Introduction to Interactive Media and Video Game Design with Macromedia Flash MX

Introduction to Interactive Media and Video Game Design with Macromedia Flash MX

TABLE OF CONENTS

Introduction

This is a book for people who know absolutely nothing about computer programming or interactive media but who want to start creating highly interactive websites and games as quickly as possible. It is designed to be completely comprehensive and self-contained.

This book has also been designed to keep in mind the wildly fluctuating state of the technology and the fact that if you happen to learn the wrong technology or the wrong series of techniques at the wrong time, all your hard earned skills will be completely out of date in 6 months time. A very specific effort has been made to concentrate on techniques that have so far withstood the test of time, and will most likely continue to do so. It is very likely that essence of the techniques you will learn in these pages will still be applicable 5 years from now. Certain areas of the technology have started to stabilize, and it is those areas that are the focus of this book.

What is required is some knowledge of how to use a computer (understanding concepts such as "turning it on" and "double-clicking" help.) And, a *basic* working knowledge of Macromedia Flash MX is essential.

How basic? Well, if you've spent a few hours going through the built-in lessons in Flash, and a few more hours playing around with some of those techniques, that's all you need to know. You need to know how to use the drawing tools to create simple graphics, how to do basic animation, how to convert graphics to symbols, and also understand the differences between *symbols* and *instances*.

(*Are you lost yet? Don't worry, turn on your computer, open up Flash MX, click on* ***Help*** *and choose* ***Lessons****. Follow each lesson step by step and come back to this book when you are finished! It will only take a few hours.)*

Perhaps even more enlightening is what you *don't* have to know to be able to make use of this book:

- Math
- Computer Programming
- Website Design
- Graphic Design
- … or practically anything else!

In fact, I'll even allow you the odd passing comment along the lines of "*I hate computers*" or, even worse perhaps, indulge you in a fantasy of hurtling some particularly heavy, blunt object at your monitor. Rest assured that the author of this book has shared exactly those same feelings at some point or another!

I also want to quickly dispel the myth that to be an interactive website or game designer you need an innate talent or share some kind of extra-sensory relationship with electronic equipment. "Talent" is only what other people say you have when they haven't seen all the countless hours of hard work and patience that you've put into a skill. As for the extra-sensory relationship – you can fake that by reading this book!

So, given that this book assumes you know as close to nothing about the subject as possible, how long will it take before you start creating your own interactive web sites and video games? Somewhere between 40 to 60 hours – so probably a few months if you spend and hour or two going through these lessons and exercises every day. All it takes is a little time, a little patience, a willingness to learn, and hopefully you'll find it easy. This book is intended to be cutting-edge, without the pain.

And, the payoff? What you're about to learn is just about the closest thing you can get to creating magic that the real world allows. Hang on for a wild ride – you'll be amazed by what you're going to start producing very quickly.

- *Rex van der Spuy*

Part 1:
Introduction to Interactivity and Non-Linear Media

How to Create a Basic Interactive Flash Movie - Introduction -

The following steps explain how to create an Interactive Flash Movie. This can be used for something more complex, such as an interactive web site, CD ROM presentation, interactive storybook, or even a simple game. In the past, creating an interactive movie was something that was traditionally done using software called Director. When people talk about creating “interactive multimedia”, they usually mean something similar to what you will be learning below.

These exercises assume that you have completed all the built in lessons in Flash. If you haven’t, open Flash, click the **Help** menu and choose **Lessons**. Work though the lessons one at a time; this is the best way to learn Flash. It won’t take you long, and the lessons are lots of fun. If you find the lesson on Buttons confusing, don’t worry, you can skip it. We will cover all you need to know about buttons in these exercises.

Up until now, all the work that you have done in Flash has been **linear**. A linear movie is one that starts at the beginning and finishes at the end, much like a storybook or film. In the following exercises you will be creating a **non-linear** movie. In non-linear movies, the user (the person viewing or “using” your movie) decides what they want to see and when they want to see it.

You can think of the difference between a linear movie and a non-linear movie as the same difference as that between a novel and an encyclopedia. Novels are **linear**; the plot of the story follows a straight line. And even though you could possibly jump to the end of the novel or start reading it at chapter 10, you would probably never want to because it would either ruin the story for you or you wouldn’t understand what was happening.

With an encyclopedia, on the other hand, you only need to turn to the page on which the information you need is presented. The information is **non-linear**; it doesn’t follow a straight line. If you were researching Ancient India, for example, the fact that you hadn’t started reading from the beginning of the encyclopedia makes no difference at all to understanding the information you’re looking for. In fact, if you had a research report due the next day, and you had to read volumes A to I of the encyclopedia in their entirety before you got to India, you may well still be sitting in the school library long after all your friends have graduated and gone on to university!

So, what does an encyclopedia have that a novel doesn’t have? An index system. To create an index system, you need the following things:

- An index system needs **storage areas**: The information must be stored in the correct place. In an encyclopedia, the storage areas would be the page numbers and section headings. In Flash, the storage areas are created by **Frame Labels**.
- You need a method of getting to the right page. With an encyclopedia, you would use the index and flip through the pages until you found the right section. Of course, your computer screen doesn’t have any pages to flip through, so

Flash uses **Buttons** to jump to the correct section.

- You need to be able to stop at the right page when you get there. What do you use to stop at the right page? Well, in real life, the device you would used is called “your brain” but, unfortunately, Flash isn’t as smart as you, so you need to use **Actions** to tell it when to start and when to stop.

Creating an interactive movie in Flash is not difficult, and is the basis for creating much more complex software like games, CD ROMs and highly interactive websites.

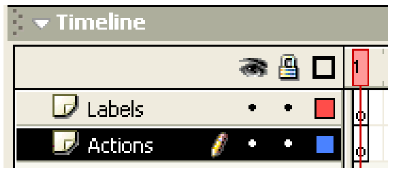
Carefully follow the steps in the following sections to create your first simple interactive movie.

How To Create a Basic Interactive Flash Movie
PART A: Setting Up Your Frame Labels

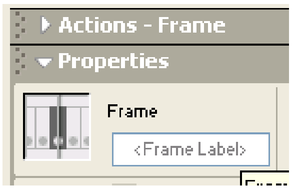
*The first step in creating an interactive movie is to set up **Frame Labels**. "Labeling a frame" means giving that frame a name. This is important because the computer needs to know which frame to jump to when a certain action, such as clicking a mouse, occurs. This is similar to a section heading in an encyclopedia – it's where your information will be stored. You can create as many frame labels as you need.*

1. Create a new Movie in Flash, and save it as "Interactive Movie".

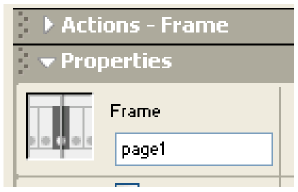
2. Create 2 new layers. Name one layer **Labels** and the other layer **Actions**. These are special layers that will be used to control the sequencing and organization of the movie. You will always create these layers when you need to make an interactive movie, and you should get into the habit of creating them as soon as you start a new movie in Flash.



