

4

More power, less script

What we'll cover in this chapter:

- *Using loops to make Flash repeat actions*
- *Several different kinds of loop*
- *Using loops with arrays to work on lots of information in one go*
- *Building a hangman game with Flash and ActionScript*

4 Foundation ActionScript for Macromedia Flash MX

It's not exactly headline news, but computers are pretty dumb. Whatever far-fetched dreams we may have once had, the 21st Century isn't buzzing with the conversation of HAL-9000s— well, not yet at least! It's actually chock full of silicon number crunchers, and (sorry to say) even your shiny new top-of-the-range PowerMac can't hold a candle to the raw intelligence of the simplest human being.

When it's a matter of crunching through commands and figures though, the computer wins easily, both in terms of speed and accuracy. It's specifically designed to work on large, complicated jobs by breaking them down into gazillions of tiny instructions, and getting them all done in the blink of an eye – so it's perfectly suited to repetitious jobs.

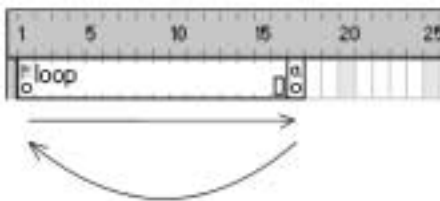
By and large though, it's human programmers who have to tell a computer what to do when, and they just don't have time to write out *every one* of those gazillions of instructions – especially when there are reruns of Star Trek to be watched on TV! And thus it was that **loops** came into being.

In Flash, a loop is a command (or collection of commands) that makes your movie do something (or collection of somethings) more than once over...

Don't worry if this statement doesn't immediately make the whole world seem like a better place to live in – it's just a definition of the word 'loop'. I'll spend the rest of the chapter persuading you that loops aren't just useful to Trekkies, but can save *you* lots of time and effort, and open up some very cool possibilities...

Timeline loops

The simplest way to make a loop in Flash is by using `gotoAndPlay()` to repeat a section of the timeline. Say we label frame 1 as 'loop', and attach `gotoAndPlay("loop")` to one of the later frames:



When this timeline plays, it'll play normally as far as frame 17, see the `gotoAndPlay()` action, and jump back to frame 1. Then it will play through frames 1-17 again, before it runs the frame 17 actions again, jumps back to frame 1, and plays from there until it reaches frame 17 and so on, and so on, and so on...

So, this will set up a loop between frames 1 and 17, and Flash will quite happily let this repetition go on indefinitely. Of course, you might well be thinking, "why bother with a `gotoAndPlay()` action? Flash normally repeats the timeline anyway." Well there are lots of very subtle differences we could talk about, but one real biggie that makes the rest redundant:

- An automatic timeline loop is simple and dumb. It plays the whole timeline. It plays the whole timeline again. It plays the whole timeline again. And so on. What's more, any old user can tweak the player settings to turn it off – no good if your movie depends on it...!
- A `gotoAndPlay()` timeline loop puts you, the author of the movie, in control of the looping. You can specify where the loop starts and where it stops, so it doesn't need to loop the entire timeline; you can even have several loops on the same timeline! You can use it within movie clips too. Even better, none of those pesky loop-stopping users can break it!

Timeline looping can be a very useful technique for getting the most out of what's already there on the stage, but it doesn't really give us much of the value-added interactive power that ActionScript promises.

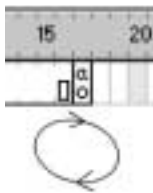
The real subject of this chapter involves another, quite different approach to looping. Instead of repeating a sequence of frames on the timeline, let's see how we can repeat a set of actions.

ActionScript loops

Loops really come into their own when we start using them to repeat scripts that we've already written. Maybe you have a great long list of variables that all need to be tweaked in just the same way, or lots of movie clip instances on the stage that all need to be shifted one pixel to the left. It's easy to write an action to change one of them, and not so hard to write a couple more. By the time you've written four hundred and twenty-eight though, you might be starting to flag a little...

If, on the other hand, you write out your action once and stick it inside an ActionScript loop, you can tell the loop to repeat that action as many times as you like. Yes, it's time to capitalize on the fact that your computer will never get bored doing simple repetitive jobs!

An ActionScript loop is a totally different beast from a timeline loop. A timeline loop involves playing through a sequence of frames, whereas an ActionScript loop exists entirely within a single frame. The ActionScript loop is defined in the lines of script you've attached to that frame, so Flash will stay on the same frame for as long as those lines are looping.



ActionScript has several looping actions, specifically designed to help us out with this sort of thing. They all use the sausage-like notation we saw in the last chapter, so they hopefully won't look too alien to you!

4 Foundation ActionScript for Macromedia Flash MX

while loops

A **while loop** is probably the simplest of all ActionScript loops to understand, since it's really not much different from a simple `if` statement:

```
while (condition) {  
    do these actions;  
}
```

In fact, the only thing that *is* different from an `if` statement is the name of the action. Well, that and the end result of course! We give it a condition (something that will *always* turn up either true or false) and some actions to perform **while** the condition's true. Unlike in an `if` statement, Flash won't stop and move on once it's run the actions – it'll perform them over and over and over for as long as the condition's true. It'll only move on to the next line of script (or the next frame) when the condition turns out to be false.

That means that something like this...

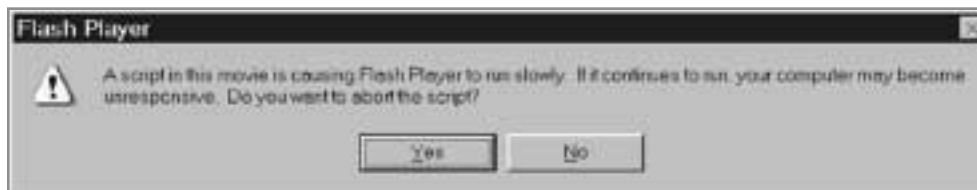
```
while (false) {  
    // actions  
};
```

...won't do a thing. The condition is always false, so any actions we put in the body will never run. On the other hand, something like this...

```
while (true) {  
    // actions  
};
```

...will kill your movie stone dead. The condition is always true, so the loop goes on forever!

You can try making this happen if you like, but be warned: your computer may lock up for a while when Flash tries to compile the SWF. When Flash spots your mistake, it will show you the following message:



If you do ever see this dialog, I strongly suggest hitting Yes. Although Flash can detect these 'forever' loops, they can still *crash your computer* if you press No and let them run – I've seen the evidence for myself.

*Any ActionScript loop that repeats indefinitely can be described as a **forever loop**. These will normally only ever happen if something's gone wrong.*

Useful things to do with while loops

So how can we do something useful with a `while` loop? Simple: base the condition on a variable, and put actions in the body of the loop that can change that variable. Here's a simple example:

```
myCounter = 0;
while (myCounter < 10) {
    myCounter++;
};
```

We set up a variable called `myCounter`, giving it a value of 0. When we start the `while` loop, the condition we give it basically says: 'the value of variable `myCounter` is less than 10'. Well, at this stage, that's most certainly true, so we run the actions inside the body of the loop.

There's only one action in the loop body, and while it may look a little odd, it's really nothing special: `myCounter++` is just a shorthand way of saying 'add one to the number stored in `myCounter`'. So that's just what Flash does, just before it reaches the end of the loop.

Since we're still working through a `while` action, we loop back to the start. `myCounter` now has a value of 1, so `myCounter < 10` is still true. We therefore run `myCounter++` and loop back to the start again. And again. And again. And this keeps happening until the value of `myCounter` reaches 9. At that point, we add 1 to the value, loop back to the start, test whether the new value is less than 10 – it's not! Now the condition is false, so Flash can stop looping and move on.

