Getting Started with *Monkey*

Welcome to the world of Monkey *games programming!*

What is *Monkey*?

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

It's intended to create games not only for the common PC platforms such as Windows and OS X, but also for more limited platforms such as web browsers (via the Flash and HTML5/Canvas standards), mobile phones and other small devices.

Who made it?

Monkey is the latest in a long line of accessible games programming languages created by Mark Sibly of Blitz Research Limited (BRL), based in New Zealand.

Who is this guide aimed at?

This tutorial 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.

We'll be going through the creation of a simple game in this tutorial, but we need to cover a few basics before we can proceed.

Please note that this tutorial assumes you're running a Windows PC, but the process of creating folders, saving files and running the *Monk* program editor should be similar on other platforms, such as the Apple Mac.

Some code

No doubt you just want to get on with trying out *Monkey* and seeing some on-screen action – a quick demonstration will follow shortly. Before we start, though, here's what a bare-bones *Monkey* program looks like:

```
Import mojo

Class MyApp Extends App

	Method OnCreate ()
	End

	Method OnUpdate ()
	End

	Method OnRender ()
	End

End



Function Main ()
	New MyApp
End
```

Don't be afraid – it may look a little baffling if you've never programmed before, and it might *still* look a little baffling even if you have. That's not a problem, though – you don't need to understand it right now.

The point is to let you get a feel for identifying different *blocks* of code. By *code*, we mean the plain text you see here which defines how a program works – it's a set of instructions for the computer to follow. (*Blocks* simply refers to multiple *lines* of code that 'belong' together.)

The code above is the basic 'framework' upon which we'll base most of the examples in this tutorial, so it's worth taking a closer look.

A simple framework

There are two main parts to this framework: the **App** *class* and the **Main** *function*. Don't worry about what these phrases mean for now, just try to identify these two separate sections in the code. Almost everything we do in this tutorial will start with these two code blocks.

Here are the same two code blocks, separated for easier identification. After the *Import* line, you'll see the *App* class at the top and the *Main* function at the bottom:

```
Import mojo

Class MyApp Extends App

	Method OnCreate ()
	End
```

```
        Method OnUpdate ()
        End

        Method OnRender ()
        End

End

----------------------------

Function Main ()
        New MyApp
End
```

The line *Import mojo* is needed for pretty much every Monkey program – just add it at the very top of every program you write and forget about it.

(Technically, you could write a Monkey program without this line, but in order to use the graphics, audio and input features of Monkey as provided by BRL, you must include the *mojo* 'module'.)

The *App* class starts with the word **Class** and ends with the word **End**. Notice that everything in-between is indented, that is, it's offset to the right so that you can easily see what code belongs inside the class and where the class ends.

The *Main* function starts with the word **Function** and also ends with the word **End**.

You can therefore see that the **End** keyword is used to denote the end of certain 'blocks' of code, such as classes and functions.

Look inside the class block and you'll see three short inner blocks of code, each beginning with the word **Method** and, again, ending with **End**.

The indentation helps us to see where each block of code starts and ends. For example, if the code was *not* indented, it would be much less clear where the class actually ends, and easy to assume it stops at the first instance of **End**. Here's an example of bad code formatting:

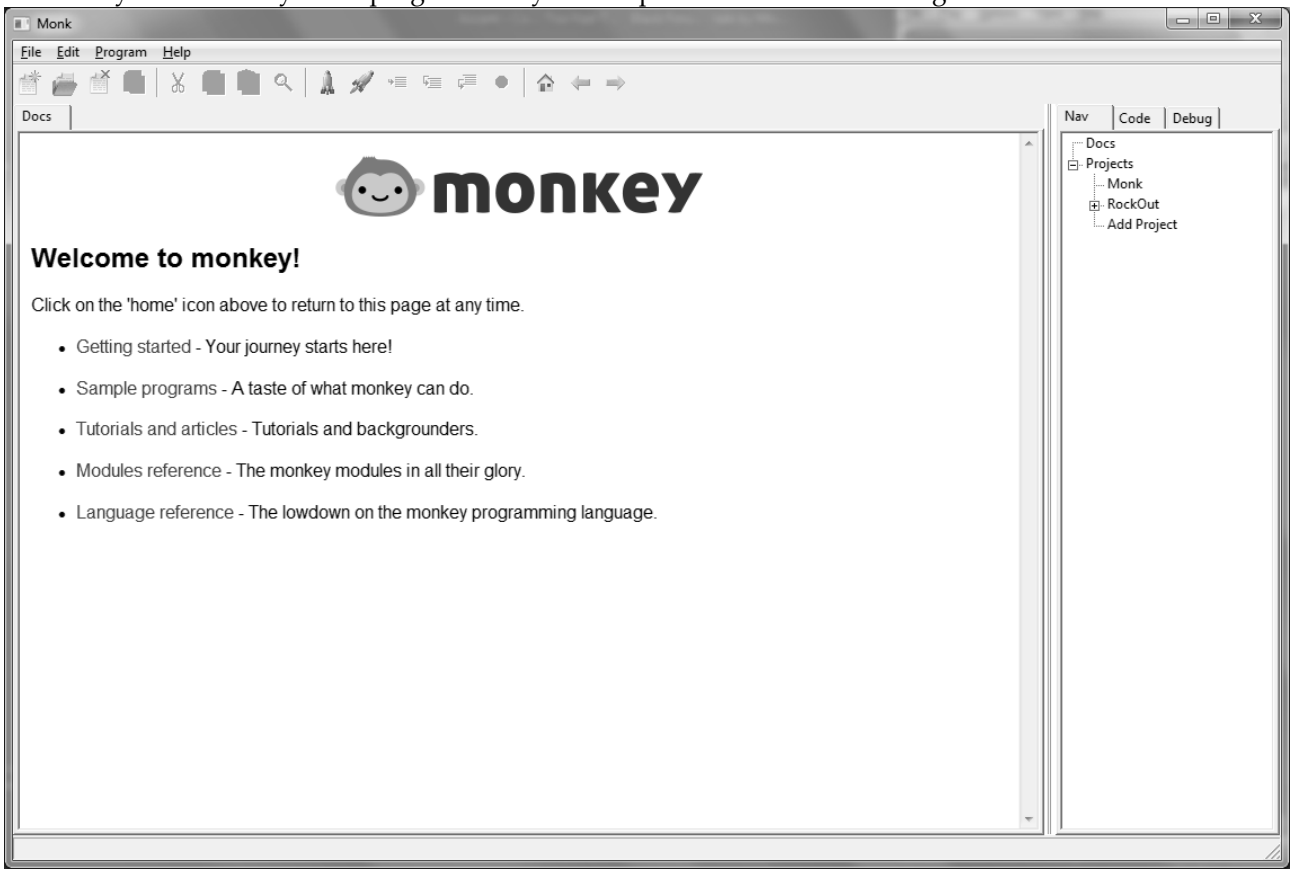
```
Import mojo
Class MyApp Extends App
Method OnCreate ()
End
Method OnUpdate ()
End
Method OnRender ()
End
End
```

Indenting code and adding blank lines improves the readability of a program greatly, as you can see if you compare it to the original.

A quick demo

Open up the *Monk* editor from the main *Monkey* folder. Depending on your computer settings and operating system, it should be listed as something like *monk* or *monk_winnt.exe*.

Run it as you would any other program and you'll be presented with something like this:



The toolbar along the top is used to control *Monk* and it looks something like this:

The individual icons carry out the following actions:

File management

1. New File *Creates a new program file*
2. Open File *Opens an existing file*
3. Close File *Closes the currently selected file*
4. Save File *Saves the current file*

Code editing

5. Cut *Cuts the selected text*
6. Copy *Copies the selected text*
7. Paste *Pastes the selected text*
8. Find *Prompts for a word to find in the current file*

Build/run program

9. Build *Builds the current program*
10. Build and Run *Builds and runs the current program*

Debugger

11. Step *Debug mode only: steps through the program*
12. Step In *Debug mode only: steps into the next function*
13. Step Out *Debug mode only: steps out of the current function*
14. Stop *Stops the program*

Document navigation

15. Home *Goes back to the documentation home page*
16. Back *Navigates to previous visited documentation page*
17. Forward *Navigates to next visited documentation page*

You can alternatively use the various drop-down menus at the top of the *Monk* window to perform the same actions – whatever suits you best.

[TODO: Create tutorials files!]

Click the *Open File* icon (number 2) and run *drawtest.Monkey* from the *Tutorials* folder by clicking on *Build and Run* (number 10). This simple program loads and displays an image – that's all! This is what it looks like:

```
Import mojo

Class DrawTest Extends App

	Field player:Image

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

	Method OnUpdate ()

	End

	Method OnRender ()
		Cls 255, 255,  255
		DrawImage player, 320, 240
	End

End

Function Main ()
	New DrawTest
End
```

You might like to try placing another image file in the same folder as *player.png* and note the name of the file. If you update the file name *between the quotes* (after **LoadImage**) and re-run the code, you should see your image on screen. Use something small or it won't fit!

Tip: If it doesn't work, make sure you leave the brackets and quote marks around the file name; also, make sure you have typed the file name correctly.

What does it all *mean*?

Let's have another quick look at the original code. Again, don't worry about the terminology here if it sounds too complex. After another working demonstration we'll go right to the basics and build from there.

```
Import mojo

Class MyApp Extends App

        Method OnCreate ()
        End

        Method OnUpdate ()
        End

        Method OnRender ()
        End

End



Function Main ()
        New MyApp
End
```

You should be able to see that the code from *drawtest.Monkey* is much the same as the above, only with some working code entered for each method (with the exception of *OnUpdate*, which isn't really needed in this instance). You may notice the addition of a *field*, too, but don't worry about this for now.

So, what do these two blocks of code, the Main *function* and the App *class*, actually do? Well, first of all...

What's a function?

At its simplest, a function contains a block of code, which may be anywhere from one line to potentially several hundred lines long, and all of the code in that block is run when the function is 'called', simply by typing its name. *(This will become clearer later on.)*

This means you can place frequently used code inside a function and only have to type the function name when you want to use that code, rather than typing out the same code over and over again. It effectively turns several lines of code into a re-useable single-word command.

