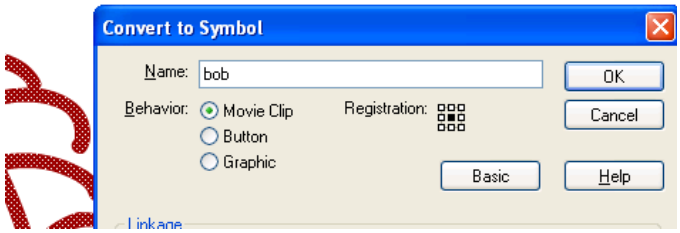
Flash Actionscript

Making A Symbol

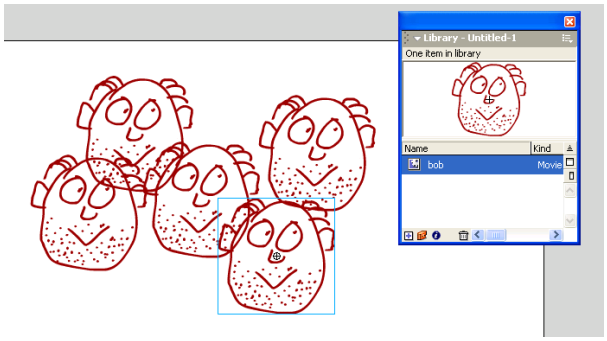
If you want to control the behaviour/appearance of an object in Flash, you first need to convert it into a symbol. To make a symbol you have two options

- draw the object normally, then highlight it with the black arrow tool and press **F8**. Make sure that you choose Movie Clip and give the clip a name.
- Press **CTRL + F8** and then draw your symbol, making sure that you choose Movie Clip and give your clip a name

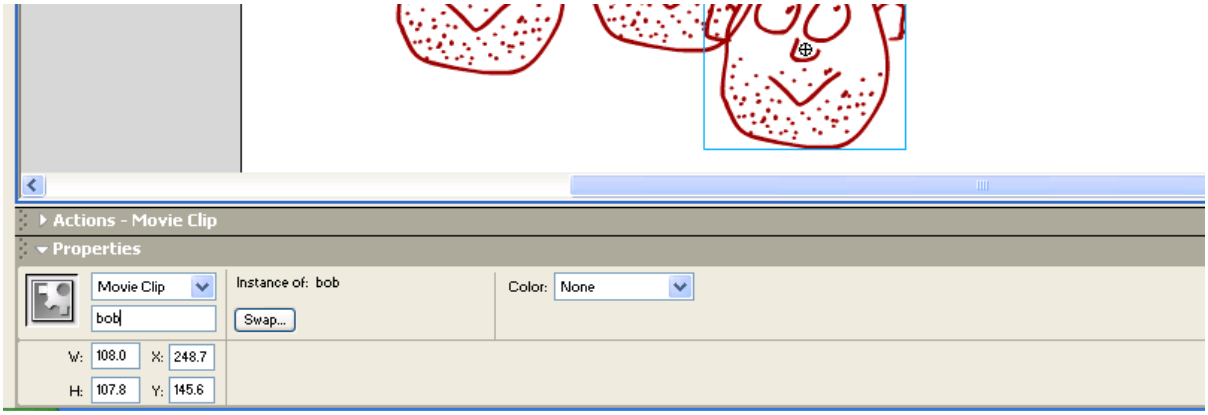


Instances

Once you have made a symbol it will be available in the Library (**F11**). You can drag as many copies as you like onto the stage.