Of course, this just gives us the bare bones of a useful loop: it's not terribly exciting if it just adds lots of 'one' to a boring old number variable... Let's flesh things out, so that our simple `while` loop actually does something practical in a movie.

Making actions run several times in a row

This quick example is going to demonstrate a few of the things it's possible to do with a simple `while` loop.

1. Create a new Flash document and select frame 1 on the main timeline. Open up the Actions panel, and click on the pin button to pin it to the current frame. Now add the following script:

```
myCounter = 0;
while (myCounter < 10) {
    // actions to loop go here
    myCounter++;
};
```

4 Foundation ActionScript for Macromedia Flash MX

This is our basic loop, which will run a complete ten cycles.

2. The simplest thing we can do with this is to trace out the value of `myCounter` for each loop:

```
myCounter = 0;
while (myCounter < 10) {
    // actions to loop go here
    trace(myCounter);
    myCounter++;
};
```

Run the movie, and sure enough, this is what we get:



This reminds us that `myCounter` takes values of 0, 1, 2, and so on up to 9, before the loop stops (when it hits 10). It's important to remember that we never actually *see* the value of `myCounter` traced as 10 because the loop stops as soon as it gets that value.

3. This business of counting from 0 to 9 instead of counting from 1 to 10 should remind you of something we saw back in Chapter 2: arrays.

As you may recall, a single array can hold lots of different variables – the **elements** of that array. We can tell Flash which element we want to work with by specifying a particular offset number (or **index**). If we use a variable to specify the offset, then the element we deal with depends on the variable's value. Then we can pick and choose from the array's elements by changing the value of that offset variable... You still with me?

Anyway, the upshot of this is that we can write generalized code for manipulating array element number `i`, and specify the value of `i` quite separately. If we happen to be looping through all possible values of `i`, this approach can be *very* powerful!

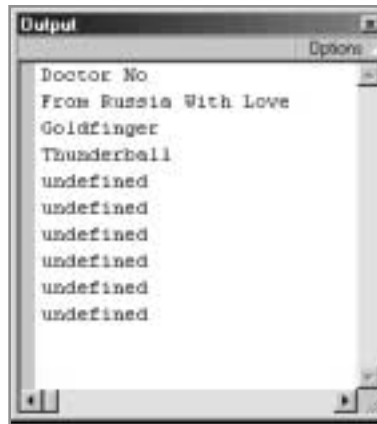
So let's try using `myCounter` as the offset value, used to specify one particular element in an array.

```

bondMovies = new Array("Doctor No", "From Russia With Love",
    ↪"Goldfinger", "Thunderball");
myCounter = 0;
while (myCounter < 10) {
    // actions to loop go here
    trace( bondMovies[myCounter] );
    myCounter++;
};

```

Now we get something more like this:



Instead of just showing the counter value, we're using it to show the first ten values from our `bondMovie` array. The only problem is that there are only four entries.

4. There's no point in looping ten times if there are only four entries in the array. In fact, we only need our loop to loop for as long as the array element we're looking at has a well-defined value. In other words, we should be looping 'while `bondMovies[myCounter]` is not undefined'. Sounds like we need to change our loop condition:

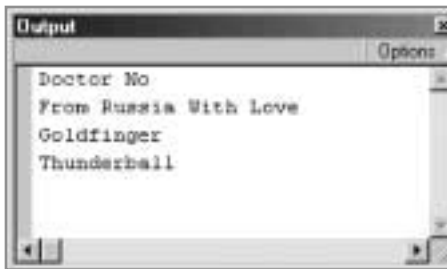
```

bondMovies = new Array("Doctor No", "From Russia With Love",
    ↪"Goldfinger", "Thunderball");
myCounter = 0;
while ( bondMovies[myCounter] != undefined) {
    // actions to loop go here
    trace( bondMovies[myCounter] );
    myCounter++;
};

```

Run the movie once again, and you'll see that this has done the trick

4 Foundation ActionScript for Macromedia Flash MX



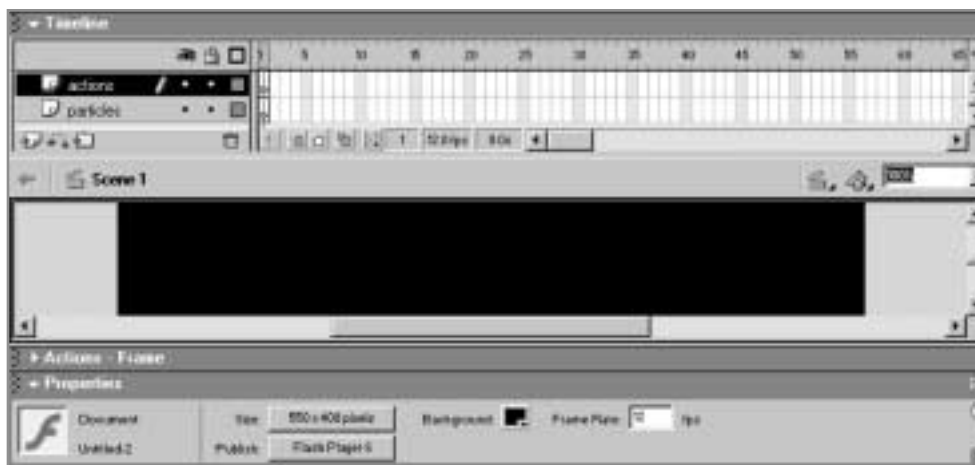
As soon as `myCounter` reaches 4, the while loop spots that `bondMovies[myCounter]` isn't defined, so the loop cuts out before it gets a chance to trace the word 'undefined'.

Okay, this may seem like pretty cool stuff if you happen to enjoy making lists of movie titles. But what about doing some real Flash stuff?

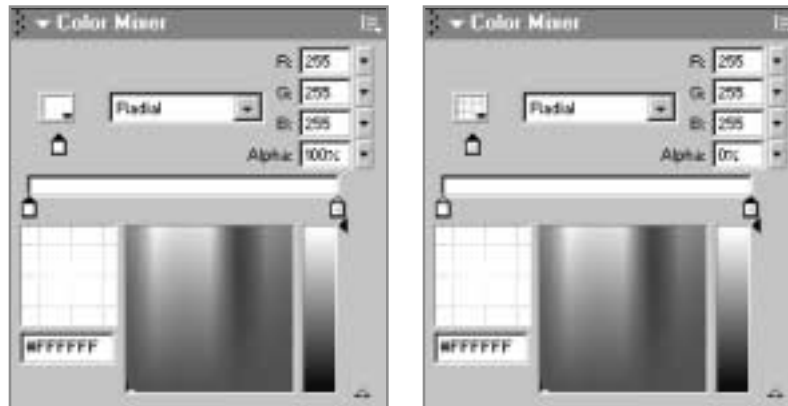
Using loops to control what's on the stage

This time, we're going to use a similar loop to put a whole set of movie clips on the stage. Along the way, I'll introduce a couple of useful tricks for controlling movie clips that we haven't seen before.