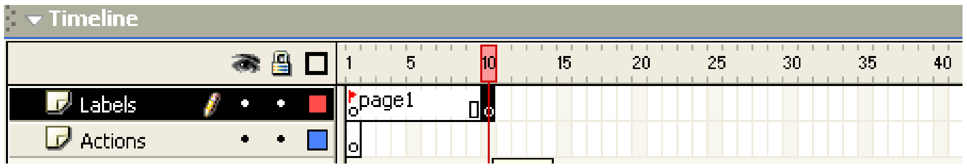
3. Click on the first frame of the **Labels Layer** to make it active. Open the **Properties** panel at the bottom of the Flash work screen (click on it once to open it if it isn't already open). Find the *field* (a "field" is a text input box) called **Frame**.
*(For Flash 5 Users: On the right hand side of the Flash work screen, you should see 4 sets of **Panels**. The panels control various settings and options. Look carefully, and you will see a panel called **Frame**. Click on this panel to make it fully visible, if it isn't already.)*



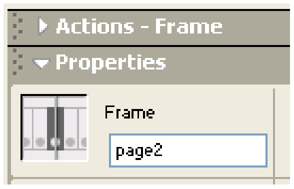
4. Enter the word "page1" in the Frame field. For simplicity's sake, make sure all the letters are lowercase and that there are no spaces.
*(Flash 5 Users: Enter "page1" in a field called **Label** in the **Frame** panel)*



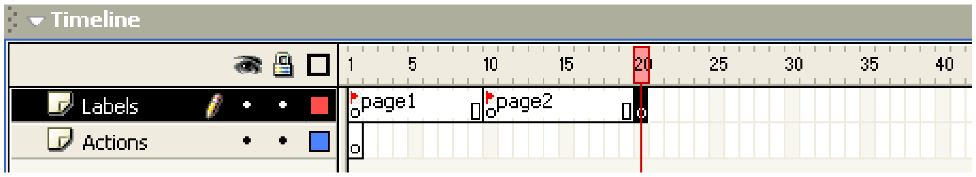
5. On the **Labels Layer**, click on **Frame 10**. Insert a **keyframe** by pressing **F6**. (You can also insert a keyframe by clicking on the **insert** menu at the top of the screen and then choosing **Keyframe** from the list of options. Pressing **F6** is much faster, however, and, as you will be creating keyframes quite a lot, it's useful to use this shortcut)



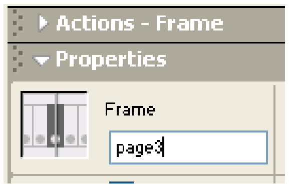
6. Make sure that frame 10 is highlighted as in the picture above. In the **Frame Panel**, enter "page2" in the **Frame** field.



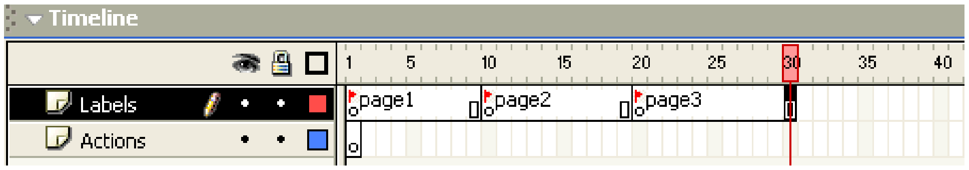
7. On the **Labels Layer**, click on frame number 20. Insert a **keyframe** by pressing **F6**.



8. In the **Frame Panel**, enter "page3" in the **Frame** field.



9. Click on Frame 30 on the **Labels Layer** and press **F5**. **F5** inserts blank frames, and you can achieve the same effect by clicking on the **Insert** menu (at the very top of the screen) and choosing **Frame.** This step inserts an additional 10 blank frames so that you can read the label's name. You don't really need to include this step for the interactive movie to work, but your work will be much easier to understand if you do.



Remember, keyframes indicate that something has changed in your movie: the graphics, the frame labels, or the actions. Blank frames merely continue whatever was changed in the last keyframe; they extend its duration.

You now have labels for each of your 3 frames. We called our labels “page1,” “page2,” and “page3,” but you could have given them any name that was useful to you. If you were creating a web site on Ancient Asia, you might have labeled your frames “india,” “china,” and ‘japan.”

Naming Conventions

It’s probably not too early to start talking about **naming conventions**. Naming conventions are a standard set of rules that that programmers and software developers use so that they don’t make mistakes when giving things names, such as frame label names in Flash. (And, oh yes, if you then think that this doesn’t apply to you because you’re not a programmer or software developer, I have news for you: if you’re following these instructions, you are!)

The reason software developers use naming conventions is because “software development platforms” (things that you make software with, in this case, Flash) are usually very picky about giving things exact names.

Here’s an example. Let’s say you labeled the first frame of you movie “ancientindia.” Then, let’s say you programmed your button to jump to a frame called “Ancient India.” Are they the same? Well, you might think they are, and all of your friends might too, but Flash considers them to be completely different names. Did I mention that Flash’s “intelligence” is somewhere between an underdeveloped blade of grass and a rather precocious pebble?

Well, it is. If you don’t tell Flash something EXACTLY it will have no idea what you’re talking about. And, it won’t even tell you that it doesn’t understand. That means, if you make one tiny mistake in any of your names, your movie won’t work.

This is the reason we use naming conventions. If we always label names in the same way, we’re far less likely to make mistakes. Below is a list of naming conventions that I use. You’re welcome to use your own system… but I recommend you use mine! It works well.

- **Single Words:**
  Use lowercase letters, without spaces. If you use numbers, <u>never</u> use them at the beginning of the name, always at the end:

```
india
china22
```

- **Multiple Words:**
  The first word should be lowercase. The second word, and any other following words, should start with an uppercase letter. Never use spaces; they always cause problems somehow:

  ```
  ancientIndia
  bangaloreSewageSystem
  aroundTheWorldInEightyDaysChapter2
  ```

And, try to keep your names short so that you're less likely to make simple typing mistakes ("aroundTheWorldInEightyDaysChapter2" is actually too long, by the way. I just put it in there to show you what you should avoid. Can you see why?) Make sure that your names are descriptive, but try to keep them under 3 words.

You won't like using this system at first, but try and force yourself. Almost half of the mistakes that beginners make when they start using Flash are that they have forgotten what or how they've named something. If you use a standard naming convention, and stick to it, you'll encounter far, far fewer problems.

And, oh yes, you may notice that I don't seem to follow any naming convention with **symbols**. Well, that's probably just sloppiness on my part (you probably should use a naming convention,) but it's also because **symbol names** don't communicate with anything else – they just sit there in the library looking pretty.
What you really need to worry about are **instance names**. Instance names can be linked to actions, and so it's crucial that you name them correctly and consistently. But more on that later….

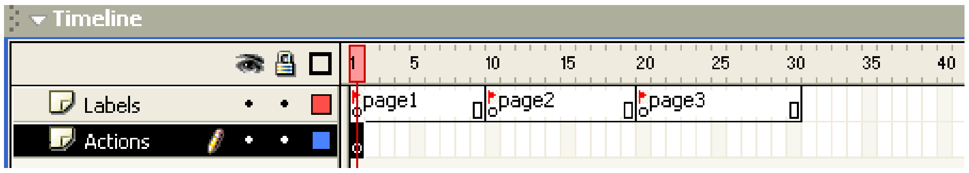
How To Create a Basic Interactive Flash Movie
Part B: Setting Up Frame Actions

*This next step is your first introduction to Flash's programming language called **ActionScript**. What you will be doing is telling Flash to stop playing the movie whenever it encounters the first frame of a labeled frame.*

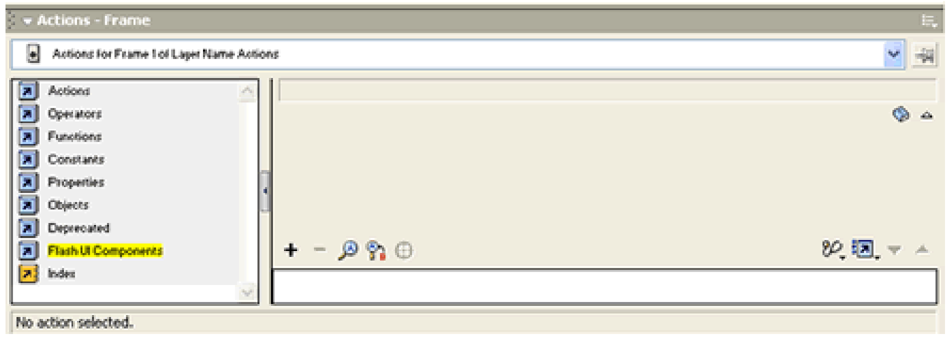
This is important, because otherwise you won't have any control over when the pages are displayed - they'll simply play in sequence without stopping, like an ordinary Flash animation.