For instance, you may write multiple lines of code to update the positions of all the enemies on the screen. By placing these lines of code in a function called *UpdateEnemies*, for example, you only ever have to type *UpdateEnemies* whenever you want these lines of code to run.

The *Main* function

It's important to note that the *Main* function is a special case – the code in this function is automatically called by *Monkey* when your program starts. **Every *Monkey* program *must* have a *Main* function in order to work!**

Here's the simplest program you can legitimately write in *Monkey*:

```
Function Main ()
End
```

This is the *Main* function, required by every single program you write in *Monkey*. When a *Monkey* program runs, it starts here. Go ahead and type this into the editor.

Of course, as it stands, this does nothing, so let's expand it. Move the edit cursor to the end of the first line and press the *Return/Enter* key; this will add a blank line between **Function** and **End** and put the edit cursor at the start of the new line.

Note, below, the indentation before the word *Print*. This is achieved by pressing the *Tab* key, found on the left of your keyboard. This allows us to easily distinguish the code within the function, and to see where the function starts and ends. Hit *Tab* and type the rest of the line:

```
Import mojo

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

Also, go to the top of the code and add the *Import mojo* line – most programs will require this in reality, so we might as well start including it!

You may even prefer to add more blank lines above and below the code to separate it out even more, but this is down to personal preference:

```
Import mojo

Function Main ()

	Print "Hello world!"

End
```

Anyway, the *Print* command in this instance simply writes a given piece of text, contained within quote marks, on to the screen. Run the program to see it in action. Change the text within the quotes and try it again.

Add another line to print a different piece of text. (You can copy and paste just like you do in any other text

editor to make this quicker and easier.)

So, you can see that the code inside the *Main* function is executed when the program runs, and when it runs out of things to do, it ends!

Technically, you can write an entire program using only the *Main* function, and we'll be doing exactly that to explore some of the core principles of programming shortly, but to get the most out of *Monkey*, we'll need the *App* class.

What's a class?

Actually, that's rather a big question, and one we'll cover in good time. For the purposes of this demonstration, think of it as a collection of related actions, that is, a bunch of actions that an application (hence the name *App*) needs to carry out.

The *App* class effectively controls the overall flow of the program. As you can see, we're using three *methods*, or actions, to control the program; **OnCreate**, **OnUpdate** and **OnRender**.

- the **OnCreate** method is used to carry out certain actions when the program first loads up; typically we would load the game's images and sounds here;
- **OnUpdate** is used to control the game itself, taking player input from the keyboard, mouse, joystick or other device and deciding how far the player will move, whether or not it has collided with an enemy (and what should happen if so), and much more;
- lastly, **OnRender** is used to handling on-screen drawing. This is where you'll decide how the game's display is put together, typically clearing the screen, drawing a background, drawing scenery, enemies and the player, and perhaps adding a score or health display.

When writing any *Monkey* game using the *App* class, we simply fill in the blanks for each method, like so:

```
Import mojo

Class MyApp Extends App

	Method OnCreate ()

		Startup code goes here!

	End

	Method OnUpdate ()

		Game code goes here!

	End

	Method OnRender ()

		Drawing code goes here!

	End
```

```
End
```

If you look at the methods in the image loading example from earlier you'll see it simply implements the instructions given above.

Monkey calls each method at the appropriate time when the program runs, and the code you have written for each one will be executed at that time.

Action!

Now that you know what a *Monkey* program looks like, let's see some action! Load and run *movetest.Monkey* from the *Tutorials* folder.

Here's what it looks like:

```
Import mojo

' Moving an image around the screen using the cursor keys...

Class MoveTest Extends App

	Field player:Image

	Field x:Float
	Field y:Float

	Field xspeed:Float = 4
	Field yspeed:Float = 4

	Method OnCreate ()

		player = LoadImage ("player.png")

		x = 320
		y = 240

		SetUpdateRate 60

	End

	Method OnUpdate ()

		If KeyDown (KEY_LEFT)
			x = x - xspeed
		EndIf

		If KeyDown (KEY_RIGHT)
			x = x + xspeed
		EndIf

		If KeyDown (KEY_UP)
			y = y - yspeed
		EndIf

		If KeyDown (KEY_DOWN)
			y = y + yspeed
		EndIf

	End
```

```
		Method OnRender ()

			Cls 64, 64,  255
			DrawImage player, x, y

		End

	End

	Function Main ()
		New MoveTest
	End
```

You may notice that the very first line here starts with an apostrophe – the ' character. This is a *comment*, and it allows you to make notes within your *source code* (the lines that make up the program).

Anything after an apostrophe is completely ignored by *Monkey*, so the first line here simply explains what the program does.

You can even add comments at the end of a line of code; for example, to explain what the *OnRender* method does:

```
	Method OnRender () ' This is the drawing code!
```

Monkey will only take into account the part before the apostrophe, i.e. *Method OnRender ()* when it executes this line.

You'll also notice several new *fields*. Simply put, these are used to store little pieces of information that are used throughout the program, such as the image used to represent the player and the player's position on screen.

Play about with the above code a little. Perhaps you could change the values after the *xspeed* and *yspeed* fields (currently 4) – these fields are used to control the speed of movement when a key is pressed. You could also modify the *x* and *y* values in the *OnCreate* method, which define the starting position for the player.

You can also change the background colour by modifying the three values after *Cls*. Each one should be a value from 0 to 255; the first value represents the amount of red light in the colour; the second the amount of blue and the third the amount of red.

Some example *Cls* colours are shown below:

```
	Cls 0, 0, 0             ' Black
	Cls 255, 255, 255       ' White

	Cls 255, 0, 0           ' Red
	Cls 0, 255, 0           ' Green
	Cls 0, 0, 255           ' Blue

	Cls 32, 64, 128         ' A subdued blue!
```

When you've had enough of fiddling with the code, and perhaps broken it a few times, stand up, take a

break, make a cup of tea (or your own preferred beverage), then take a deep breath. When you come back we'll get down to the basics!

Variables

In almost any kind of program, we need to be able to store information and also to be able to read and update that information. (A simple example of such information would be the number of 'lives' a player has left.)

Variables are used to store this information, effectively using words (of our own choosing) to represent pieces of data – numbers, words, images, and so on.

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 store this information, but it makes sense to use something relevant:

```
age = 25
```

Here we've created a variable called *age* and given it a value of 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!

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

- only letters, numbers and the underscore symbol can be used in a variable name;
- you can't have any spaces in the name, as this would cause confusion when *Monkey* comes to read the program; this is why the underscore character is allowed, as a substitute. For example, *max health* would not be a valid variable name, but *max_health* would be fine;
- you can't start a variable name with a number; the variable name must begin with a letter, or the underscore symbol. *12gunsound* is invalid, but *_12gunsound* or *gunsound12* are fine;
- you can't use an existing *Monkey* keyword or function name.

So, a variable is essentially a label used to represent a piece of information in your program. Some common uses for variables would be:

- the player's position on the screen, which we would update depending on the player's input via the keyboard, mouse or other input device;
- the player's health, which we might decrease depending on damage inflicted by enemy shots, collisions, etc;
- we might also increase the player's health on picking up a health boost, but want to limit it to a maximum value, so this maximum value could be stored in the *max_health* variable from our earlier example.

Here's a runnable example, but there are two details we haven't covered yet that you might notice here: the special 'keyword' *Local* and the suffix *:Int* on the end of each variable name. Don't worry about their meanings at this point.

For now, all of our variables will be *declared* with the keyword *Local* and we'll add *:Int* to the end of each one. (To *declare* a variable means that we're announcing our intention to *Monkey* that we'll be using this name to represent some data.)

```
Import mojo

Function Main ()

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

	Print age
	Print energy
	Print ammo

	Print 100

End
```

Note that we don't use the *:Int* suffix after the variable has been declared, just the variable name itself.

Run this and you'll just see a list of numbers, but you should see that each one matches the values specified above. Try changing the values and checking the resulting output.

Note that we're using Print *again, but that what we're asking it to display isn't contained in quotes this time. We use quotes to print specific pieces of text, but when there are no quotes we're either printing the contents of a variable – the value it represents – or a specific value, which is what the* Print 100 *line does here.*

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

Create some variables of your own – remember to use *Local* and *:Int* as for the previous examples – and add a *Print* line for each one. At this point you should *only* store *whole numbers* in your variables, or you'll run into problems!

Basic variable types

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

- *Integers* store whole numbers;
- *Floats* store fractions;

- *Strings* store text.

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

In general, to declare a variable, you use the keyword *Local*, the name of the variable, and add the relevant suffix. Here are some examples:

Keyword	Name	Suffix	Example
Local	apples	:Int	*Local apples:Int*
Local	elephants	:Int	*Local elephants:Int*
Local	lives	:Int	*Local lives:Int*

The three basic variables types are:

Integers

Suffix: Int
Example values: 0, 100, 5000, -99

For our purposes, integers are simply whole numbers.

When you create an integer variable, or *int*, you can assign any number to it, from −2,147,483,648 to +2,147,483,647.

This means that an *int* has a range of about four thousand million, and not many games will need numbers outside of this range! (There are, however, other variable types to deal with this if necessary.)

If you don't set a value for an *int* when you declare it, as below, then *Monkey* will automatically assign it a value of zero. You can see this by running the demo below:

```
Import mojo

Function Main ()

	Local bullets:Int

	Print bullets

End
```

As in the earlier example, we've used the keyword *Local* to declare a variable, that is, to tell *Monkey* we're going to be using a variable named *bullets* to store a value.

As we've added the suffix *:Int* to the variable name, we're also telling *Monkey* that this will be an integer variable, i.e. one that's used to store whole numbers.

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 English, the first line inside the function reads: "*bullets* equals ten."

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

Floats

Suffix: Float
Example values: 0.1, 0.75, 2000.075, -100.55

Simply put, *floats* are fractions, or numbers including fractions. In short, any number including a decimal point – numbers that *aren't* whole, such as a half (0.5), a quarter (0.25), and so on.

The word *float* comes from *floating point number*. You may be surprised to learn that your computer's processor (it's "brain") only knows how to handle whole numbers. No, really!

Floating point representation is the technical term used to describe a way of *approximately* representing fractions in a system that can only deal with whole numbers.

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

This means they are not accurate enough for strict scientific or mathematical usage, but certainly good enough for most games, as the errors are minuscule.

(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 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
```

As with integers, if you don't assign a value when declaring the variable, it'll be assigned a default value of zero.