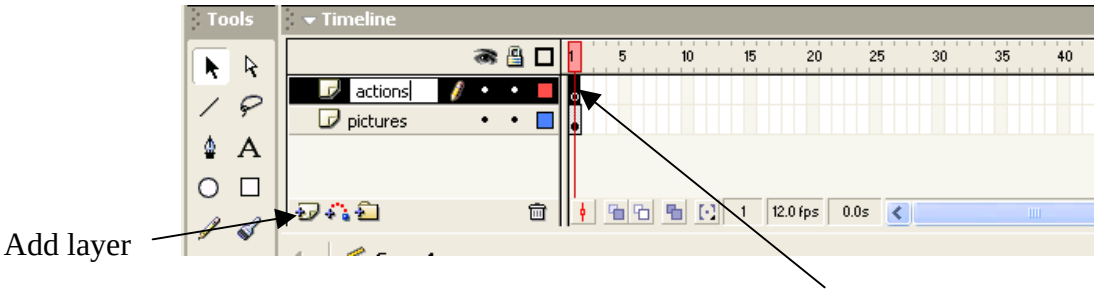
Each copy of the movie clip is called an **instance**. If I want to give instructions to each individual instance, they will each need their own name. To name an instance, you click on it, (so that a blue box appears around it) and then give it a unique name in the **properties** panel.



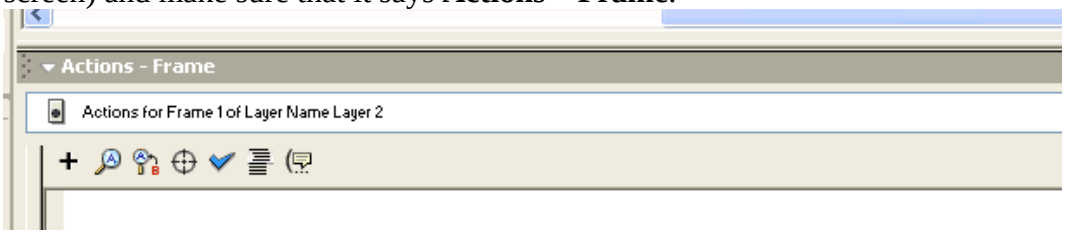
The Actions Layer

Once the instance has a name, you can give it instructions in the actions layer. It is advisable to keep all of your actionscript on its own layer and to ensure that you only add keyframes in this layer.

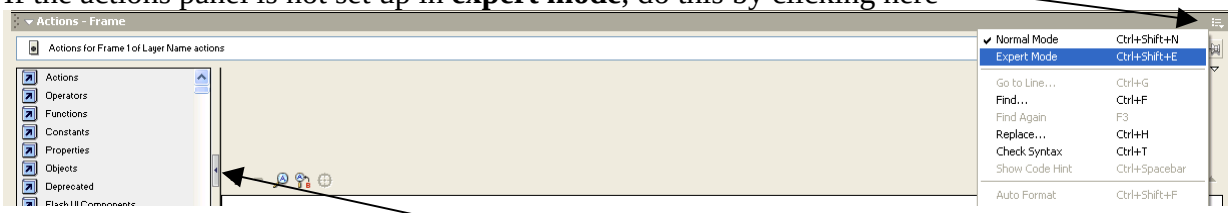
In the timeline, click on the add layer button and name your new layer **actions**



Before you type any actions, make sure that you have clicked here, so that frame 1 in the actions layer is black. Now click into the **Actions** panel (at the bottom of the screen) and make sure that it says **Actions – Frame**.



If the actions panel is not set up in **expert mode**, do this by clicking here



and then close the list on the left by clicking here.

You are now ready to give instructions to your instance.

Basic Actionscript Instructions

A typical actionscipt instruction looks like

```
bob._x=50;
```

The first part of this instruction identifies the instance that I want to give an instruction to. In this case, my instance is called **bob**. Make sure that the name you give here is identical to the name that you gave to the instance in the properties panel.

After the name you add a dot, to indicate that you have finished identifying the instance.

Now you add the property of **bob** that you want to set, (in this case the x position). There are various set properties in Flash, and these are always preceded by an underscore

```
bob._x=50;       //bob's x position
bob._y=100;      //bob's y position
bob._xscale=150; //stretches bob in the x direction, to 150% of original size
bob._yscale=200; //stretches bob in the y direction, to 200% of original size
bob._width=30;   //sets bob's width to be 30 pixels
bob._height=20;  //sets bob's height to be 20 pixels
bob._rotation=50; //rotates bob by 50 degrees, in the clockwise direction
bob._alpha=30;   //sets bob's alpha (transparency) to 30% (100% means solid)
bob._visible=false;  //sets bob to be invisible
```

The final part of each of the instructions above is a semi-colon. In Flash (and many other programming languages) each line of code is finished with a semi-colon. One of the most common problems with code is forgetting to add a semi-colon.