1. Click on the first frame of the **Actions Layer** so that it is highlighted. Next, click on the **Actions Panel** (just below the stage) to open it.
   *(For Flash 5 Users: Double click on the first frame of the **Actions Layer**. This opens up the **Frame Actions** panel, which is usually hidden. You can also open the Frame Actions panel by clicking on the **Window** menu and choosing **Actions.**)*

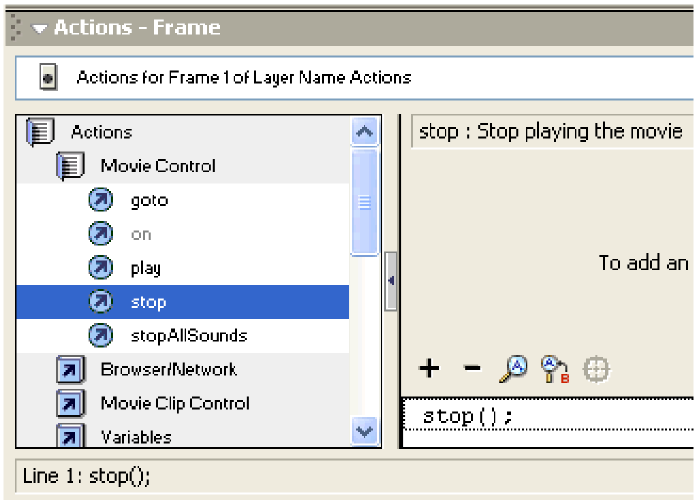
   Your timeline should look like this:



   The Actions Panel should look like this:



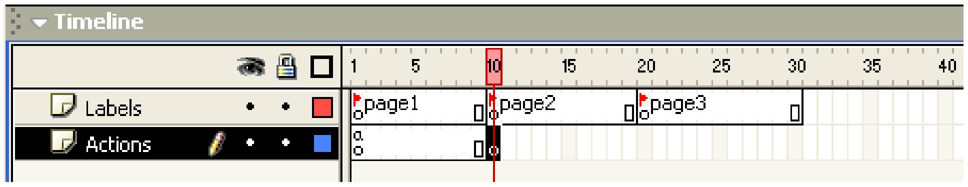
2. In the **Actions Panel,** Click once on **Actions** and then again on **Movie Control**. Double-Click on **stop**. You should see "`stop ();`" appear in the **Script Pane** (the mini-window just below the plus and minus signs.) The movie will now stop at this frame.
   *(For Flash 5 Users: Click on **Basic Actions.** Choose **stop** from the list and double-click on it. You should see "`stop ();`" appear on the right side of the panel.)*



```
stop();
```

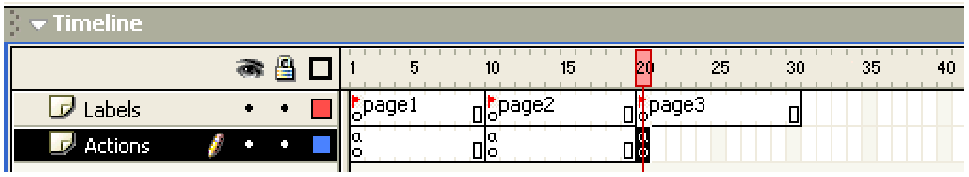
Line 1: stop();

3. Click once on **Frame 10** of the **Actions Layer** and press **F6** to add a keyframe.



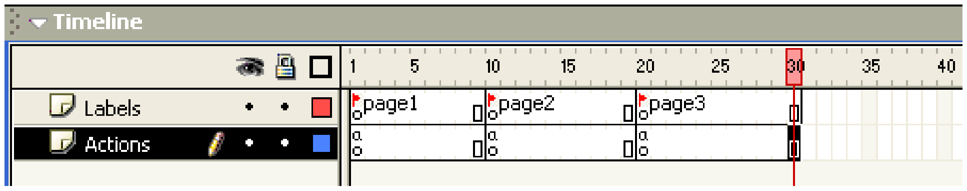
4. The **Frame Actions** panel should still be open. Click once on **Actions** and then again on **Movie Control**. Double-Click on **stop**. Make sure that you see "`stop ();`" appear in the **Script Pane**. <u>This is exactly the same procedure that you followed in step 2.</u>

5. Add a keyframe at **Frame 20** of the **Actions Layer**:



6. Add the **stop action** to this keyframe as well, in exactly the same way you did in steps 2 and 4. You should be an expert at this by now! Notice that each time you add an action to a frame, a small "a" symbol appears in the frame. This tells you that the frame contains an action.

7. Click on **Frame 30** and press **F5** to add blank frames. This is the same technique we used earlier to "fill out" the rest of the frames.



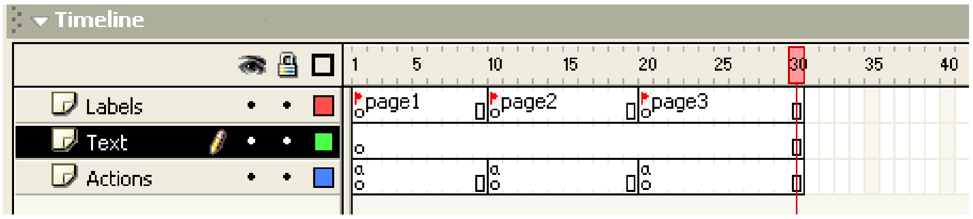
We don’t really need to add these final bank frames because they don’t contain anything and don’t do anything. However, as you’ll see in the next sections, there’s a very good reason for them to be there.

How To Create a Basic Interactive Flash Movie
Part C: Add Your Content.

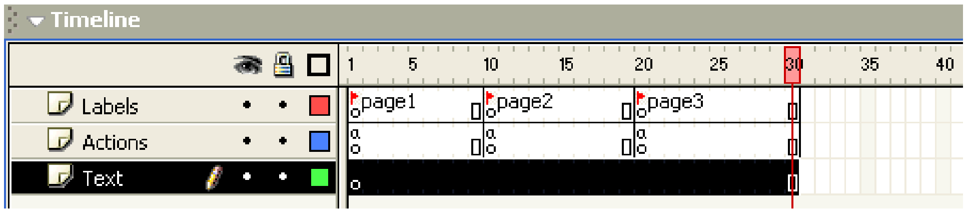
The next thing to do is to add something to each page of your movie for your viewers to see. In almost all cases, from now onwards, you will be adding Graphic Symbols and Movie Clip Symbols. Any animation you want to add should be in the form of a Movie Clip Symbol. <u>Avoid using any animation that is done directly in the main timeline</u>. Your main timeline should be reserved only for setting up your different sections, organized using labels and actions.

In this simple exercise, you will add some plain text to each labeled frame of the movie. This is similar to filling the pages of your encyclopedia with words and pictures.

1. Create a new layer and name it **Text**. (You can call it anything you like, however. But remember, layer names should always describe as clearly as possible what the layer is being used for.)