Here's another demonstration, this time assigning a value to each variable at the time of declaring it:

```
Import mojo

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, 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 we're using an integer, we can't then give it a fraction to work on; 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.

Strings

Suffix: String
Example values: "Hello world!", "The quick brown fox jumps over the lazy dog", "a"

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!"*, just as *Print 10* contains an integer (10) and *Print 0.5* contains a float (0.5).

When strings and numbers are used directly like this, without being assigned to a variable, they're called literals*:* string literals, integer literals *and* float literals*. They are also often referred to as* hard-coded *strings or values, as they cannot be changed while the program is running.*

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:

```
Import mojo

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 + symbol, so we can create some more interesting output:

```
Import mojo

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:

```
Import mojo

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:

```
Import mojo

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:

```
Import mojo

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:

```
Import mojo

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.

Variable assignment

Variable assignment is something we've already done a good number of times now; for example, *intro = "Hello world"*, or *a = 100.*

We're taking a string variable, *intro*, and *assigning* it a value, *"Hello world"*; or assigning the value *100* to the variable *a*.

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, so *sausages* will contain the value *150*.

When we say *result = calculation (*or *sausages = 100 + 50*) we're assigning the *calculation*, the part *after* the equals sign, to the *result*.

TODO: Tidy this up!

Any time we use the equals sign to say *a = 1 + 2*, or *dinner = sausages + beans*, or even *greeting = "Hello" + "*

world", *Monkey* will first work out the part *after* the equals sign, then place the result of this calculation into the part *before* the equals sign.

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 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.)

```
Import mojo

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 assign the value 5 to *apples*;
- we assign the calculation *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).

That last variable was declared *on-the-fly*, by the way, just to show that you can do this: we assigned the

result of *fruit plus 100* to *produce* in the same line as *produce* was declared, whereas with the previous variables we declared them first.

You could have declared *produce* and *fruit* at the top with the other variables, but sometimes it's just more convenient to do it as you go along.

It does make for better code layout to declare a bunch of variables together before you use them, as it makes it easier to see what's going on when you later revisit your code, but declaring them on-the-fly like this is still valid (and oh-so-convenient!).

Mathematical operations

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

```
Import mojo

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 raw numbers, of course:

```
Import mojo

Function Main ()

        Print 1 + 2 + 3 + 4 - 1

End
```

The result is 9, as you can no doubt work out on your own.

You already know that these whole numbers could have been assigned to integer variables. This is exactly the same program, except each number is first assigned to an *int*:

```
Import mojo
```

```
Function Main ()

	Local a:Int = 1
	Local b:Int = 2
	Local c:Int = 3
	Local d:Int = 4
	Local e:Int = 5

	Print a + b + c + d - e

End
```

(These variables could still be named as words rather than letters if you preferred, and, outside of simple examples like this, usually would be.)

How about combining all of these into one variable, as we did when combining strings?

```
Import mojo

Function Main ()

	Local a:Int = 1
	Local b:Int = 2
	Local c:Int = 3
	Local d:Int = 4
	Local e:Int = 5

	Local result:Int

	result = a + b + c + d - e

	Print result

End
```

Here we've declared an integer, *result*, and assigned the combined values of all the other variables to it. The *result* variable in the end contains the value *9*, the sum of all the other variables.

This means you can perform a calculation once and simply use the 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, 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:

```
Import mojo
```

```
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 that's not going to end up as a round number.

Making values negative

You can also make numerical values negative by prefixing them with a minus sign:

```
Import mojo

Function Main ()

	Local a:Int = 100

	Print a
	Print -a

End
```

Here's a slightly trickier example:

```
Import mojo

Function Main ()

	Local a:Int
	Local b:Int
```

```
        a = 100
        b = -a

        Print a
        Print b

End
```

In this example, we first set the value of *a* to *100*. We then make variable *b* equal *-a* ("negative a" or "minus a"). Since *-a* means *minus 100* here, *b* will end up with the value *-100*.

Doing it the other way around, that is, taking a variable which is holding a negative value and putting a minus sign in front of it, will create a positive value:

```
Import mojo

Function Main ()

        Local a:Int = -100

        Print a
        Print -a

End
```

Hard-coded values in calculations

This section is really just for your future reference; don't worry if it seems a little complex right now. Just skip it if you find it hard-going!

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 *fraction = 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 uses a temporary float variable to store the result instead.

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, 2.5, will be stored correctly as a float before then being assigned to the *fraction* variable. (We've specified both numbers with *point-zero* just for consistency and easy recognition as floating point values.)

