

```
If auto.fuel = 0
        Print “Out of fuel”
        Print “Game Over!”
Endif
```

Another example

You can give a new class almost any name you like, just as you can with variables and functions, but the name should of course relate to the object you're describing.

Here's a working example; we'll create a simple *Rocket* class for variety; we'll assume that our rocket has fuel and can provide a certain level of thrust, so we first create a class containing these two features, then in our *Main* function we'll create a Rocket-type object and access its fields:

```
Class Rocket
        Field fuel:Int = 100
        Field thrust:Int
End

Function Main ()

        Local player:Rocket = New Rocket

        Print "Fuel: " + player.fuel
        Print "We have a full tank of fuel, captain! Applying thrust!"

        player.thrust = 10

        Repeat

                player.fuel = player.fuel – player.thrust
                Print "Fuel: " + player.fuel

        Until player.fuel <= 0
```

```
		Print "Out of fuel!"

	End
```

In this example, you might notice that the fuel level (100) is declared in the class definition, similar to the way you can assign a value to a variable when you declare it, as in *Local fuel:Int = 100*.

Setting the value of a field in the class – in the actual "design plans", as it were – means that any new *Rocket* object will start with *fuel* already set to 100 upon creation. If you create a thousand rockets on-screen, they'll all have a fuel level of 100, yet we've only had to set this value once, in the blueprint itself.

As with standard variables, if you don't manually set the values of fields, they'll receive default values: zero for numbers or an empty string for strings.

In the code above, we print the amount of fuel available, set the thrust level, then repeat in a loop, reducing the fuel according to the thrust applied. When it reaches zero, we're "all outta gas" and we exit the loop.

Note the code that reduces the fuel value:

```
	player.fuel = player.fuel - player.thrust
```

This is no different in construction to:

```
	lives = lives - 1
```

The only practical difference is the way we refer to the fields, via *object.field* convention.

(In the above code example, I've used the *less than or equal to* comparison when checking the fuel value, in case you want to play with the thrust level and the final step takes it below zero. If I used the *equals* comparison, the fuel value might jump below zero (so never *equalling* zero) and the program would never end.)

Class versus object

Early on, it's easy to mix up class names and object variables; here's a common point of confusion, for instance:

```
	Class Rocket
		Field fuel:Int
		Field thrust:Int
	End

	Function Main ()

		Local player:Rocket = New Rocket

		Rocket.fuel = 100
```

```
        Rocket.thrust = 50

End
```

Run this, then see if you can tell why it doesn't work, *without* reading on!

The problem is that we've mixed up class and object; we should be accessing *player.fuel*, not *Rocket.fuel*:

```
Class Rocket
        Field fuel:Int
        Field thrust:Int
End

Function Main ()

        Local player:Rocket = New Rocket

        player.fuel = 100
        player.thrust = 50

End
```

The difference between class and object is that:

- the *class* here is *Rocket* – that's the blueprint for creating *Rocket* objects, just a *description*;
- the *object* here is *player*, the variable created from that description.

We generally only use the class name, *Rocket*, during the *creation* of objects, to tell Monkey which blueprint it should build an object from. To perform in-game actions using the newly-created *Rocket* object, we refer to the *player* variable, hence ***player***.*fuel*, ***player***.*thrust*, and so on.

Why bother?

You could still write this program using standard integer variables and no class, but when you start to define lots of rocket features, like so...

```
Local name:String
Local sprite:Image

Local x:Float
Local y:Float

Local xspeed:Float
Local yspeed:Float

Local fuel:Int
Local thrust:Int
Local ammo:Int
Local damage:Int
```

… and then want to add a second rocket to the game, it all gets a little wordy... and clumsy:

```
' First player...
```

```
Local name:String
Local sprite:Image

Local x:Float
Local y:Float

Local xspeed:Float
Local yspeed:Float

Local fuel:Int
Local thrust:Int
Local ammo:Int
Local damage:Int

' Second player...

Local name2:String
Local sprite2:Image

Local x2:Float
Local y2:Float

Local xspeed2:Float
Local yspeed2:Float

Local fuel2:Int
Local thrust2:Int
Local ammo2:Int
Local damage2:Int
```

What if you want to create 10 players? Well, you could keep copying, pasting and editing the number on the end of each variable, I suppose. How about if you're creating bullets to fly across the screen? You might have thousands of bullets over the game's running time!

You could potentially use arrays for this purpose, but classes bundle all of an object's features into a single definition:

```
Class Rocket

        Field name:String
        Field sprite:Image

        Field x:Float
        Field y:Float

        Field xspeed:Float
        Field yspeed:Float

        Field fuel:Int
        Field thrust:Int
        Field ammo:Int
        Field damage:Int

End
```

Rather than being a long list of variables, as in the previous example, classes create a powerful mental model of a single 'thing' – an object – that *possesses* all of these features.

Now you can create an infinite number of *Rocket* objects and you've only had to define their features once:

```
Class Rocket

	Field name:String
	Field sprite:Image

	Field x:Float
	Field y:Float

	Field xspeed:Float
	Field yspeed:Float

	Field fuel:Int
	Field thrust:Int
	Field ammo:Int
	Field damage:Int

End

Function Main ()

	Local player1:Rocket = New Rocket
	Local player2:Rocket = New Rocket
	Local player3:Rocket = New Rocket

	Print player1.fuel
	Print player2.fuel
	Print player3.fuel

End
```

Objects are 'Null' by default

One point that's worth making here is that declaring a variable of a given class isn't enough to create an object:

```
Class Rocket
	Field fuel:Int = 100
End

Function Main ()

	Local player:Rocket

	Print player.fuel

End
```

If you run this, you'll receive an error. We've declared a *Rocket* object and tried to print out its fuel level, but it fails. What's wrong?