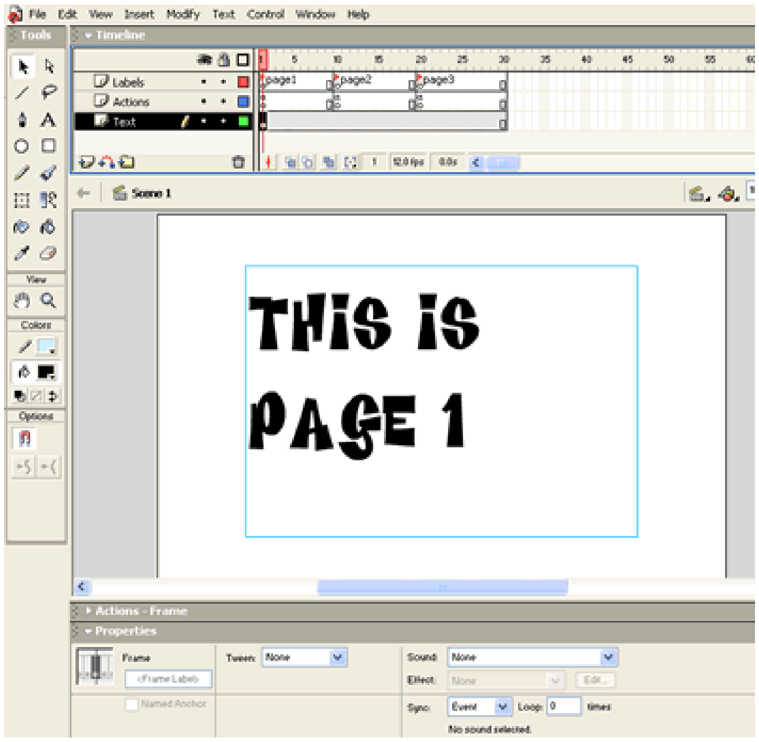


2. Move the **Text Layer** so that it is <u>under</u> the **Labels** and **Actions Layer**

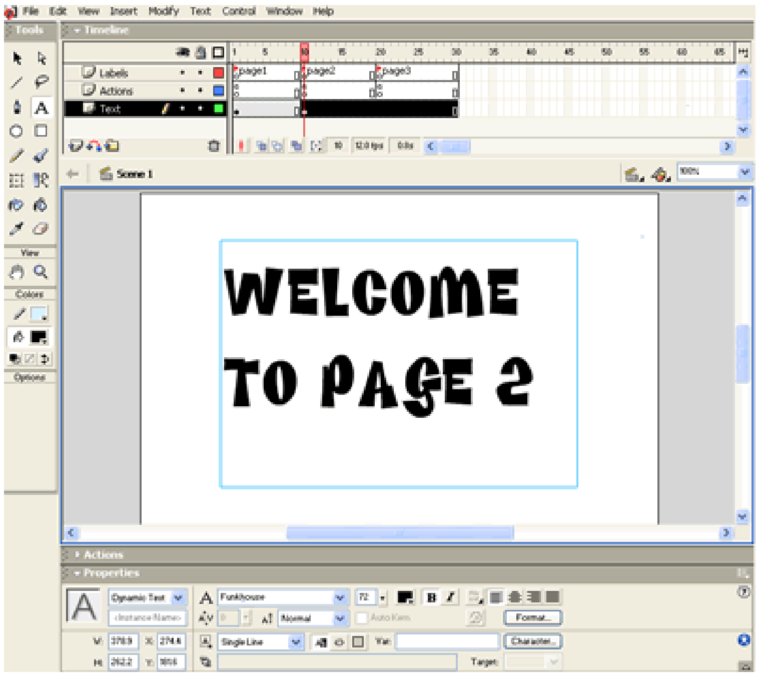


The **Labels** and **Actions Layer** should always be the first two layers of your movie so that you can find them easily.

3. Click on Frame 1 of the **Text Layer**. Use the **Text Tool** from the drawing toolbar and type the words: "This is Page 1."

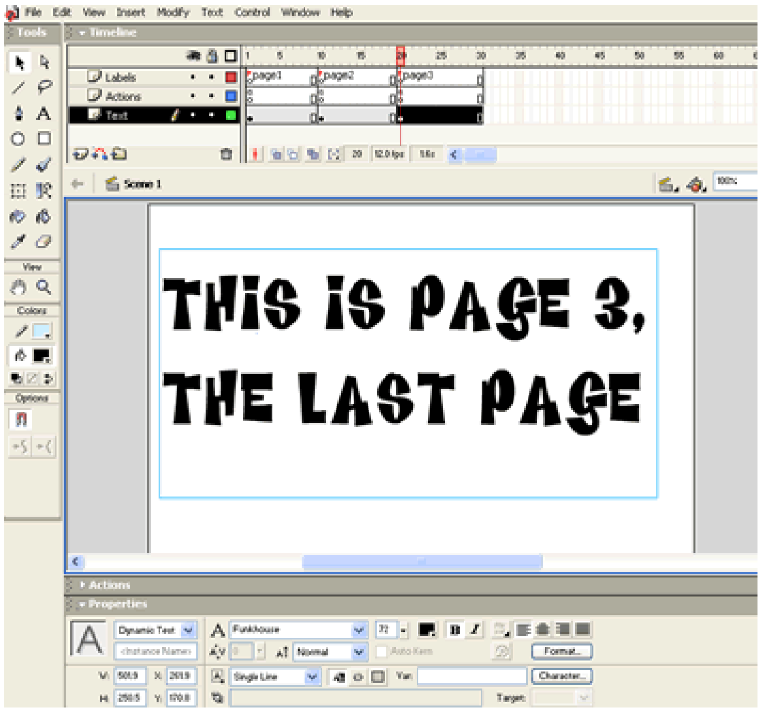


4. Add a new **keyframe** on **Frame 10** of the **Text Layer** (press **F6**). On Frame 10 of the **Text Layer**, add the words: "Welcome to Page 2" with the text tool. You'll know that you've added the keyframe correctly if you see a small black dot in Frame 10.

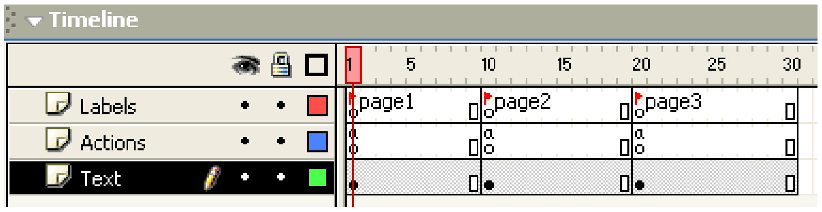


5. Add a new **keyframe** on Frame 20 of the **Text Layer**. Add the words: "This is

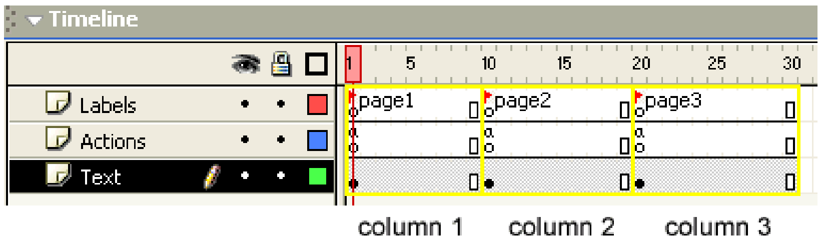
Page 3, the last page."



When you are done, your timeline should look something like this:



Have a look at it closely. Do you notice something about the way the information has been organized? You might notice that **columns** are beginning to appear. Each “page” of your movie is organized into its own column.



Your movie is no longer just organized horizontally, it is also organized **vertically**. Every column contains one page of your movie, and is completely separate from the others.

This is essential to understanding how non-linear, interactive multimedia works. You need to stop thinking in terms “My movie starts here and stops there” and start thinking in terms of the **columns of information** that you want to keep separated.

When you are designing interactive movies, it’s extremely important that you make sure your separate pages of information line up in neat columns like those above. One of the most common mistakes that is made by people when creating their first interactive movies is that information from one column (an image, a sound, or an action) accidentally overlaps into part of another column. If you create your movie carefully and make sure that your columns are neatly defined, you won’t run into nearly as many problems.

Because we added **stop** actions to the beginning of each frame, there is no way for any of the pages to interact with the others. “page2” won’t play after “page1”. You can observe this by testing your movie at this point (hold down **crtl** and press the **Enter** key, or click on the **Control** menu and choose **Test Movie**). You should only see Page 1.

But what if we want “page2” to come after “page1”? Or, better still, what if we want to start at “page1,” view “page3” and then jump back to “page2”?

That’s what buttons are there for, and you’ll learn how to create and program buttons in the next sections.