Variable modification

Modifying a variable's contents is something you will do quite often; for example, updating a player's position on screen, increasing a score or decreasing a player's ammunition. We do this by performing a calculation involving the variable in question, then assigning the result back to the variable.

You may well be asking: *"What on earth does that mean?!"* Well, let's assign a variable, *lives*, to itself:

```
Import mojo

Function Main ()

	Local lives:Int = 1

	Print lives

	lives = lives

	Print lives

End
```

The value of *lives* at the end is the same, of course. We set lives to 1, print it, make lives equal lives, and print it again.

When we write *lives = lives*, it looks like we're saying *lives equals lives*, which seems rather silly; that's like saying that 1 equals 1. Duh! Of course it does! However, this is not what's being said here.

A statement of the form *result = calculation* means we are performing the *calculation* (the part after the equals sign) and then assigning the outcome to the variable *result* (the part before the equals sign).

In the case of *lives = lives* it's pretty pointless, but what about *modifying* the *lives* variable and *then* assigning the result to *lives*?

lives = lives + 1

Let's say that *lives* starts out with a value of 3.

To ease understanding, 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 of 4 is stored in the *lives* variable. The calculation *lives = lives + 1* therefore increases the value of *lives* by one.

This is a real-world example, too – you can literally perform this calculation when the player collects a 'power-up' to increase the number of 'lives' available:

```
Import mojo

Function Main ()

	Local lives:Int = 1

	Print "Lives: " + lives

	' Got power up!

	lives = lives + 1

	Print "Lives: " + lives

End
```

There's no reason you can't modify a variable in other ways, such as multiplying its value by something else, or, in the case of the player losing a life, subtracting 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 point is that modifying a variable is as simple as:

variable = calculation involving variable

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

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 looking at a value and taking an action based on that value. In practise this usually means looking at 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"

You can probably tell how this works even without any explanation – it's almost plain English!

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:

```
Import mojo

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.

When it comes to 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, so when the program comes to check the value of *lives* and find it's *not* zero, the action following *Then* is simply ignored.

The *If/Then* statement takes the form:

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

What if you want to take an action based on the *something* not being true? You use the *Else* keyword:

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

Taking the previous example:

```
Import mojo

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 *not* zero when the program checks it (it's one), the action following *Else* is carried out instead, and it prints *Still Alive!*

The *If/Then* statement is useful for quick tests like this, but it's very limiting if you need to take many actions based on a given result. *Monkey* lets you string together what would normally be multiple lines by using the semi-colon character to separate them:

```
Import mojo

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
```

Although you can easily do this sort of thing...

***If** lives = 0 **Then** money = money – 1000; ammo = 0; Print "Game Over!"; Print "Try Again!"*

… it's really too complex to read easily, or to change later on. For anything more complicated than a single action, we use *If/EndIf*:

__If__ lives = 0

money = money – 1000
ammo = 0
Print "Game Over!"
Print "Try Again!"

__EndIf__

Now it's much easier to locate and read the individual actions being carried out, which makes it much easier to modify things later.

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

Let's see this in action:

```
Import mojo

Function Main ()

	Local lives:Int = 0

	If lives = 0

		money = money - 1000
		ammo = 0

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

	EndIf

End
```

Run this and the inevitable happens – the block of code is executed as expected.

However, 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* statement is being skipped because *lives* is no longer zero.

Let's add some actions to be taken if the value of *lives* is *not* 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:

```
Import mojo

Function Main ()

	Local lives:Int = 3

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

	If lives = 0
```

```
                money = money - 1000
                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.

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 a equals 100 then do something

We'll also need to know the exact opposite in many situations:

If a does not equal 100 then do something

How about if one value is less than another?

If a is less than 100 then do something

Or greater than another?

If a 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 a 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 purposes, there are only three symbols to remember:

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

Let's see them in action:

```
Import mojo

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* to see the different results. Check the code to see which line has been executed, and therefore which comparison was true.

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*:

```
Import mojo

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 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:

```
Import mojo

Function Main ()

	Local a:Int

	If a <= 10 Then Print "a is less than or equal to 10"
	If a >= 10 Then Print "a is greater than or equal to 10"

End
```

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

Run the program, then try giving the *a* variable a value of ten (it starts at the default value of zero, obviously less than ten), then give it a value more than 10. You might have to think a little more than before about the output you're seeing!

In summary, the valid combinations of these symbols are:

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

Too many decisions!

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

If/Then/Else is fine for simple *one-way-or-the-other* decisions, but what if you want to test for a range of values and perform different actions for each?

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

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 one, it will run the code directly after *Case 1*; if *bananas* equals two, it will run the code after *Case 2*; and so on.

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

Importantly, 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 change the value of bananas to *two* and *three*:

```
Import mojo

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 following *Case* blocks.