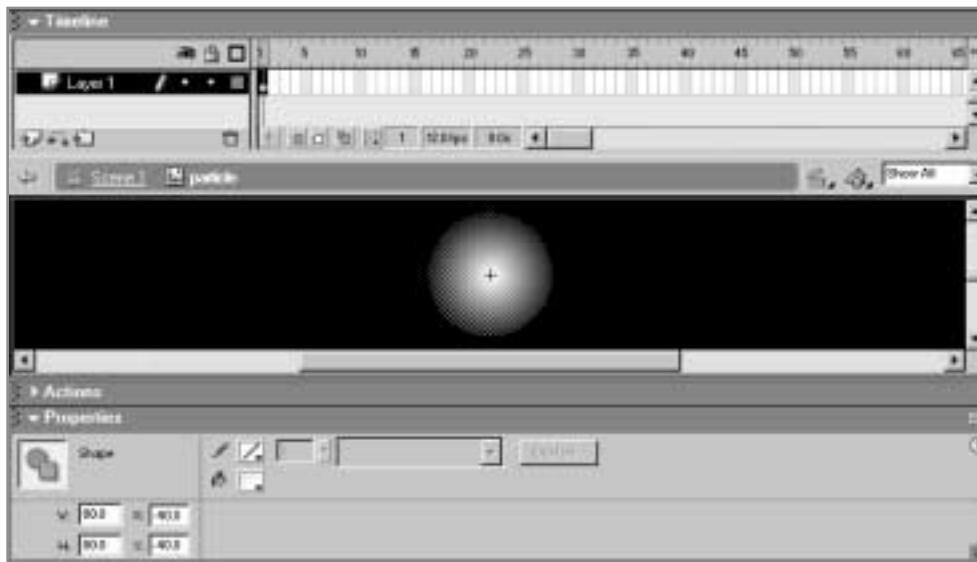
1. Create another new Flash document, make sure the stage is active and use the Property inspector to give it a black background. In the Timeline panel, rename Layer 1 as particles, and create a new layer above it called actions:



2. Select the particles layer and use Insert > New Symbol (or press CTRL/CMD+F8) to create a new movie clip called particle. Now use the Color Mixer panel to set up a radial fill style that fades from opaque white at the center to transparent white at the perimeter:



3. Add a circle with no stroke to the particle movie clip. Use the Property inspector to give it a diameter of 80 pixels (W:80.0, H:80.0) and place it bang in the center of the stage (X:-40.0, Y:-40.0):



Now click on the Scene 1 link to head back to the main timeline.

4. Press CTRL/CMD+L to call up the Library panel, and drag the particle item onto the stage to create an instance of the movie clip. Select the instance and use the Property inspector to place it at X:290, Y:190. While you're at it, don't forget to give it an instance name: `particle_mc`.

4 Foundation ActionScript for Macromedia Flash MX



Remember, we always put `_mc` at the end of a movie clip instance name to remind us what sort of instance we're dealing with.

- Now select frame 1 of the actions layer, open up the Actions panel, and click on the pin button to pin it to the current frame. Now add the following script:

```
i = 0;
while (i < 10) {
    // actions to loop go here
    i++;
}
```

This is just the same loop as we used before, except that we're using a variable called `i` as our counter instead of `myCounter`.

So what about all that stuff I said in Chapter 2 about naming your variables sensibly? Okay, this is a bit of an exception: when we're writing actions inside loops, we often end up using the counter variable rather a lot, for array offsets and other such things. If you call it something like `myCounter`, these actions can get very long and complex looking. It's therefore quite common to use a variable called `i`, `j`, or `k` instead. I won't bore you with an explanation of *why* these particular names are used; suffice to say they hark back to the days of mainframe computers that had just 16K of memory! I promise it won't take you long to get used to this little quirk.

- Now we're going to add an action into our loop that will duplicate the particle movie clip for us:

```
i = 0;
while (i < 10) {
    // actions to loop go here
    duplicateMovieClip(particle_mc, "particle"+i+"_mc", i);
    i++;
}
```

This is something totally new – and very useful! It shouldn't take a genius to work out *what* an action called **duplicateMovieClip** does, but *how it works* is another matter...

This action takes a grand total of three arguments. The first is the instance name of the movie clip we want to duplicate – in this case it's `particle_mc` – which doesn't need any quotes around it. The second argument is where we specify what the duplicate should be called. Each instance

should have a unique name, so we build this up from two string literals and the counter variable. We're naming each duplicate instance as:

```
"particle"+i+"_mc"
```

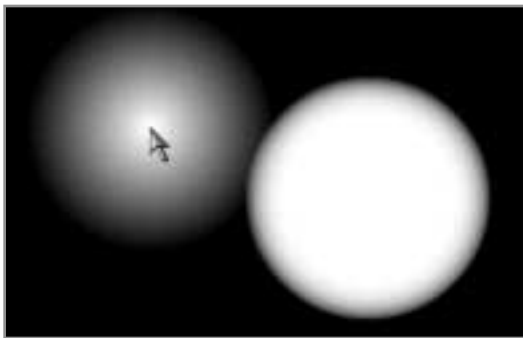
So when `i` has a value of 0, this works out as `particle0_mc`. Likewise, when `i` has a value of 1, it works out as `particle1_mc`. And so on. The third argument tells Flash which **depth level** to place the duplicate on. The only thing you really need to know about this (for now at least) is that you can only have one movie clip on each depth level; put two at the same depth and the second will replace the first. We therefore need to change this argument for each new duplicate, so we use the counter variable `i` again.

The end result is that we wind up with ten duplicates of the `particle_mc` movie clip instance, called `particle0_mc`, `particle1_mc`, `particle2_mc`, and so on. All this from one action inside a loop! Let's do something with them...

7. Now that we have a grand total of eleven movie clip instances on the stage, we can start dragging them around the place in a variety of weird and wonderful ways. Say we recycle the mouse following callback we brewed up at the end of Chapter 1:

```
onEnterFrame = function() {  
    // move particle to mouse position  
    particle_mc._x = _xmouse;  
    particle_mc._y = _ymouse;  
};
```

Just pop this on the end of your script, run the movie, wiggle your mouse around, and watch that particle go!



8. Ahem. Well, we now have *one* particle following the mouse around; what about the other ten? Of course, we don't want to write separate actions to control every single one. That would be rather silly now that we have the power of loops to fall back on.

4 Foundation ActionScript for Macromedia Flash MX

```

onEnterFrame = function() {
    j = 0;
    while (j < 10) {
        // move particle to mouse position
        particle_mc._x = _xmouse;
        particle_mc._y = _ymouse;
        j++;
    };
};

```

The only problem with this is that we're now updating the position of `particle_mc` ten times over, when we actually want to reposition a *different* instance each time.

9. Ideally, we want to build another string like `"particle"+j+"_mc"`, and use *that* to specify which instance to reposition. Unfortunately, we can't just put.

```
"particle"+j+"_mc"._x = _xmouse;
```

because Flash will think that you're trying to assign the value of `_xmouse` to something that isn't a variable or a property. It's not smart enough to spot that you're actually talking about the `_x` property of an instance called `particle0_mc` or whatever

We need to give it a helping hand, and this leads us straight on to our second crafty trick for the day: instead of writing...

```
"particle"+j+"_mc"._x = _xmouse;
```

we can put...

```
_root["particle"+j+"_mc"]._x = _xmouse;
```

The `_root` bit on the beginning tells Flash to look in the main (or 'root') timeline, and the stuff inside the square brackets `[]` tells it the name of the thing to look for. It's as if `_root` is an array whose elements hold things like movie clips, and use names instead of numbers. Once we've pulled out the element we want, we can use it as normal:

```

onEnterFrame = function() {
    j = 0;
    while (j < 10) {
        // move particle to mouse position
        _root["particle"+j+"_mc"]._x = _xmouse;
        _root["particle"+j+"_mc"]._y = _ymouse;
        j++;
    };
};

```

Now you'll find that all the duplicates follow your mouse pointer around, while the original sits still in the middle of the stage. Progress to be sure – but we're not done yet!