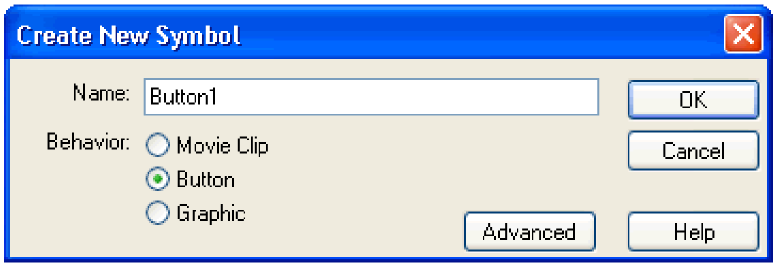
How To Create a Basic Interactive Flash Movie
Part D: Create Your Buttons

*Buttons are what you use to create your **navigation control**. When you click on a button, the button will take you to another **labeled frame** of your movie.*

Buttons are much like the index of an encyclopedia, with the added extra that you don't need to flip through pages by hand to find what you are looking for – the button takes you straight there.

Although you could use any of the pre-made buttons in Flash's "Common Libraries," it's almost always best to create your own. The following steps explain how to do this:

1. Click on the **Insert** menu and choose **New Symbol*.***

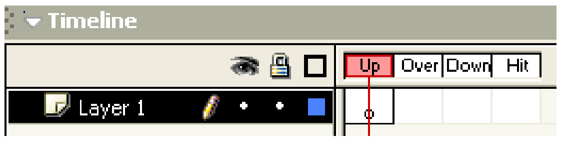


Name: Button1

Behavior: ( ) Movie Clip [✓ Button] ( ) Graphic

2. Choose ***Button*** as the "Behaviour" and give it a name (You could call it "Button1".) Click ***OK*** when you're done.

3. The Symbol Editing window will open.



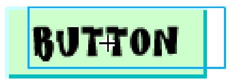
Button Symbols have their own unique timeline with only 4 frames. These frames work in a very special way:

UP: How the button looks when it's not being pressed.
OVER: How the button looks when the mouse moves over it.
DOWN: How the button looks when you click on it with the mouse.
HIT: The area of the button that is sensitive to the mouse. (The HIT area is invisible when the movie plays.)

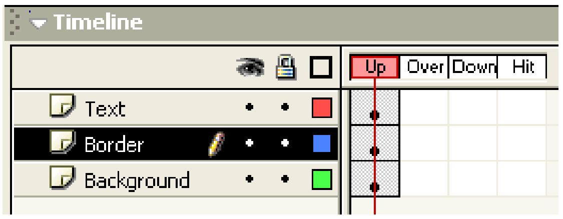
You don't use actions to control when these frames are displayed. The button frames that are displayed are entirely based on how the user interacts with the button using the mouse.

The next steps involve drawing each frame of the button:

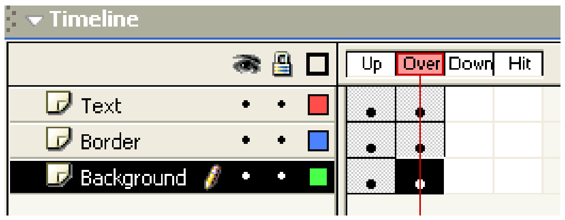
4. Click once on the **Up** frame and draw something that looks like this:



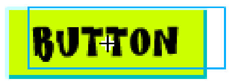
Make sure the text, background colour and button border are all on *different layers*. Don't forget, you can create as many layers as you need to with buttons, just as you can with ordinary graphics. Any elements such as text or colours that are likely to change should be on different layers.



5. Click on the **Over** frame. Add keyframes to all the layers :

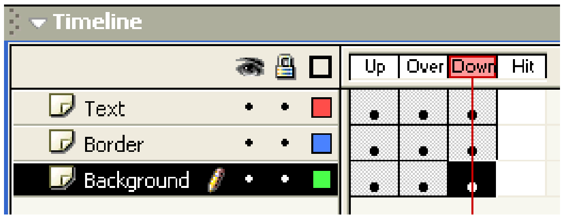


6. Now change the colour of the button's *background*, so that it's slightly different from the first frame:

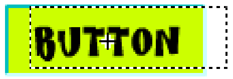


This is how the button will look when the mouse moves over it. Often, designers make the background a little brighter, to show that the button has become active.

7. Next, add keyframes to the button's **Down** frame on all the layers:

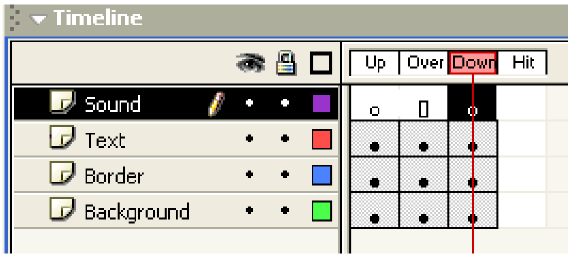


8. Change the button so that it looks like it is being pressed down. The most common way to do this is to make the background darker. Or, you can use this trick:

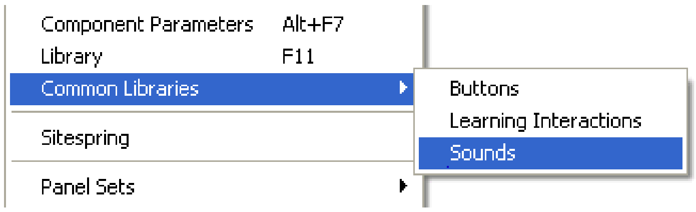


Compare this button image with the image we created in step 6. Do you see the difference? The border colours have been reversed and the text has been moved slightly down and to the right. This creates a very realistic 3D button effect. The button will really look like it is being pushed down.

9. Let's add some sound to our button. Create a new layer called **Sound**, and add a keyframe on the **Down** frame.



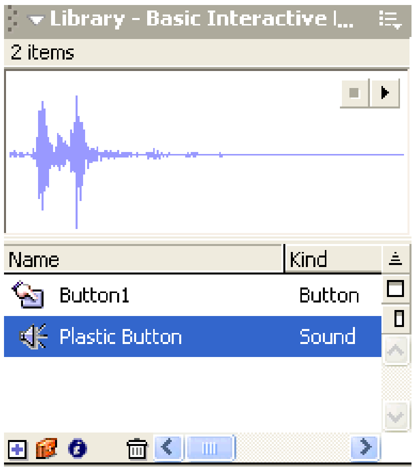
10. Click on the **Window** menu at the top of the screen, find the **Common Libraries** option and click on **Sounds**.



A new library will open up, with the sounds that come included with Flash. You can easily import your own sounds into your library, but for now we'll use these built in ones.

11. Look through the sound library and find a sound called **Plastic Button**. Hold down the mouse button over it and *drag* the sound into your own library. You

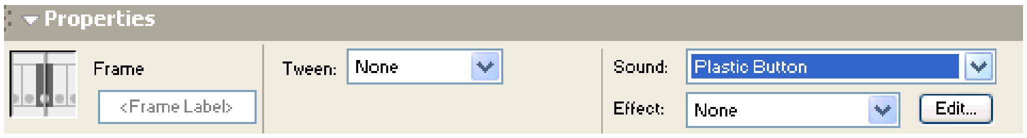
should now have 2 symbols in your library: **Button1** and **Plastic Button**.


Library - Basic Interactive I...
2 items

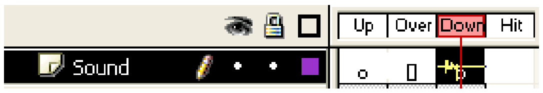
Name	Kind
Button1	Button
Plastic Button	Sound

If the Plastic Button symbol is highlighted, you can click on the play button at the top right corner of the library window to play the sound.