A little exercise: add a case for the value of *bananas* being zero. Print something funny relating to a 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 based on the situation where *none* of the *Case* values matches the value of *bananas*? Luckily, *Monkey* provides the *Default* keyword for just this eventuality:

```
Import mojo

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."

	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.

(You can add a case 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, perhaps perform a few tests and then very quickly come to an end.

Most real-world programs operate in a loop, repeating the same block 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 a real-world *Monkey* version:

```
Import mojo
```

```
Function Main ()

	Repeat
		Print "Hello"
	Forever

End
```

(A real-world program would be doing much more than this, but we'll come to that shortly.)

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 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 do this is with the *Exit* keyword, whenever a certain condition is met:

```
Import mojo

Function Main ()

	Local a:Int

	Repeat

		a = a + 1
		Print a

		If a = 100 Then Exit

	Forever

	Print "Exited from loop!"

End
```

This loop will keep increasing the value of *a*, check whether or not it equals 100, then exit the loop if that condition is met.

Another way to do this is with the *Repeat*/*Until* loop, 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:

```
Import mojo

Function Main ()

	Local a:Int

	Repeat

		a = a + 1
		Print a

	Until a = 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 *a* equals 100". It starts out with a value of zero, then on each *iteration* of the loop, *a* is increased by one. When *a* equals 100, the program exits the loop. (To *iterate* through a section of code means to repeat it.)

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

```
Import mojo

Function Main ()

	Local a:Int

	While a < 10

		a = a + 1
		Print a

	Wend

	Print "Exited from loop!"

End
```

This also repeats a section of code, but places the test at the start of the loop, whereas *Repeat/Until* 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, *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.

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

The Game Loop

A typical *game loop* tests for player input from the keyboard, mouse or other controller, updates the player's position on screen based on this input, updates enemy positions on screen, checks for collisions between the player (or the player's bullets) and the enemies, and draws everything on screen, many 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). *Monkey* requires you to fill in the code to be run when it calls the *OnUpdate* and *OnRender* methods:

- **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 this process a little bit, but behind the scenes, and very simplified, *Monkey* does something like this:

```
Repeat

        OnUpdate' Your game update code
        OnRender' Your drawing 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.)

Functions

Functions make it easy to bundle frequently-used code into a single 'command'. Take this example:

```
Import mojo

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 at various points throughout your program – an unlikely scenario, but nice and simple!

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

```
Import mojo

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

        If a = 999
                Print "Hello!"
                Print "Hello again!"
                Print "Hello, for the last time!"
        Endif

End
```

Pretty tedious – and messy! How about this?

```
Import mojo

Function Main ()

        Local a:Int = 100

        If a = 100
                PrintStuff ()
        Endif

        If a = 101
                PrintStuff ()
        Endif

        If a = 102
                PrintStuff ()
        Endif

        If a = 999
                PrintStuff ()
        Endif

End

Function PrintStuff ()

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

End
```

In reality, we'd probably use *Select* here to choose from the various values for *a*, but the point is to show the calling of the function *PrintStuff* for each possibility, rather than typing out those three *Print* statements in full every time.

I've placed the *PrintStuff* function below the *Main* function in this example, but there's no reason it can't go above it instead. 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 point being that the program doesn't just run through each one in turn!)

Notice that the repeated code has now been moved into the *PrintStuff* function. Now each time we want to run those three lines of code, instead of typing them out, we just 'call' *PrintStuff*.

As with variables, the name you give 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 easy to remember as possible.

Function parameters

Functions can take *parameters*, that is, values that modify how they work. Let's say that instead of the word "Hello" in each case, you want the above example to print "Goodbye" in some cases. We can *pass* a string value as a parameter, like this:

```
Import mojo

Function PrintStuff (message:String)

	Print message

End
```

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

We've added a parameter, *message*, in between the brackets (more correctly called parentheses, but most people call them brackets, so we will). In this case, it's a string-type parameter.

When you add a parameter of any type, it's similar to declaring a new variable that *only the code within the function can access*. The point of a parameter, though, is to accept a (changeable) value when the function is called:

```
Import mojo

Function Main ()

	PrintStuff ("Hello")

End

Function PrintStuff (message:String)

	Print message

End
```

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

The *message:String* parameter will *receive* the value you passed when calling *PrintStuff*, so in the original example, the *message* string within *PrintStuff* will be "Hello". Change the value passed, and *message* will contain that value instead. The *Print* command then displays whatever string value you passed.

Parameters are effectively variables that receive the values *you* pass to them when the function is called.

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

```
Import mojo

Function Main ()

	Local a:Int = 100

	If a = 100
		PrintStuff ("Hello")
	Endif

	If a = 101
		PrintStuff ("Hello")
	Endif

	If a = 102
		PrintStuff ("Hello")
	Endif

	If a = 999
		PrintStuff ("Goodbye")
	Endif

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 999, the messages displayed by *PrintStuff* will change. As simple as this example may appear, it does show that you can change the effect of a function call by passing different values.

Let's pass an integer value to a function, and we'll use that value to print different messages depending on what value is passed:

```
Import mojo

Function Main ()

	DoStuff (1)

End

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
```

So, we pass a value of one when calling the *DoStuff* function, and then, inside the *DoStuff* function, the *action* parameter receives that value.

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

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

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

```
Import mojo

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 shot image to be used when the player fires, and 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 you might call *Fire* every time the player presses the *Space* key.