- To finish things off, let's stagger the duplicate particles so that they form a blobby line between the original and the current mouse position. The original particle is centered on X:290, Y:190, so we start by working out the x and y differences between this point and the position of the mouse:

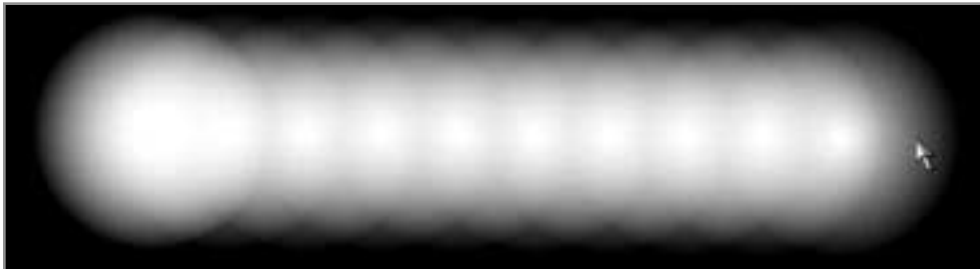
```
onEnterFrame = function() {
  // work out distances from center to mouse
  xDist = _xmouse - 290;
  yDist = _ymouse - 190;
```

Now, we're ready to move on to our loop, where we'll move each particle to X:290, Y:190 plus j tenths of the distance to the mouse:

```
  j = 0;
  while (j < 10) {
    // move particle to center plus j tenths of the distance
    _root["particle"+j+"_mc"]._x = 290 + (xDist * j/10);
    _root["particle"+j+"_mc"]._y = 190 + (yDist * j/10);
    j++;
  };
};
```

This means that `particle0_mc` will be in the same place as the original instance, while `particle1_mc` will be one tenth of the way to the mouse pointer. `particle2_mc` will then be two tenths of the way there, and so on, until we reach `particle9_mc` at nine tenths of the way to the pointer.

Test the movie again, and you'll see some fairly impressive results, even if we did need to do a little math to get them...



It's still not *quite* what we set out to achieve though! There's a big fat blob at one end (where `particle0_mc` sits on top of the original) and a conspicuous gap at the pointer itself, where we have no particle at all. This all boils down to that old 'counting from zero' business I mentioned before. This is one time where it would be much more useful if we could start counting from 1 (placing the first duplicate at one tenth of the distance) and finish at 10 (placing the last duplicate at ten tenths of the distance – smack bang on top of the pointer!)

4 Foundation ActionScript for Macromedia Flash MX

- 11.** In fact, it turns out that nothing could be simpler. Both of our loops currently use `i++` (or `j++`) to bump up the loop counter at the end of each cycle. So what about changing the counter at the *start* of each loop instead? You'll need to change both loops, so here's the full script:

```
// set up particles
i = 0;
while (i < 10) {
    i++;
    // duplicate the particle_mc instance
    duplicateMovieClip(particle_mc, "particle"+i+"_mc", i);
};
onEnterFrame = function() {
    // work out distances from center to mouse
    xDist = _xmouse - 290;
    yDist = _ymouse - 190;

    j = 0;
    while (j < 10) {
        j++;
        // move particle to center plus j tenths of the distance
        _root["particle"+j+"_mc"]._x = 290 + (xDist * j/10);
        _root["particle"+j+"_mc"]._y = 190 + (yDist * j/10);
    };
};
```

Now we start each loop with a counter value of 0. The first thing we do is bump it up to 1, use that value to do stuff with `particle1_mc`, and then loop back. The counter's less than ten, so we run the loop actions again: bump the counter up to 2, use it to do stuff with `particle2_mc`, and then loop back again.

We keep doing that until the counter's 9. That's still less than 10, so we run the loop actions, which bump it up to 10, use it to do stuff with `particle10_mc`, and loop back again. Only now does the `while()` action wise up to the fact that its `counter<10` condition *isn't actually true any more...*!

Run the movie one last time, and you'll see a beautiful, homogenous line of particles stretching out from the middle of the stage to wherever your mouse pointer happens to be. Hey presto!



Now that we've seen a little of what you can do with a `while` loop or two, let's take a quick look at the other sorts of loop available in ActionScript. "Other sorts of loop?" I hear you cry... "Surely there can't be any *more* ways to send scripts running round in circles?" Well, no.... and yes.

Most of the loops we've looked at so far use a pretty standard system: set up a counter variable, add one to its value every cycle, and test each time whether it's gone above a certain value. This approach is so ubiquitous that there's a special action devoted to it...

for loops

Whenever you build a loop that uses some kind of counter variable to count how many cycles you've gone through, it's worth considering the humble `for` loop as an option. It doesn't actually add anything new to the possibilities offered by `while()`, but it lets you set things out in a way that brings all the counter-specific actions together on one line, neatens things up, and should help you to keep things nice and clear.

Let's consider a loop that we've already looked at:

```
myCounter = 0;
while (myCounter < 10) {
    trace(myCounter);
    myCounter++;
};
```

There are four important parts to this loop, one on each of the first four lines:

- initialize the counter variable `myCounter`
- specify the loop condition `myCounter<10`
- perform the main loop action `trace(myCounter)`
- bump up the value of the counter with `myCounter++`

If we rewrite this loop as a `for` loop, we can cut this down to two lines:

```
for (myCounter = 0; myCounter < 10; myCounter++) {
    trace(myCounter);
};
```

We've now defined the whole range of the loop within the first line, so there's no need to set up a counter before the loop, and no need to change its value as one of the looped actions. `ActionScript` takes care of all that for us.

Okay, you're probably not convinced yet. These `for` loops certainly don't lend themselves to useful explanations like 'while the counter's less than 10, perform the following actions', and they *will* take a bit of getting used to.

4 Foundation ActionScript for Macromedia Flash MX

*The crucial thing to register is the fact that every factor that could possibly influence how many cycles this loop goes through is **specified in the `for()` action itself**. Since these factors are all in one place, you'll find it a lot easier to keep track of what the loop's doing when you come across it buried in a mountain of other stuff. Since you can see how many loops it needs to cycle through, it's less likely to give you a nasty surprise when you test your code. Also, you can easily modify a `for` loop so that it only cycles once – this can be tremendously useful while developing your code, since if you can see it working for a single loop, you can be confident that it will run for many cycles.*

Of course, we can still use the counter for all sorts of things other than simply counting how many times we've looped: as the index for an array or variable, or for deriving a sequence of numbers (such as a math times table; 5, 10, 15, 20, 25, and so on.)

Let's take a closer look at each of the three arguments we stick in the `for()` action.

init

The first argument is where you set up the counter variable. It's normal to set this to zero, though you're free to use any value you like. You can also name the counter as you wish. The total number of loops must always turn out to be a numeric variable, so valid start counter values are:

<code>counter = 0</code>	the counter starts at zero
<code>i = startValue</code>	the counter <code>i</code> starts at the value of numeric variable <code>startValue</code>

condition

This is just like the condition in an `if` statement or a `while` loop: as long as the expression evaluates to true, the loop will continue cycling. Inside a `for` loop though, the condition should always refer explicitly to the counter; normally in a form like 'counter is less than the stop value'. Valid examples are:

- `i < 10` loop as long as `i` is less than ten.
- `counter >= 200` loop as long as `counter` is greater than or equal to 200.
- `x < xMax` loop as long as `x` is less than `xMax`.

In the last case though, it's important that you *don't let your loop actions change the value of `xMax`* – well, not unless you really know what you're doing (in which case you'll probably be using another kind of loop anyway!) Flash won't stop you doing this, but it will make your code very unpredictable, and may even result in a forever loop.