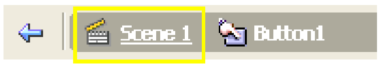
12. Make sure that the **Down** frame of the **Sound Layer** is still highlighted. Choose **Plastic Button** from the **Sound** menu. Any sounds that are in your library will be available here.
*(Flash 5 Users: Find the **Sound** panel and choose the same sound from the drop-down list.)*


Properties
Frame
<Frame Label>
Tween: None
Sound: Plastic Button
Effect: None

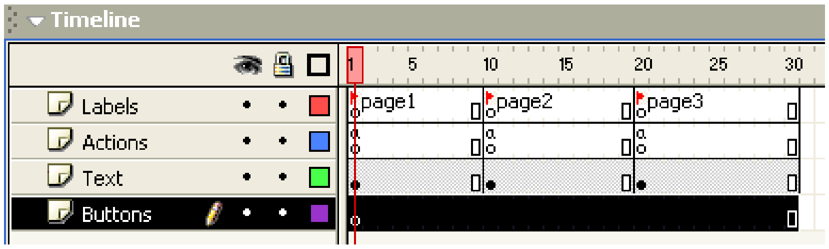
13. Check to make sure that the *wave pattern* (visual shape of the sound… some squiggly lines) is visible in the frame.



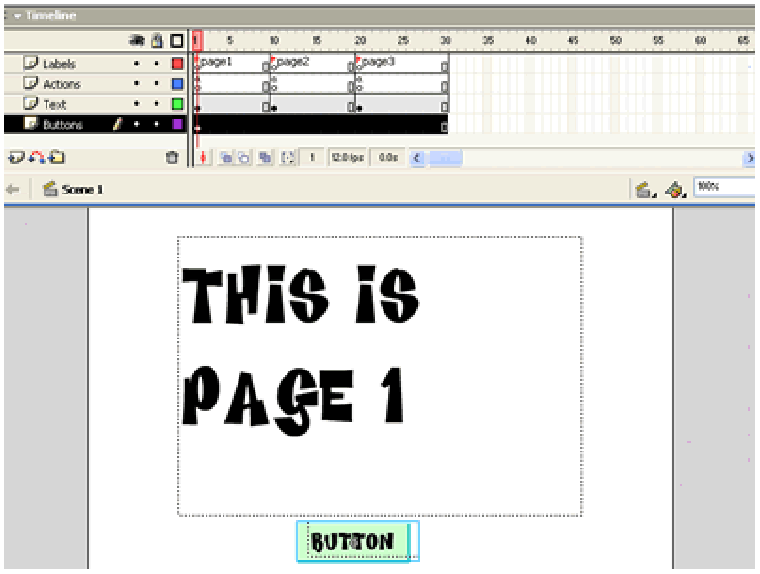
14. Click on **Scene 1** to exit the button symbol editing window and return to the main timeline



15. Make sure that you are on Frame 1 of you movie, and create a new layer called **Buttons**.



16. Drag **Button1** from the library onto the main stage. Make sure that it's on the **Buttons Layer**.



17. Test your movie. You should now have a fully functioning button.

What about the Hit Frame?

You usually don't need to add anything in the **Hit** frame, unless you want your button to be sensitive to an area that is larger than the images you have placed in the previous frames. Whatever you draw in the **Hit** frame, however, remains invisible when the movie plays – which is good. It's just there to define the shape of your button.

As an experiment, try creating a button that only contains text (without a background or border.) When you test it, you will notice that the mouse is only sensitive to the exact shapes of the letters, not the general area around it. If you then try drawing a rectangle the same height and width of you letters in the **Hit** frame, the button will react when the shape is detected. The shape itself, however, will always be invisible when your movie plays.

More Than Just Graphics?

It's important to remember that you don't just have to add graphics to the button frames. You can add animated movie clip symbols if you want to. You can also add different sounds to any of the button frames. Experiment a little, and with a little imagination you'll be amazed at what you come up with.

But it Still Doesn't Take Me to the Next Page!

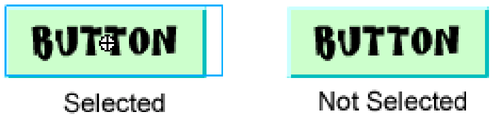
I know. We've made the button, and we can click on it, but we still haven't told it what it needs to do when we click on it. You can think of what we've just created as a key on a keyboard that hasn't been plugged into the computer yet. You can press the keys, but they don't make the computer do anything.

That's the next step…

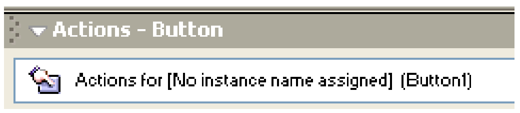
How To Create a Basic Interactive Flash Movie Part E: Program Your Buttons

You will have noticed that when you tested the movie your button didn't actually do anything productive. It may have looked nice, but it didn't take you to the next page. What you need to do next is ***program*** *your buttons. You have to tell them what to do when someone clicks on them.*

1. Click once on the first frame of your **Buttons Layer**. Next, <u>*and this very important*</u>: click once *on the button instance on the stage*. You need to make sure that the button itself is selected, <u>not the frame</u>. If the fame is selected, the next few steps won't work. You'll know that you have selected the button if there's a blue rectangle around it.



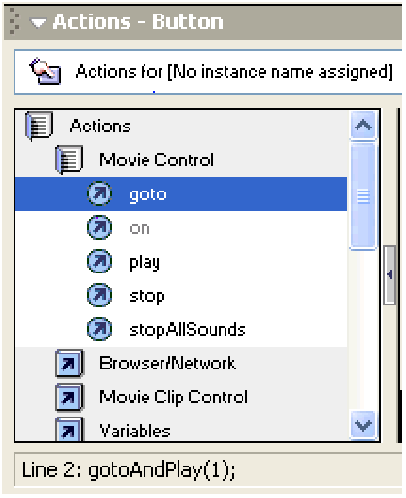
2. Open the **Actions Panel**. It should say **Actions – Button** in the title bar. In Flash, you can have Frame actions, Button actions or Movie Clip Actions. Somewhat confusingly, they all use the same panel, and you have to be very careful to make sure that the right object has been selected.


Actions - Button
Actions for [No instance name assigned] (Button1)

(Flash 5 Users: Open the ***Object Actions*** *panel by either clicking on the* ***Window*** *menu and choosing* ***actions*** *or by clicking on the small blue arrow at the bottom of the stage.)*

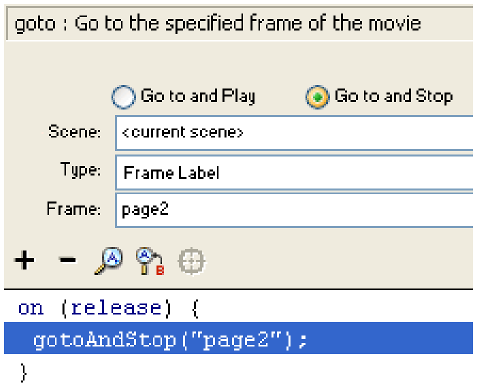
NOTE: *If the new window that opens says "Frame Actions" it means that you haven't properly selected the button. Click on the button on the stage once to select it. When the window reads "Button Actions" (or "Object Actions" for Flash 5 users) you're on the right track.*

3. Double-click on the **goto** action from the Movie Control actions. **Goto** is an action that is used whenever you want to jump to a different frame. *(Flash 5 Users: Make sure that the **Basic Actions** menu is open. Double click on **Go To.** New parameters become available at the bottom of the window that you can change. Make sure that "`gotoAndPlay ("1");`" is highlighted in dark blue - it should be. If it isn't, click on it <u>once</u>.)*



```
Line 2: gotoAndPlay(1);
```

4. Change the parameters in the **Actions Pane** so that it looks like this:



( ) Go to and Play [✓ Go to and Stop]

Scene: <current scene>

Type: Frame Label

Frame: page2

```
on (release) {
 gotoAndStop("page2");
}
```

What are we doing here? Well, let's think about what we want our button to do: When we click on it, we want it to go to and stop at the frame that we've labeled "page2." So, all we're doing is telling Flash, step by step, exactly what it has to do to accomplish this.