Although we've declared the *Rocket* variable, *player*, we haven't actually 'built' a *Rocket* object to assign to it – remember the *New* keyword?

The *player* variable effectively refers to a non-existent object at this point; we call this value *Null*, and if we try to perform any operations on a *Null* object, the program will fail: the object doesn't exist.

Here's a corrected version, which assigns a *Rocket* object to the *player* variable:

```
Class Rocket
	Field fuel:Int = 100
End

Function Main ()

	Local player:Rocket = New Rocket

	Print player.fuel

End
```

Alternatively, you can declare a *Rocket*-type variable in advance and assign a *Rocket* object to it later:

```
Class Rocket
	Field fuel:Int = 100
End

Function Main ()

	Local player1:Rocket
	Local player2:Rocket
	Local player3:Rocket

	player1 = New Rocket
	player2 = New Rocket
	player3 = New Rocket

	Print player1.fuel
	Print player2.fuel
	Print player3.fuel

End
```

This is similar to declaring a normal variable in advance and assigning a value later:

```
Function Main ()

	Local fuel:Int

	fuel = 100

	Print fuel

End
```

One reason you might wish to declare a *Rocket* variable on one line, but assign an object to it on a separate line, is to keep your code organised and compartmentalised. You might want to create a list of all Rockets (and of other game objects) at the top of your code, then assign them all in another section, like so:

```
Function Main ()

	Local player1:Rocket
	Local player2:Rocket
	Local player3:Rocket
	Local player4:Rocket
```

```
        player1 = New Rocket
        player2 = New Rocket
        player3 = New Rocket
        player4 = New Rocket

End
```

Most experienced programmers would probably advise you to separate the *initial declaration* of variables and the *assignment of values* to those variables like this, but it's a matter of personal preference. Here's another way to do the same thing:

```
Function Main ()

        Local player1:Rocket = New Rocket
        Local player2:Rocket = New Rocket
        Local player3:Rocket = New Rocket
        Local player4:Rocket = New Rocket

End
```

Just be aware that if you initially declare your class variables separately, you still need to assign objects to them, or they'll contain *Null* values.

Handles

Object variables are often referred to as 'handles'. Take this code, for example:

```
Class LaserGun
        Field temperature:Float
End

Function Main ()

        Local weapon:LaserGun = New LaserGun

        If weapon.temperature < 90
                Print “Zap!”
        Endif

End
```

By way of explanation, we've created a laser weapon and decided, in typical videogame fashion, that it overheats with prolonged use. If you run this, *temperature* will have a default value of zero, so the weapon will fire.

In general, we can think of *weapon* as a *LaserGun* object and treat it accordingly, so if this laser-gun's temperature were to rise to 90 or above, it would fail to fire.

However, in reality, *weapon* is only a *handle* to a *LaserGun* object. Although the object exists somewhere, we really only have a variable that *refers* to it.

Kites and kite handles

Let's create a *Kite* object in order to get a 'handle' on this! We can stretch this analogy pretty far, as you'll see:

```
Class Kite
	Field colour:String
End

Function Main ()
	Local handle:Kite
End
```

OK, so we have a *Kite*-type variable called *handle* but it's not attached to anything; we haven't yet used the *New* keyword to create a *Kite* object. Therefore *handle* has a *Null* value at this point.

In fact, Monkey's *Null* keyword can confirm this for us. Run this and you'll see a message telling you there's no *Kite* object attached to the handle:

```
Class Kite
	Field colour:String
End

Function Main ()

	Local handle:Kite

	If handle = Null
		Print “This handle has no kite object attached!”
	Endif

End
```

So, you're standing there, on a lovely summer's day, holding a kite handle. There's no kite and there's no string to attach to a kite. You just have a handle *suitable* for a kite. That's not much use!

Add this line to the above code, just after the *handle* declaration:

```
handle = New Kite
```

Run it and you'll find there's no message now, which, if you think about it, confirms that the kite handle is no longer *Null* – it's attached to a *Kite* object!

We've effectively built a kite and run a string from the kite to the handle; the kite object is attached to the handle.

Now, the *Kite* class contains only a colour, so all we can do right now with our kite is set that colour. Let's make it red:

```
Class Kite
	Field colour:String
End
```

```
Function Main ()

        Local handle:Kite

        handle = New Kite

        handle.colour = “Red”

        Print handle.colour

End
```

Here's where the analogy admittedly falls apart a little! It reads as if we're setting the colour of the handle. That's only because the *Kite* variable is called *handle* here, for the sake of making a point.

Let's give it a more sensible name:

```
Class Kite
        Field colour:String
End

Function Main ()

        Local mykite:Kite

        mykite = New Kite

        mykite.colour = “Red”

        Print mykite.colour

End
```

This makes more sense, right? It reads as if we're changing the colour of a kite now.

For all the talk of handles attached to kites, you can still think of the renamed *mykite* variable here as if it's a whole *Kite* object, at least after it's been assigned a new *Kite* object – we'd usually consider the handle to be part of a kite anyway.

So, for the purpose of manipulating its values (*colour* in this instance), do think of *mykite* as a *Kite* object, but just be aware that technically it's a handle *attached* to a *Kite* object.