One condition that beginners often get stuck with is:

- `counter == 10`

Although this is okay as a condition, it's probably not what you want for a `for` loop – unless `counter` equals 10 on the first loop, it will stop the loop immediately, and not do anything at all.

next

This is where you specify what should happen to the counter after every loop. Possible things you can add here are:

- `i++` add 1 to `i`.
- `counter--` subtract 1 from `counter`.
- `i = i+2` add 2 to `i`.

We've already encountered the `++` trick for adding one to the value of a variable; here's its partner in crime: `--` which takes one *off* the value. Anything else you want to do to the counter needs to be written out in full.

Some useful examples of for loops

If you want to see the following `for` loops in action, simply use the Actions panel to attach them to frame 1 of a new movie, and hit `CTRL/CMD+ENTER` to give the movie a test run. The Output window will then show you the `counter` value for each cycle (or iteration) of the loop.

Simple loop

This loop will count upwards from 0 to 12:

```
for (counter=0; counter <= 12; counter++) {
    trace (counter);
}
```

Reverse loop

This one will count down from 12 to 0:

```
for (counter=12; counter >=0 ; counter--) {
    trace (counter);
}
```

Two at a time

This will count upwards from 0 to 12 in steps of 2, and show the value of `counter` once the loop has terminated:

```
for (counter= 0; counter <= 12; counter = counter+2) {
    trace (counter);
}
trace (counter);
```

This demonstrates that the value of the counter just after the loop completes is *the first value that doesn't satisfy our condition*. In this case, that's 2 more than the maximum value of 12: namely, 14.

4 Foundation ActionScript for Macromedia Flash MX

Looping through elements in an array

As we've now established, loops are pretty darned useful for making a lot of actions run in a very short space of time. In particular, `for` loops give us a very tidy way to cycle through lots of values with just a couple of lines of code. One particularly useful thing we can do with loops is using them to work with arrays.

Say you've written a set of actions that all act on element `i` of an array. By looping through all available values of `i`, you can quickly apply them to every element in the array. So, a small piece of code can be made to work on large amounts of data.

We already saw a simple example of how you can use a `while` loop to look at all the elements in an array – we even persuaded it to stop when the array ran out of defined elements! In fact, the combination of loops and arrays is so important that we're going to return to it now, and look at a few more of the things they let us do.

Applying an operation to all the elements in an array

Consider an array of two hundred values between 1 and 20, which we've stored in an array called `rawdata`. I'd like to create a corresponding list of percentage values in a second array called `percent`. The non-looping way to do this (or should it be 'the *totally* loopy way?') would be to write out two hundred lines of script like this:

```
percent[0] = rawData[0]*5;
percent[1] = rawData[1]*5;
percent[2] = rawData[2]*5;
percent[3] = rawData[3]*5;
...
percent[198] = rawData[198]*5;
percent[199] = rawData[199]*5;
```

The sensible way to do it would be to use a loop – and since we know how many elements there are, it makes sense to use a `for` loop. First, we need to work out a generalized action for our 'convert to percent' process.

That's not too hard: `percent` just means 'out of one hundred', and 1 in 20 is just the same as 5 in 100. So we just need to multiply all our numbers by five. Assuming we're using a counter variable called `i`, we can just say...

```
percent[i] = rawData[i]*5;
```

Now we loop through all values of `i` from 0 to 199, applying it each time until you've converted the whole array:

```
for (i=0; i<200; i++) {
    percent[i] = rawData[i]*5;
}
```

What if we don't know how many elements there are in the array? We've already seen one way to deal with it using a `while` loop, which we *could* translate into `for`-style notation:

```

for (i=0; rawData[i]!=undefined; i++) {
    percent[i] = rawData[i]*5;
}

```

Hmm. Well Flash won't actually *stop* us doing this, but it still goes against everything we've just learnt about `for` loops. Sure, the loop condition still depends on `i`, but only very loosely; it depends a lot more on the contents of the `rawData` array, and it's not at all obvious how many times we expect to loop... This is one situation where things are best left to `while` loops!

On the other hand, if we had some way to ask the array directly how many elements it had, then we could still get away with a condition like `i<lengthOfArray`. Of course, there's just such a way: all arrays have a property called `length`, which tells you how many elements it contains. So it looks like our best bet is to use `i<rawData.length` like this:

```

for (i=0; i<rawData.length; i++) {
    percent[i] = rawData[i]*5;
}

```

The length property is common to all arrays, and tells us the number of elements they contain. This technique gives us a very simple way to put limits on all our array loops.

Searching an array for a specific value

Another thing we can do using a combination of loops and arrays is to search through the elements in an array until we find a particular value. Say we have an array of book titles called `myLibrary`, and use a variable called `searchString` to hold the name of a book we're looking for. There are two ways we could approach this.

We could say:

```

searchString = "The Wasp Factory";
i = 0
while (library[i] != searchString && i < library.length) {
    i++
}
if (library[i]==searchString) {
    trace("Match found: item number " + i);
} else {
    result = "No matches found.";
}

```

We don't know in advance how many loops we're going to need before we find a match, so we set up a `while` loop. We then keep adding one to the value of `i` until `library[i]` matches our search string or we run out of array elements. We then use an `if...else` to check whether the loop ended because a match had been found (in which case we trace out the relevant element number) or because it got to the end of the array without finding any matches at all (in which case we say just that).

4 Foundation ActionScript for Macromedia Flash MX

On the other hand, we might want to do it like this:

```
searchString = "The Wasp Factory";
matchesFound = 0;
for (i = 0; i < library.length; i++) {
    if (library[i] == searchString) {
        trace("Match found: item number " + i);
        matchesFound++
    }
}
trace("Number of matches found: " + matchesFound);
```

This time, we assume that we're going to search the whole library, on the basis that there may be more than one copy of the book we're looking for. We therefore use a `for` loop which cycles as many times as there are elements in the array (thanks, once again, to the `length` property). Every time around, the `if` statement will check whether there's a match with the search string. If there is, we trace out details of the match, and bump up the number of `matchesFound`. Once the whole array has been searched, we trace out the total `matchesFound` value.

Cross-indexing a pair of arrays

Once we've figured out how to search an array, we're only one step away from cross-indexing with another array. Let's consider two arrays, each containing a list of plant names: each element in the first contains the common name of a plant, while the corresponding element in the other holds the Latin name of the same plant. They might look like this:

```
commonNames = new Array("foxglove", "palm tree", "rose",
    ➤ "Californian poppy", "witch hazel")
latinNames = new Array("digitalis", "trachycarpus", "rosa",
    ➤ "Escholzia californica", "hamamelis mollis")
```

What if we know the common name of a particular plant, but want to find out what its Latin name is. We might ask the question, 'what is the Latin name for the Californian poppy?' We just need a loop that will cycle through elements in the `commonNames` array until it finds one that contains the string "Californian poppy". It could then use the successful index value (in this case 3) to call up the corresponding element from the `latinNames` array and find the answer: "Escholzia californica".

Here's how you might do it:

```
offset = 0;
search = "Californian poppy";
while (commonNames[offset] != search && offset < commonNames.length) {
    offset++;
}
if (commonNames[offset] == search) {
    trace("Match found: " + latinNames[offset]);
} else {
    result = "No matches found.";
}
```

In this case, we don't necessarily want to search on every element – we'll stop just as soon as we've found a match. Since we can't know in advance how many elements we'll need to search on, a `for` loop isn't suitable, so we use a `while` structure. As before, it loops around incrementing our `offset` variable for as long as (a) the element `commonNames[offset]` doesn't match our search term, and (b) `offset` is less than the array length.

So, when the `while` loop finishes, it could be for one of two reasons: either we've found a match or we've reached the end of the array and found none. This method – finding an index and applying it to another related array – is known as **cross-indexing**.