The program will stop working if you do not have the semi-colon!!!!!

If you experiment with the instructions above, you should be able to scale/move/twist bob as much as you like. What we cannot do yet is make bob change before our eyes. To do that, we need to start using the timeline.

The 3 frame program

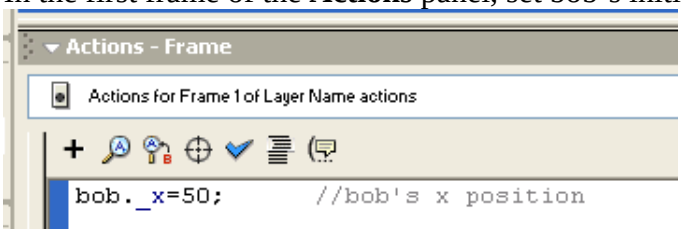
A very common procedure in Flash is to have code spread over 3 frames. The role of these frames is

- Frame 1 – set up the initial conditions (where does bob start).
- Frame 2 – change **bob** in some way
- Frame 3 – go back to frame 2, so that bob keeps on changing.

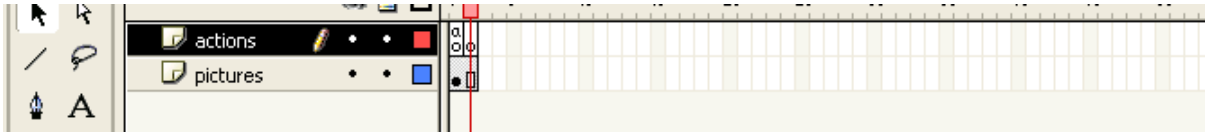
Make sure that you have an instance on the screen, in the pictures layer, **not** the actions layer. Also make sure that it has been named in the **properties** panel. If you want to copy my code exactly, your instance will need to be called **bob**.

Frame 1

In the first frame of the **Actions** panel, set bob's initial x position.



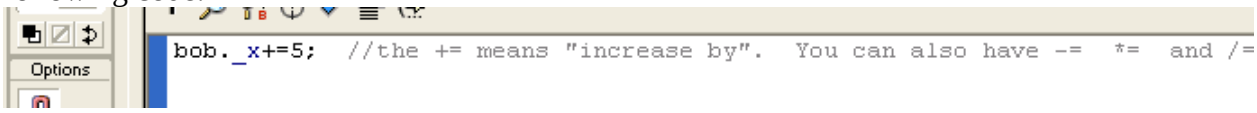
You now need to add a new keyframe to the actions layer. You can do this whilst in the actions panel, by simply pressing **F6**. You should see that a new frame is added in the timeline.



The cursor should already be waiting for you in the **Actions** panel.

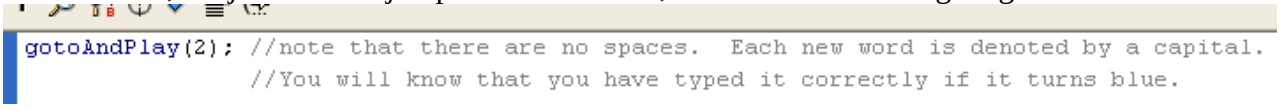
Frame 2

In this frame we want bob to move to the right, but we do not want to just set its x position to 55. Instead we want to increase its x position by 5. We do this with the following code.



Frame 3

In this frame, we just want to jump back to frame 2, so that bob moves right again.



Now when you play the movie, bob should move across the screen, from left to right, until he disappears off the right side.