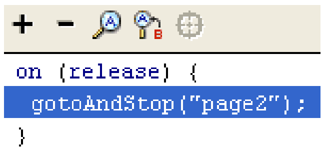
First, we need to tell it that it should **stop** at the frame we choose. (Although if you ever wanted it to start playing from a particular frame, you could select "Go to and Play.")

Next, we want to make sure that we jump to a **Frame Label**, <u>*not*</u> a **Frame Number**. We *could* make our button jump to a specific frame number in the movie, and there are some cases where this might be a good thing. However, for most interactive movies that you'll be making, you'll almost always want to jump to a **Frame Label**.

Why? Because Frame Labels will always remain the same no matter what frame number they are on. That means that if you ever reorganize your movie and change the frame number of you labeled frame, your movie will still work. Also, Frame Labels are easier to use and remember, because they tend to describe the content on that frame. "Frame 10" has really no meaning, but "ancientIndia" does. If you were working on a CD ROM with hundreds of pages of information, you could easily forget what was on frame 460. Wouldn't it be much easier if that frame was called "navyBlueTshirts"?

The last thing we need to do is to tell Flash which labeled frame it needs to jump to. Conveniently, you'll find all the labeled frames from your movie listed in the **Frame** drop down menu.

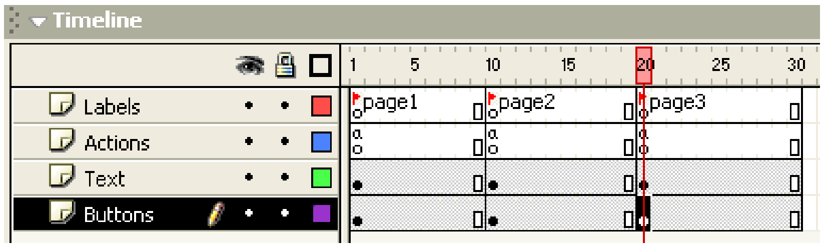
"But what's this???"



```
on (release) {
  gotoAndStop("page2");
}
```

Oh. That. I was hoping you wouldn't ask that question. Remember earlier I mentioned that Flash uses a programming language called ActionScript? Well, what happens when you choose parameters from the Actions Pane is that Flash automatically converts your selections into ActionScript programming code. When you play your movie, Flash is actually reading the ActionScript code; that's what that distressingly math-like-looking stuff is. In later lessons, you'll learn how to program ActionScript code directly yourself. Don't be afraid! It only looks scary because you don't understand it. Once you understand it, however, you'll actually find easier to use than choosing selections from the drop-down menus – I promise! But, for now, you can safely ignore it.

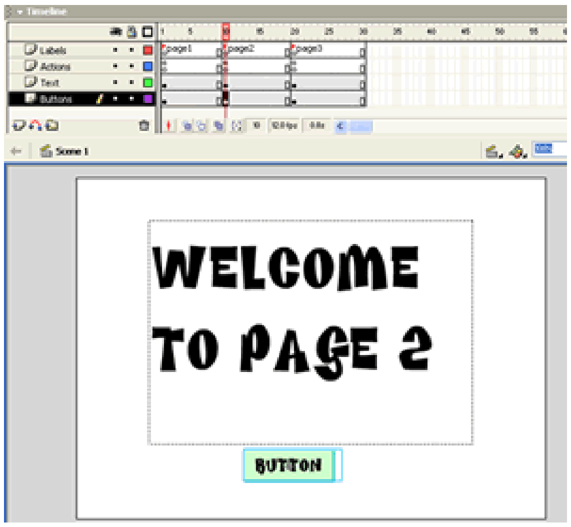
5. The next thing you need to do is to create **keyframes** on the **Buttons Layer** at frames 10 and 20.



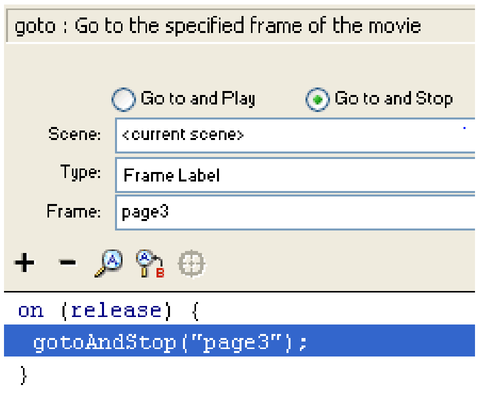
Make sure that you have 3 neatly organized columns.

6. Make sure that **frame 10** is highlighted and click the button on the stage with the arrow tool. Although this button looks the same as the button on frame 1, *it isn't*.

<u>Because we created a new keyframe, we've created a new button *instance* so we can change the way this button behaves without affecting the buttons on frame 1 and frame 20.</u>



7. Open the **Actions Panel** for the button on frame 10. Change the parameters so that they look like this (if you can't find any parameters to change, click on "`gotoAndStop("page2");`" in the **Actions Pane** and they should appear):



goto : Go to the specified frame of the movie

Go to and Play [✓ Go to and Stop]

Scene: <current scene>

Type: Frame Label

Frame: page3

```
on (release) {
 gotoAndStop("page3");
}
```

You only need to change one thing: change "page2" to "page3" in the **Frame** field.

8. See if you can figure out the final step for yourself: Program the button on Frame 20 so that, when someone clicks on, it jumps back to "page1".

Test your movie and see if it works.

You now have all the skills you need to make a simple interactive book in Flash. "But," you're thinking, "this all looks a little *linear* to me. I mean, sure, I can click on the buttons to go from one page to the next, but I can't go back or jump to a completely unconnected page."

Actually, if you think about it a little bit, you have all the skills to do just that. I'm sure you can figure it out….? But, just to help you out a little, have a look at Part F, the next section.

How To Create a Basic Interactive Flash Movie
Part F: Creating a Navigation Bar

Sometimes, you don't just want to use buttons to jump from one page to next. Sometimes, or even usually, you want them to act as an index to the contents of your movie.

*Think of an ordinary website that you use on a regular basis. On the website there are probably a set of buttons that, when you click on them, take you to a new page, but never disappear from the site. Usually you'll find these buttons on the top or left hand side of the page. As a whole, these buttons are called a **Navigation Bar**.*

Navigation bars are the key to making interactive movies in Flash that contain a lot of content, like a website or informational CD ROM. They act like an encyclopedia's index. You always know where to find them and they always take you to the page or section you want to visit.

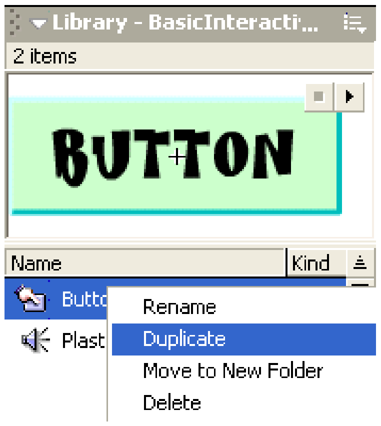
Creating a navigation bar is really quite simple. All you need to do is create a layer of buttons that are visible across every section of your movie (every column defined by labeled frames) and whose functionality doesn't change with each section. That means that a button called "Ancient India" always takes you to the page called "Ancient India" no matter which other page you are on.

Carefully follow the instructions below to modify the movie you were working on in Parts A – E to create a navigation bar:

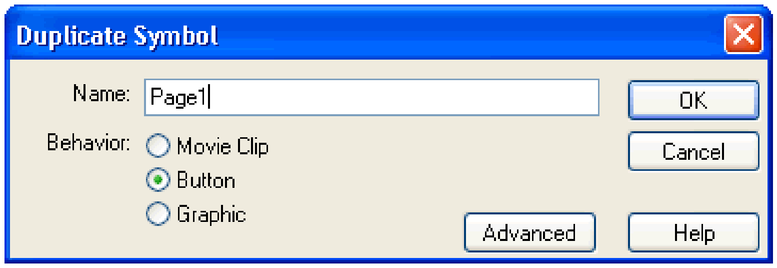
Duplicate Your Button

The first thing we're going to do is make a few copies of the button we created in the previous steps:

1. Make sure the **Library** is open. *Right-click* on the **Button1** symbol. A new selection menu will open. Choose **Duplicate**.