Now that you've seen a few theoretical examples of combining loops with arrays, it's time to put your newfound skills to work. We're going to use Flash and ActionScript to create a game of 'Hangman', using arrays and loops together with more than a little string variable manipulation.

As ever, a nice practical example should help lift the whole thing right off the page!

Hangman

It's an old game that's been around for centuries. Pick a word at random, write it out using blanks in place of each letter, and challenge a friend to guess what it is by picking letters from the alphabet. If they choose a letter that's in the word, you show them where it is in the word by filling in a blank. If they pick one that isn't, you add pieces to a drawing of a scaffold. After five or six wrong guesses, the scaffold is complete, and you start adding a figure hanging from the gibbet. After about ten wrong guesses, the figure is complete – HANGMAN! The only way for your friend to win is by guessing all the letters in the word before they're hanged.

Hangman is a brutal game when you think about it, and there's no hiding the fact that it's a game about someone getting hanged. All the fluffy bunnies in the world aren't going to soften the blow, but I've decided all the same to get my own version looking like it was all drawn and painted by a child.

Let's start with a quick walkthrough, to give us an idea of how the gaming experience should pan out for our players:

1. When we start the movie, we'll see the following scene:



4 Foundation ActionScript for Macromedia Flash MX

An empty field on a sunny day, with a title, a solitary flower and play game button, which we can press to start the game.

2. Once we press the button and the game starts, we're presented with a simple interface: a box for typing in our guesses, an enter button to submit them, and a list of question marks indicating how many letters there are in the word we have to guess:

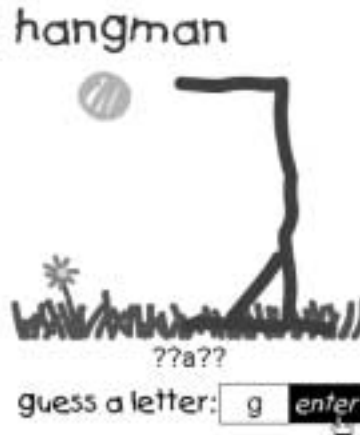


3. The next step is to type in a letter and hit the enter button. Let's say our mystery word is 'flash' and I've just tried the letter 'a'. This is what I should see:



There's one 'a' in the word 'flash', so Flash replaces the relevant question mark with the letter itself.

4. Say we now try a few more letters, such as 'u', 'x', 'q', and 'z'. Okay, let's just say we're being deliberately dumb here... None of these letters occur in the word 'flash', so they're all bad guesses – so we have to pay the price:



Four wrong letters, so four sections of the gallows appear... If we don't start making some decent guesses soon, we'll be hung out to dry! As you can see from the screenshot, I'm just about to guess the letter 'g'. I haven't pressed enter yet, so all is not lost...

5. Let's assume I now replace the 'g' with an 'f', hit enter, and suddenly realize what the word is. Follow it up with 'l', 's', and 'h', and the word is complete. This is what I should now see:



The gallows vanish, and our would-be victim is now free to wander through the sunlit fields with a big red flower in his hand. How sweet.

6. However, there's an alternate reality in which I didn't recant my faith in the letter 'g'. I left a 'g' in the input box and hit enter! With no clue to what the word could be, I floundered around with such desperate guesses as 'm', 'n', 'o', 'p', and 'q'... Yes, I was so desperate (and forgetful) that I tried 'q' again! By that time though, there was no hope:

4 Foundation ActionScript for Macromedia Flash MX



Our hero meets a grisly end, and the only solace to be found is in the little button that reappears behind the gallows, hinting that we can actually have another go.

Well, that's the game – and I did warn you things might get a little unpleasant! Now that we've got a good idea of how the game 'hangs' together, let's start looking at how we can build it in Flash.

If you're in a rush and don't want to create all these graphics from scratch, you can find them all in this chapter's download, in the library of a file called `hangman_start.fla`. In case you're wondering, the font I've used throughout is called "kids".

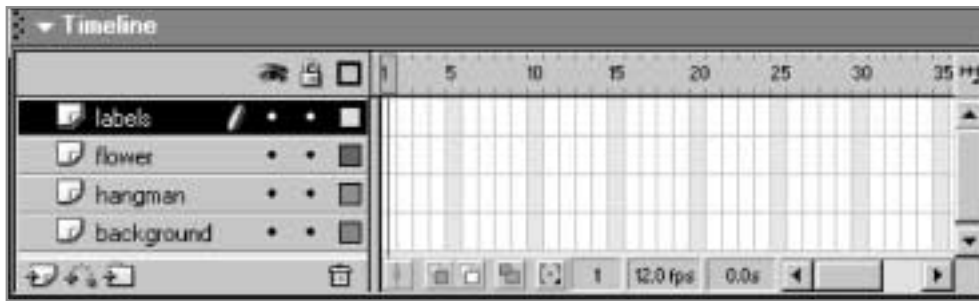
We'll start by breaking it down into symbols that we need in the library. I've used four graphics (called flower, grass, sun, and title – no prizes for guessing what they all are...), plus two buttons (enter and play) and put all the 'hanging' material and interface stuff into a couple of movie clips (called hanging man and interface).

There's nothing terribly special about the buttons (one's a black box behind a bit of static text that says 'enter', the other's just a bit of text that says 'play game' – select and press F8 to convert them to symbols and you're done), and the graphics should be self-explanatory (the sun's an orangey circle, the title says 'hangman', and so on – select and hit F8 again... I'm sure you get the idea) so I'll skip straight on to how we build the two movie clips!

The 'hanging man' movie clip

This is where we pull together our scenery, and animate the awful process of a hanging...

1. First, we need to create a total of four layers, which you should call labels, flower, hangman, and background:



2. Select the background layer, and pull some sun and grass onto it from the library:



Select frame 19 and press F5 to insert a frame. We want this scenery on display throughout the clip.

3. Now select frame 1 on the labels layer, and press F6 to insert a keyframe. Press F5 to insert a frame at 19, and F6 to add keyframes at 10 and 15. Use the Property inspector to label your three keyframes as play, lose, and win respectively. Your timeline should now look like this:



4. Now for the creative part... we're going to add the frame-by-frame animation of the gallows and hanging man. Select frame 1 of the hangman layer and use F6 to insert a keyframe. Select frame 19, and press F5 to insert a normal frame. It should now look like an albino version of the background timeline.

Select frame 2, hit F6 to add a keyframe, and draw a horizontal brown line along the top of the grass. Select frame 3, hit F6 to add a keyframe, and add a vertical brown line at the right-hand end.

4 Foundation ActionScript for Macromedia Flash MX

Select frame 4, hit F6 to add a keyframe, and add a diagonal brown line connecting the two lines you just drew.

Now keep going, building up the gallows a piece at a time, and then adding our hapless victim (one piece at a time – no, I've never understood that bit either) until you reach frame 10, which is where folks go when they've lost the game. Each keyframe represents one step closer to doom! By this point, you should have something like this:



I've also added in some text here, just to emphasize the fact that our tale is over...