Adding some control (keeping bob on the screen)

At the moment, bob just keeps moving to the right, even after he is no longer on the screen. To stop him we need to alter the code in frame 3. When bob reaches the right hand side of the screen, we no longer want to keep looping back to frame 2. Instead we want to stop. We do this using an **if** statement. The whole code looks like

```
if(bob._x<500)      //this sets the condition that we are looking for
{
    gotoAndPlay(2);  //this says what to do if bob's x pos is less than 500
}
else
{
    stop();         //this says what to do when he reaches 500
}
```

The first line of the code is the **if** statement `if(bob._x<500)` . The condition that you are looking for is given inside normal brackets. This checks to see if bob's x position is less than 500.

The next part of the code says what should happen if the condition is met. This part of the code must go inside curly brackets. In general, if you want to group a set of instructions, you place them inside curly brackets.

After the `gotoAndPlay(2);` instruction we need to close the curly brackets.

We then need to say what to do if the condition is not met. We do this with `else`.

Then, in another set of curly brackets, we say what to do.

`stop();` The stop command must always have an empty set of normal brackets after it.

Variables

Suppose you want to be able to adjust the speed at which bob moves across the screen. You could go into frame 2 and change the code to

```
bob._x+=10;
```

An alternative approach would be to change the code to

```
bob._x+=xJump;
```

where `xJump` is a **variable,** saying how much bob should move to the right each time. A variable can either be a number, a string (i.e. some text) or a true/false. You are able to change the value of the variable, whilst your Flash movie is playing, and so change what happens. In our case, we could change the value of `xJump`, so that bob's movement changes.

To show you how variables can be used in practice, we will make bob bounce around the screen.

We will use the same 3 frame movie, and just change the code slightly.

Frame 1

In frame 1 we will add two lines of code to define the variables `xJump` and `yJump`.

```
bob._x=50;      //bob's x position
bob._y=50;      //bob's y position

xJump=5;        //how much bob moves in the x direction each time
yJump=5;        //how much bob moves in the y direction each time
```

Frame 2

In frame 2 we will make bob's x and y positions change.

```
bob._x+=xJump;
bob._y+=yJump;
```

If you play the movie now you should see bob move across and down the screen. If you change the values in frame 1, you can change the speed at which bob moves.

Frame 3

It is in Frame 3 that we can make bob bounce. We want him to change direction when he gets to the edge of the screen.

To make him bounce off the right hand side of the screen, we want to check to see if his x position is greater than 500. If it is, we can make him change direction by making his `xJump` negative. We can do this by multiplying the current value of `xJump` by –1. The code is shown on the next page.

```
if(bob._x>500)       //we now want to act when x is greater than 500
{
    xJump*=-1;       //this code just multiplies the existing value by -1
}

gotoAndPlay(2);      //we now want to frame 2 all the time
                     //this movie will not stop
```

If you play the movie now, you should see bob move across the screen, bounce off the right hand side and travel left off the screen.

To make him bounce off the left hand wall we simply need to add code to make `xJump` change sign when bob's x position is less than 0.

```
if(bob._x>500)
{
    xJump*=-1;
}

if(bob._x<0)
{
    xJump*=-1;
}

gotoAndPlay(2);
```

If you play the movie now, bob should bounce from side to side as he moves down the screen.

Since both of the **if** statements above invoke the same action (changing the sign of `xJump`) they can be combined into a single **if** statement, that checks to see if either the x position is greater than 500 **or** less than 0.

```
if((bob._x>500)or(bob._x<0))
{
    xJump*=-1;
}
```

Note that each of the conditions is written in a pair of brackets and that they are both contained within an extra pair of brackets

Since we want to make bob bounce off both the sides and the top and bottom, we can add the same code to change his y direction. The final code in frame 3 becomes:

```
if((bob._x>500)or(bob._x<0))
{
    xJump*=-1;
}
if((bob._y>400)or(bob._y<0))
{
    yJump*=-1;
}
gotoAndPlay(2);
```

If you run the movie, bob should now bounce off all of the walls.

Following the Mouse

You can control a movie clip by making it follow the mouse. In actionscript you can find the x and y positions of the mouse, using `_xmouse` and `_ymouse`.