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*:

```
Import mojo

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.

Optional parameters

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

```
Import mojo

Function Main ()

	CountBullets ()

End

Function CountBullets (ammo:Int = 100)

	Print "Bullets: " + ammo

End
```

Notice that the output from our call to *CountBullets* tells us we have 100 bullets, even though we passed no parameters. The default value is defined in the *ammo* parameter of *CountBullets*, so if we choose to pass no value for *ammo*, a default value of 100 will be used.

The *ammo* parameter, because it *has* a default value defined, is optional, meaning you don't have to specify it when calling *CountBullets* (in which case it will contain the default value within the function).

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'

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)

… but this is – note that the parameters are the same, just in a different order:

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

Any parameters with default values *must* be defined *after* those that *require* values to be passed. Hence, *gun_type* here must come before *ammo*.

Here's a more complicated version; the meanings don't matter here, but notice how the parameters that need you to pass values are defined first, and those with optional 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, you *must* pass the first two parameters as they have no default values.

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 ten:

```
Import mojo

Function Main ()
	Local years:Int = 20
	PrintAgeInTen (years)
End

Function PrintAgeInTen (age:Int)
	age = age + 10
```

```
        Print age

End
```

That's easy enough; the PrintAgeInTen function takes the value you pass to it – 20 in this case – and prints the result.

But what if you want to do something 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:

```
Import mojo

Function Main ()

        Local years:Int = 20

        Local result:Int = GetAgeInTen (years)

End

Function GetAgeInTen: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 *GetAgeInTen*. You can then do what you want with this result.

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

We know we're looking for a result that tells us the player's current age, plus ten years, so we've created a new variable called *result*. (You could call this 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 bullets:Int = 5*), and that's what we're doing here.

The only difference is that instead of giving it a hard-coded value, or the contents of an existing variable (e.g. *Local bullets:Int = ammo*), we're assigning the *result* of a function call.

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

```
Import mojo

Function GetAgeInTen:Int (age:Int)

        age = age + 10

        Return age

End
```

The first line looks like the previous examples, except we now have an *:Int* suffix after the function name.

This function is intended to perform a calculation and return a result. 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.

In this case, the *GetAgeInTen* function returns an integer parameter – this is what the *:Int* suffix after the function name tells us.

The *age = age + 10* line simply takes the parameter, *age*, and adds ten to it.

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

```
	Local result:Int = GetAgeInTen (years)
```

The *result* variable here will receive the value *returned* from *GetAgeInTen*.

[TODO: … skipping ahead a bit!]

Classes and Objects

You've briefly encountered classes; remember reading about the *App* class at the beginning of this guide? It looked like this:

```
	Class MyApp Extends App

		Method OnCreate ()
		End

		Method OnUpdate ()
		End

		Method OnRender ()
		End

	End
```

In this case, the class contains a set of actions (*methods*) that an application needs to perform. We'll come to methods shortly, but we'll start with a very simple class and build from there:

```
	Class Rocket
		Field fuel:Float
	End
```

Here we've defined a *class* with the name *Rocket*. You can give a class almost any name you like, just as you do with variables, but it should of course describe the object you're defining. In this case, we're going to

define the features of a rocket, so we've called it *Rocket*.

At its simplest, a class is a collection of variables that store the basic features of a rocket; so far, we have fuel, and we'll add more as we go on.

The *field* you see here is really just another Float variable and works much the same way as any other variable. However, the variables describing a class are declared with the *Field* keyword instead of the *Local* keyword you're used to as there are subtle differences.

Let's see this class in action:

```
Import mojo

Class Rocket
	Field fuel:Float
End

Function Main ()

	Local player:Rocket = New Rocket
	player.fuel = 100.0

	Print player.fuel

End
```

OK, a few new things here, so let's go through them: we have the standard *Main* function and we have our *Rocket* class from before.

Within the Main function, we have this line:

Local player:Rocket = New Rocket

Taking the part before the equals sign, we have what looks like any normal variable declaration: the *Local* keyword, the name of the variable, and the variable's type.

In this case, the class effectively creates a new variable type, and we're using that.

After the equals sign we have something new; rather than assigning a number, or the result of a calculation, we're assigning the player variable the result of this statement:

New Rocket

It's as simple as it looks: we're creating a new *object* that has the properties of the *Rocket* class.

The new *Rocket* object is then assigned to the *player* variable. If that's a little confusing or unclear, don't worry – analogy forthcoming! Let's just finish looking at this example.

Now, we've effectively created a variable called *player* which has the properties described in the *Rocket* class; we refer to such a variable as an *object*, because it describes a real-world 'thing' that has a specific set of features – so it's an object!

Let's say we want to set the amount of fuel in our rocket ; we know that the *Rocket* class contains a field (a variable) called *fuel*, and we know that this is just a normal floating point variable.

To interact with normal variables, we just use the variable's name; to read or modify the fields of an object, we state the object's name, add a dot, then the name of the field (that is, *object.field*).

Other than that, we treat a field like any other variable, so we can assign it a value, as in the example, and use Print to show the contents of the variable:

```
player.fuel = 100.0
Print player.fuel
```

[MORE TO COME!]