- Of course, if our player has won the game, we need to reward their efforts and give them a nice uplifting image. Select frame 15, and hit F6 to add a keyframe. You can delete the gallows from this frame, and replace them with something happier:



- Now for a nice little finishing touch. Select frame 1 of the flower layer, drag a copy of the flower symbol onto the stage, and nestle it down somewhere safe in the grass:

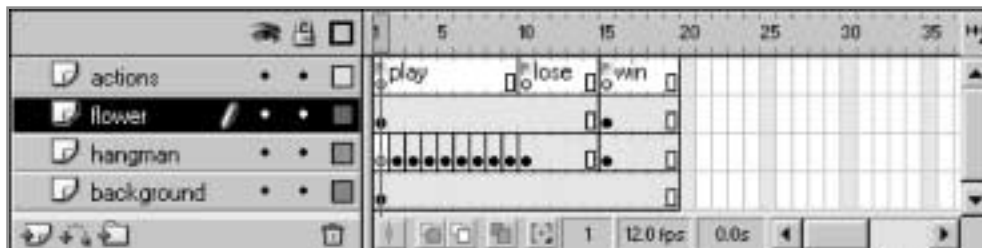


Now add a keyframe at frame 15, and drag the flower across so that it looks as if it's being held in our hero's hand:



Select frame 19 and press F5 to insert a frame.

Phew! After all that hard work, your 'hanging man' timeline should be looking like this:



Aren't you glad I didn't show you that before we started? Now that we've dealt with the aesthetics of our game, let's take a look at the user interface.

The 'interface' movie clip

You'll be glad to hear that this movie clip's a great deal simpler than the last one! There's one layer, one keyframe, and no fancy graphics to labor over. In fact, the following image sums it up quite well:



4 Foundation ActionScript for Macromedia Flash MX

There are just four things on the stage:

- A dynamic text field called `display_txt` at the top (for which `Var` is set to `display`)
- A static text field on the left, reading 'guess a letter'
- An input text field called `input_txt` (for which `Var` is set to `input`)
- An instance of the enter button, which I've called `enter_btn`

We only want the player to enter lower case characters, so select the input text field and hit the Character button on the Property inspector. In the Character Options dialog, check the boxes for Only and Lowercase Letters (a-z), and press Done to confirm.



Once you've created each of these and set them up on the stage as described, you're all done. Now we're ready to take a look at the main timeline, arrange our movie clips, and wire them all up with a little ActionScript...

The main movie timeline

This timeline needs two layers, movie and actions, with just one keyframe on each:

- movie contains...
 - an instance of the title graphic, which reads 'hangman'
 - an instance of the hanging man movie clip called `hangman_mc`
 - an instance of the interface movie clip called `interface_mc`
 - an instance of the play button called `playGame_btn`
- actions will hold all our ActionScript

Here's how the stage should look once everything's in place:



Well, we've got all the pictures in place – how about a little code? Now the code for this game might look a little hairy at times, but don't panic! Nothing we do here will be hideously complicated; if it all starts to look like a mess of brackets and semicolons swimming about in front of your eyes (yes, that sometimes happens to me too) just take it one line at a time. Look for the structures you understand and think how they're being used to form the game we described earlier on.

Adding in the ActionScript

Select frame 1 on the actions layer, open up the Actions panel, click down that pin button, and get ready to wire everything together. Although there are lots of actions for us to put in over the next couple of pages, they all need to be attached to the same frame. What's more, they break down into three neat pieces:

- Movie setup – actions to run when the movie starts
- Game setup – actions to run when we click play game to start the game
- Game play – actions to run when we click on enter to submit a guessed letter

Let's get stuck in!

1. The first chunk of code we need to add will set our movie up when it starts running. We begin by creating three arrays:

```
// initialize array of words
wordList = new Array("computer", "apple", "coffee", "flash",
    ➤ "animal", "america", "panel", "window",
    ➤ "import", "number", "string", "cake",
    ➤ "object", "movie", "aardvark", "kettle");
// initialize arrays for letters
```

4 Foundation ActionScript for Macromedia Flash MX

```
lettersNeeded = new Array(10);  
lettersGuessed = new Array(10);
```

The first of these lists all the words we're going to choose from. If you want to, you can add more by simply sticking them on the end of the list. Words must be in lower case letters (since that's all we'll allow the user to enter) and can have up to ten letters, but no more.

This is simply because the two other arrays we define here are only big enough to hold ten characters. Remember that zero is counted as the first element, so asking for ten elements will give you elements numbered from 0 to 9.

We're going to be using the elements of `lettersNeeded` to store letters from our mystery word, while `lettersGuessed` array elements will hold a mixture of question marks and letters that the player has guessed correctly – these are the letters we'll show them while they're playing the game.

2. We then have to stop the main timeline, stop the `hangman_mc` timeline, and hide the game interface:

```
stop();  
hangman_mc.stop();  
interface_mc._visible = false;
```

Now that we've set up the movie with arrays and stopped timelines, it's time we laid down some actions that will get us ready for a game.

3. When the player presses the play game button, there are several commands we need to give Flash before we can let them start guessing letters. We can put these into an event handler for the `onRelease` event of the `playGame_btn` instance. It'll start out like this:

```
playGame_btn.onRelease = function() {  
    hangman_mc.gotoAndStop("play");  
    playGame_btn._visible = false;  
    interface_mc._visible = true;
```

First, we initialize the hangman movie clip, sending its play head to the first frame. In fact, at this point in the game, it's probably there already – but we'll want to run these actions again when we set things up for a rematch, so this is an important thing to do.

We then make the play game button invisible (since we don't need it now) and make the game interface visible; we're certainly going to need that!

4. Next up in the event handler, we select a word at random from the `wordList` array:

```
randomNumber = Math.round(Math.random()*(wordList.length-1));  
selectedWord = wordList[randomNumber];
```

At first glance, you may think the first line looks pretty terrifying. All it actually does is pick a random whole number between 0 and `wordList.length-1`, and store it in a variable called `randomNumber`.

We then use that number to specify one of the elements in the `wordList` array, and pull out the word it contains, and store this randomly selected word in `selectedWord`.

5. The next step is to set up a variable that's going to help us keep track of how many more letters the player still has to guess. At this stage, they've guessed none, so this is just the number of letters in the word:

```
lettersLeftToGo = selectedWord.length;
```

6. Now it's time to break our word down into a list of letters. As I mentioned before, the `lettersNeeded` array is going to store all the letters our player has to guess. We set this up using a `for` loop, and cycle through each letter in the word, putting it into the corresponding element in the array:

```
for (i=0; i<selectedWord.length; i++) {
    lettersNeeded[i] = selectedWord.charAt(i);
}
```

The `charAt(i)` bit at the end of the second line is new, but all it actually means is 'get the character at position `i`' of the `selectedWord` variable it's stuck on the end of.

7. We do a similar thing for the `lettersGuessed` array, but fill every slot with a question mark – only fair, since the player hasn't guessed any letters yet!

```
for (i=0; i<selectedWord.length; i++) {
    lettersGuessed[i] = "?";
}
// initialize display of lettersGuessed by player
interface_mc.display = lettersGuessed.join("");
};
```

The last line here, just before we finish off the event handler, uses another new trick to join together all the element values in `lettersGuessed`. We send the result (which will be a long string of question marks) to `interface_mc.display`; that is, the variable called `display` from the `interface_mc` movie clip, whose value gets shown in the dynamic text field.

Phew! Let's stop for a moment and take a quick breather. That's a lot of script, and a fair few new tricks along the way. Try running the movie now, and press the play game button. Sure enough, the button vanishes, the interface appears, and a string of question marks appears just below the grass:



4 Foundation ActionScript for Macromedia Flash MX

Now select Debug > Show Variables, and you can check out the contents of all our arrays:



```

Output
-----
12:"object",
13:"movie",
14:"sardvark",
15:"kettle"
}
Variable _level0.lettersNeeded = [object #2, class 'Array'] [
  0:"a",
  1:"g",
  2:"g",
  3:"l",
  4:"e"
]
Variable _level0.lettersGuessed = [object #3, class 'Array'] [
  0:"?",
  1:"?",
  2:"?",
  3:"?",
  4:"?"
]
Variable _level0.selectedWord = "apple"
Variable _level0.lettersLeftToDo = 5
Variable _level0.l = 5
Movie Clip: Target="_level0.hangman_mc"
Movie Clip: Target="_level0.interface_mc"

```

As well as helping to reassure you that things are going to plan, this should also serve as a reminder that the information you store inside a Flash movie isn't that hard to pull out again. Sure, once you've compiled a SWF and set it loose in the outside world, it won't be quite this easy to peer inside for a quick peek at what's going on; but it's certainly not impossible. Anyone with a bit of tech-savvy and enough perseverance can figure out exactly what words you've got stored in wordList and cheat you out of a victory... You have been warned!