Let's change our movie so that bob follows the mouse around the screen. To make him take position himself over the mouse we use the following code.

```
bob._x=_xmouse;
bob._y=_ymouse;
```

We should put this code in both Frame 1 and Frame 2. (Although it might seem pointless putting the code in 2 frames, it is a good idea to keep the 3 frame structure. We will actually be altering the code in frame 2 in a minute).

In frame 3 we just want to `gotoAndPlay(2);`

When you play the movie, bob should stick to the mouse like glue. This is okay, but it would be a far more interesting effect to make bob chase after the mouse. To achieve this we want bob to move towards the mouse, without actually catching it. We will do this in 2 steps

- Find the distance between bob and the mouse.
- Move bob on half that distance.

We will do this in Frame 2. We will start by creating the variables `xDistance` and `yDistance`, which will be the distances from bob to the mouse. These will look like.

```
xDistance=_xmouse-bob._x;
yDistance=_ymouse-bob._y;
```

Now that we know how far bob would need to move to catch the mouse, we can make him move **half** the distance. To do this we will replace the `xJump` in our previous movie with `xDistance*0.5`. The code we need is

```
xDistance=_xmouse-bob._x;
yDistance=_ymouse-bob._y;

bob._x+=xDistance*0.5;
bob. _y+=yDistance*0.5;
```

Now if you run the movie you should find that bob chases the mouse with a bit of a lag.

Improving the script

The script above works perfectly well, but what if we want bob to take longer to catch the mouse? We would need to change the 0.5 in the last two lines of code.

This is not very efficient code. Every time we want to change the speed, we need to change two values, even though they are actually the same. This may not seem terribly inconvenient in this case, (especially since the lines are next to each other), but what if the same value needed to appear throughout the movie. You would need to find each place where the value was used and change it. Missing one of them could ruin the whole program.

For example, what if you decided to change the xscale of some pictures in the movie to 70. At a later point you might realise that they need an xscale of 60 to fit on the screen. If you miss one of pictures, the whole scale of the movie will be ruined.

The solution to this problem - and to our problem – is to use a variable. Let's call the variable "speed". We can write the following code in Frame 2:

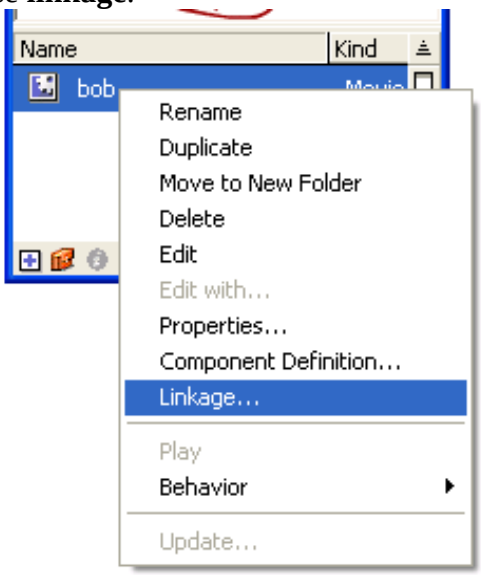
```
xDistance=_xmouse-bob._x;
yDistance=_ymouse-bob._y;

speed=0.5;    //define the value of speed

bob._x+=xDistance*speed;  //multiply xDistance by speed, instead of 0.5
bob._y+=yDistance*speed;
```

Now if you want to change the speed at which bob chases the mouse, you only have to do it in one place.

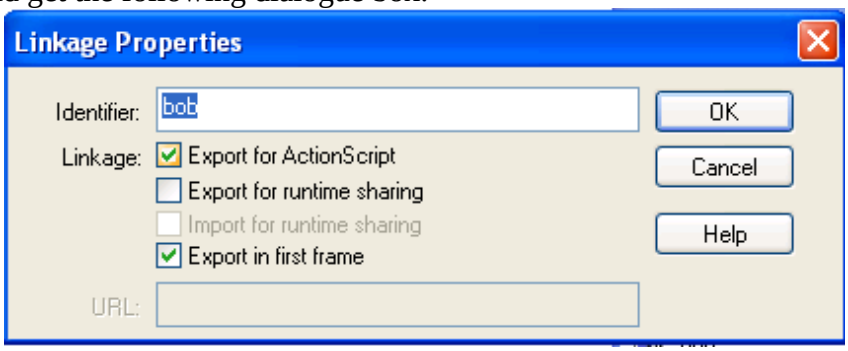
Attaching Movies with Script

If you want to have multiple copies of a symbol on the screen, or you want to add an instance of a movie clip to the screen while your Flash movie is playing, you can do this using the actionscript `attachMovie` command.

Before you can use `attachMovie`, you need to set the **linkage** of the movie clip. You do this in the **library** (**F11).** In the library, right-click on the symbol that you want to attach and choose **linkage**.



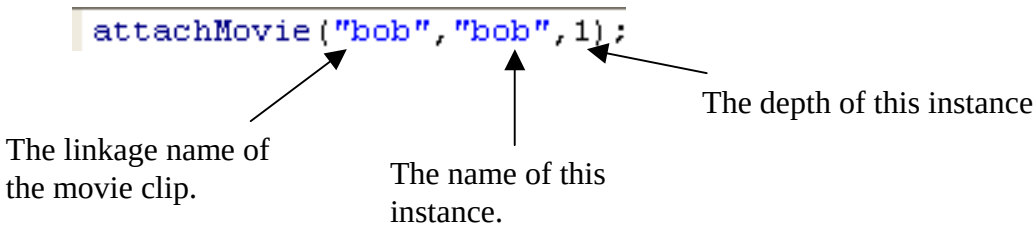
You should get the following dialogue box.



When you click on Export for ActionScript a name should appear in the identifier box. (If it does not, type a name in). You will now be able to attach bob using actionscript.

Let's test that this works. In your movie, delete bob from the main stage. If you have done this properly, the pictures layer in the timeline should be white

Click into the **actions** layer in frame 1. Before the lines setting bob's x and y positions, we now need to add some script to actually attach bob to the movie. The `attachMovie` script has 3 parts to it.



The linkage name is just the name that was given above.

The instance name is the thing that we typed into the **properties** panel earlier. This is the name that we will need to use when we try to write code for instance.

The **depth** of the movie clip is rather like a layer. Every movie clip that you attach must go on its own layer. If you try to attach 2 clips with the same depth, the second one will simply replace the first. The greater the depth, the higher up the layers a clip is, so the object of depth 2 will appear on top of the object with depth 1.

If you put the above line at the start of the code for Frame 1, you should have

```
attachMovie("bob","bob",1);

bob._x=_xmouse;
bob._y=_ymouse;
```

If you run this movie it should appear identical to the previous one.

We can, however, now add more instances of bob. The script below will add an extra bob, that will stay put, but will be faint (just to prove that we really can set its properties).

```
attachMovie("bob","bob",1);
bob._x=_xmouse;
bob._y=_ymouse;
attachMovie("bob","bob2",2);
bob2._alpha=50;
```

Attaching Multiple bobs

If you want to attach a clip lots of times, you can save a lot of work by using a **for….loop**. These look confusing at first, but their structure is always the same. Let's first look at the whole script and then examine it piece by piece.

```
for(i=1;i<6;i++)
{
    attachMovie("bob","bob"+i,i);
    this["bob"+i]._x=50*i;
    this["bob"+i]._y=100;
}
```

Let's take it line by line.

```
for(i=1;i<6;i++)
```

We will look at the first line properly in a minute, but to start with all we need to know is the first term in the bracket defines a **variable** – i – whose value is 1. The important set of instructions is inside the curly brackets.

```
    attachMovie("bob","bob"+i,i);
```

The first line of these instructions is just like the attachMovie instruction that we used above, but with 2 changes.

First, the instance name is **`"bob" + i`**.