That aside, we've got a game to finish off, and an enter button to wire up. Let's get to it!

8. The enter button (instance name `enter_btn`) is actually *inside* the movie clip instance called `interface_mc`, so we need to set up our callback like this:

```
interface_mc.enter_btn.onRelease = function() {
```

Suffice to say, `interface_mc.enter_btn` just means 'the instance called `enter_btn` that's *inside* the instance called `interface_mc`'. We'll learn a lot more about these dot-based shenanigans in the next few chapters; so don't let this oddity put you off just yet.

9. The first thing Flash has to do once the player hits enter is to assume that they've guessed wrong:

```
wrong = true;
```

Now we just loop through all the letters in the word (as stored in the `lettersNeeded` array) and change this value as soon as we find a match with the input text field in `interface_mc`:

```
for (i=0; i<selectedWord.length; i++) {
    foundLetter = lettersNeeded[i] == interface_mc.input;
    if (foundLetter) {
        // guess matches a letter in the word
        wrong = false;
        lettersLeftToGo--;
        lettersGuessed[i] = interface_mc.input;
    }
    interface_mc.display = lettersGuessed.join("");
}
```

The variable `foundLetter` gets its value from the expression `lettersNeeded[i] == interface_mc.input`, which will always be Boolean. It will only be true if the letter our player just entered is the same as the letter in element `i` of the `lettersNeeded` array.

We also bump down the number of letters left to guess, change the appropriate question mark in the `lettersGuessed` array so that it shows the correct letter, and use `join()` again to update the display.

Hang on a minute though! What if we picked the same letter more than once? If it matched the first time, it would match again, and again; `lettersLeftToGo` would hit zero before very long, suggesting that we'd won the game. It looks like we need to be a little more picky with the condition in our `if` statement.

10. Make the following highlighted additions to the `for` loop we just wrote:

```
for (i=0; i<selectedWord.length; i++) {
    foundLetter = lettersNeeded[i] == interface_mc.input;
    notGuessed = lettersGuessed[i] != interface_mc.input;
    if (foundLetter && notGuessed) {
        // guess matches a letter we haven't already found
        wrong = false;
        lettersLeftToGo--;
        lettersGuessed[i] = interface_mc.input;
    }
    interface_mc.display = lettersGuessed.join("");
}
```

Now we check that our player's input is *not* the same as element number `i` in the array `lettersGuessed`. Now, if `foundLetter` and `notGuessed` are *both* true, we know we've got a brand new correct guess!

4 Foundation ActionScript for Macromedia Flash MX

- 11.** Next, we empty out the input box, so that it's all ready for our player to have another go:

```
interface_mc.input = ""; // reset input text box
```

- 12.** If the player's guess was no good (or they were foolish enough to guess at a letter they'd already got) the variable `wrong` will still have a Boolean value of `true`, in which case the next few lines kick in:

```
if (wrong) {
    hangman_mc.nextFrame();
    if (hangman_mc._currentFrame == 10) {
        // GAME OVER!!
        interface_mc._visible = false;
        playGame_btn._visible = true;
    }
}
```

First, we use `nextFrame()` to bump the `hangman_mc` play head onto the next frame. We then check whether that happens to be frame 10, also known as HANGMAN!

If it is, we run the 'Game Lost' actions: these just hide the interface and show the play game button (so the player can play the game again if they feel the need). If it's not frame 10, we don't need to show anything here.

- 13.** Last bit now! We just need to test whether all the letters have been guessed correctly. If they have, the value of `lettersLeftToGo` should be down to 0. We use this fact to control whether or not the 'Game Won' actions should be run:

```
if (lettersLeftToGo == 0) {
    // GAME WON!!
    hangman_mc.gotoAndStop("win");
    interface_mc._visible = false;
    playGame_btn._visible = true;
}
};
```

These simply send `hangman_mc` to the win frame (where we get to see our hero holding a flower and smiling in gratitude), hide the interface, and show the play game button.

Now we're all done, you can sit back and relax – and maybe have a quick game! Of course, the tension won't be quite so high now that you've seen the full list of words our movie gets to choose from. All the same though, it's nice to see that all this script will actually *do* something.

Summary

In the course of this chapter, we've looked at three different ways to make something loop in Flash:

- Timeline loops using the `gotoAndPlay()` action
- ActionScript loops using the `while()` action
- ActionScript loops using the `for()` action

That's not to say there are no other looping actions: there's **do...while**, which is almost exactly the same as `while`, except that it tests its condition *after* running the loop actions each time; there's also **for...in**, which we can use to loop through all the elements in an array, or all the instances in a timeline. When all's said and done though, they don't really offer much beyond what we've already achieved. As we've seen, even `for` loops are really just a specialized version of your basic `while` loop; just a particularly useful one! I haven't set out to try and be comprehensive here, since the three we've covered should do for most of your needs. The only question is when you should use which one.

Here are some simple rules of thumb for deciding when (and when not) to use an ActionScript loop or a timeline loop:

- If you want Flash to perform an action (or set of actions) many times over in the shortest possible period of time, you should use an ActionScript loop. Every cycle of the loop is attached to one frame, so (unless you run over 200,000 cycles, when Flash starts getting a little nervous) they'll always complete before you move on to the next frame.
- If you want to make something move gradually across the stage, you're best off using a timeline loop to nudge it by a few pixels for each frame. An ActionScript loop will work its magic much, much faster, so all you're likely to see would be the start and end points.

If you've settled on an ActionScript loop, picking between `while` and `for` loops is fairly simple:

- If the values you use to define the looping behavior are all known in advance, then it's best to use a `for` loop. Otherwise, a `while` loop will almost always do the trick.

This chapter has had the biggest variety of examples so far, because the full range of your ActionScript skills you've been learning is really starting to come together. You've learned some important new techniques, and even more important, you've seen large sections of ActionScript working together in a finished application.

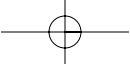
Don't worry if you've only skimmed through this. The important thing is that a standard has been set in your mind, and you've probably picked up a lot of useful little snippets without even realizing it. That's what coding is all about. You're probably looking at all the long pieces of code and saying 'Why has he done that? Why didn't he do it this way?' The proof is when you start doing your own programs and recognize things you're trying to do in terms of the programs I've listed here.

Each of the last three chapters has been an important step forward; building up the range of things you can do once you start throwing information around behind the scenes of your movies. Now that you've

4 Foundation ActionScript for Macromedia Flash MX

seen how to work with variables, arrays, decision-making, and looping, you should be getting a very good idea of just how much a few lines of ActionScript can add to your Flash movies.

You're not even halfway through the book yet, and you're already moving away from the level of 'novice' ActionScripter. In the next few chapters, we're going to look at some more ways of bundling together actions and information, and start thinking about how our scripts are structured. You may already feel confident enough to wire the odd snippet of ActionScript into your own projects; if so, that's great! Soon enough though, you'll be all set to start taking on some full-sized challenges.



More power, less script 4