But we have said that **i** has a value of 1, so this means that the instance will have the name **bob1**.

Secondly, the depth is **i**, but again, since i = 1, this just means that the depth is 1.

```
    this["bob"+i]._x=50*i;
```

This line is setting the x position of **bob1**. The "`this`" is a reference to the fact that in Flash you can have various different levels, with movies within movies, within movies. Sometimes when you want to identify a movie clip you need to go up a level and sometimes you have to go down a level. In this case we have a simple movie, on only one level, so we say `this`.

You might wonder why we don't just call the movie **bob1**. Inside the **for…loop**, if something has been defined in terms of i, it must always be referred to in terms of i.

You will also notice that the x position is given by 50*i. This means that for **bob1** the x position will be 50.

```
    this["bob"+i]._y=100;
```

The final line just sets the y position of **bob1** as 100.

Now that we know what is going on in the curly brackets, let's look in detail at the first line of code.

```
for(i=1;i<6;i++)
```

We have already seen that the first part of the code defines a variable i, with a value of 1.

The last part of the bracket is

```
i++
```

This is a shorthand way of saying “increase i by 1”. (We could have used i+=1).

When you run the **for…loop** the code is first carried out, using `i`=1.
After the code has been executed, `i` is increased to 2 and the whole loop is repeated.
This means that a movie **bob2** is attached, with an x position of $50 \times 2 = 100$.
We then repeat the loop with `i`=3, to attach **bob3** with x position 150.
This keeps on happening, as long as `i`<6.

If you run the movie, you should get 5 bobs, spread out across the screen.

Make the bobs chase the mouse

Now let’s make all 5 of our bobs chase the mouse. We will do this by adapting the code in Frame 2.

The first thing to do is to make the first bob chase the mouse. The existing code refers to **bob**, so we need to change this to **bob1**.

```
xDistance=_xmouse-bob1._x;
yDistance=_ymouse-bob1._y;

speed=0.5;

bob1._x+=xDistance*speed;
bob1._y+=yDistance*speed;
```

You should now find that the first bob chases the mouse, but the rest will just sit where they are.

We want **bob2** to chase **bob1**, **bob3** to chase **bob2** and so on. We can do this with the following code.

```
for(i=2;i<6;i++)
{
    xDistance=this["bob"+(i-1)]._x-this["bob"+i]._x;
    yDistance=this["bob"+(i-1)]._y-this["bob"+i]._y;

    this["bob"+i]._x+=xDistance*speed;
    this["bob"+i]._y+=yDistance*speed;
}
```

The first time through this loop we take I to be 2. This makes the first line effectively

```
xDistance = bob1._x – bob2._x
```

so the line finds the distance from bob1 to bob2.

The third line moves bob2 half of this distance in the x direction.

As we run through the loop, each instance of **bob**, closes in on the previous instance of **bob**, creating a slinky effect.

Improving the code (again)

What if we want to attach more bobs. We would need to change the number in the **for….loop** in Frame 1. If we want to have 10 bobs, the first line of our loop would need to be:

```
for (i=1;i<11;i++)
```

If you make this change and play the movie, you will find that 10 bobs are attached, but only 5 will chase the mouse! This is because the **for…loop** in Frame 2 only goes i<6. This illustrates the problem mentioned earlier.

The solution is to define a variable at the start (bobs = 10) and use this in both **for…loops**. This will change the codes in Frames 1 and 2 to

Frame 1

```
bobs=10;

for(i=1;i<bobs+1;i++)
{
    attachMovie("bob","bob"+i,bobs+1-i);
    this["bob"+i]._x=50*i;
    this["bob"+i]._y=100;
}
```

Frame 2

```
for(i=2;i<bobs+1;i++)
{
    xDistance=this["bob"+(i-1)]._x-this["bob"+i]._x;
    yDistance=this["bob"+(i-1)]._y-this["bob"+i]._y;

    this["bob"+i]._x+=xDistance*speed;
    this["bob"+i]._y+=yDistance*speed;
}
```