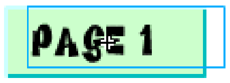
2. Change the **name** of the new button to "Page1". Make sure that the **behaviour** is set to **Button**.


Duplicate Symbol

Name: Page1

Behavior: ( ) Movie Clip [✓ Button] ( ) Graphic

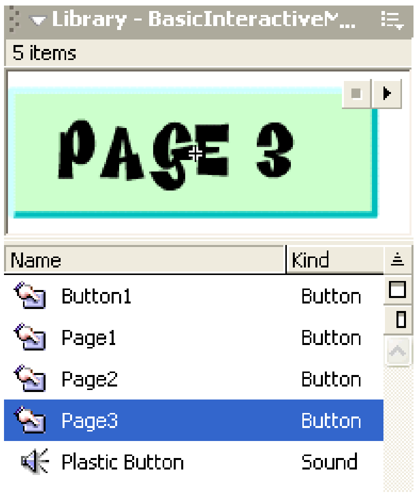
3. Double-click on the new **Page1** symbol in the Library to open its symbol editing window. Change the button's text, in all three frames, to "Page 1".



4. Duplicate the **Page1** symbol in the same way that you duplicated the first button. Give this new symbol the name "Page2".

5. Double-Click on the **Page2** symbol to open up its symbol editing window and change the text on all three button frames to read "Page2"

6. Following the same procedure, create a third button called "Page3." You should now have 3 new button symbols in the library called **Page1**, **Page2** and **Page3**.


Library - BasicInteractiveM...

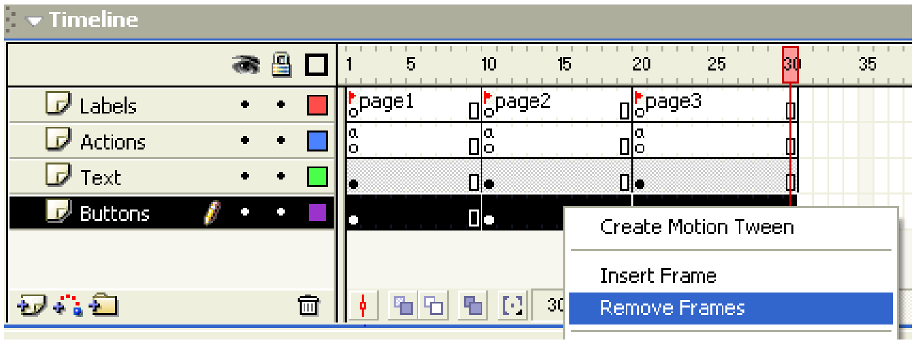
5 items

Name	Kind
Button1	Button
Page1	Button
Page2	Button
Page3	Button
Plastic Button	Sound

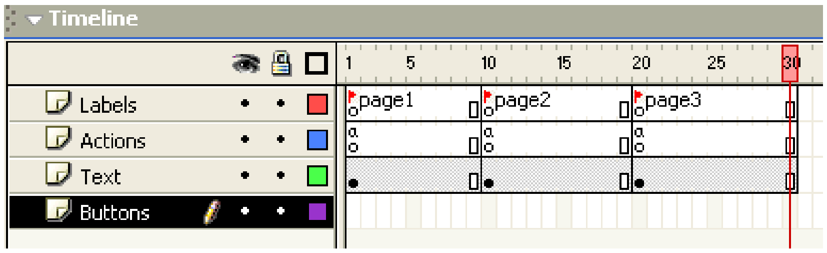
Add Your New Buttons

1. Return to the main timeline (Scene 1.) Delete all the frames on the **Buttons Layer**. The easiest way to do this is to:

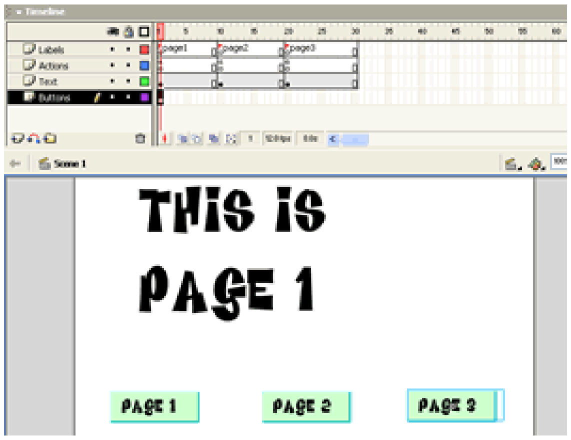
- Highlight all the frames with the arrow tool.
- Right-click on them.
- Choose **Remove Frames** from the option menu.



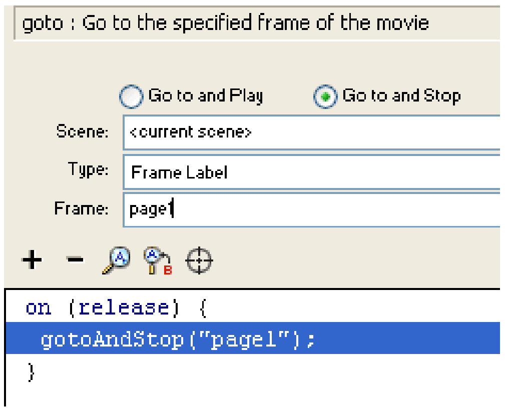
2. Your **Buttons Layer** should now be completely blank, like this:



3. Insert a **keyframe** on the first frame of the **Buttons Layer**. Then, drag one copy each of the **Page1**, **Page2** and **Page3** buttons from the library onto the stage so that they line up neatly near the bottom. (You *must* insert a keyframe on the first frame of the **Buttons Layer**, because Flash will not let you insert an object into an empty frame.)



4. The next step is to program our buttons so that when the user clicks on them, they take him/her to the correct page. Now, you *should* know how to do this by now, but, just in case you need a reminder, this is what the button actions should look like for the **Page1** button:



goto : Go to the specified frame of the movie

Go to and Play [✓ Go to and Stop]

Scene: <current scene>

Type: Frame Label

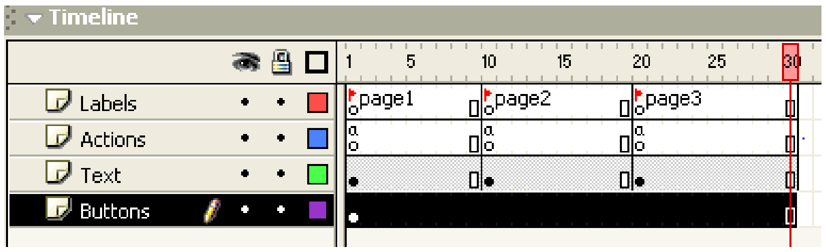
Frame: page1

```
on (release) {
  gotoAndStop("page1");
}
```

Remember: *Make sure that the <u>button</u> is selected, <u>not the frame</u>. To select the button, choose the arrow tool and click on the button once. When you see a blue bounding box surrounding it, you know it is selected. If these instructions don't work, it probably means that you accidentally selected the frame.*

5. Repeat the above steps for the **Page2** and **Page3** buttons. The only thing that you will need to change is the frame name.

6. Click on frame 30. Press **F5** to add blank frames across all three columns of your movie, like this:



You might be thinking that this defies what we discussed earlier about creating neatly organized columns in the previous section. But just think about it for a moment: a navigation bar is something that is visible in *all* sections of your movie. No matter what page you're on, you'll need to see the same buttons.

7. Test the movie and see what happens…a perfect navigation bar! It's visible on all pages, and every button you click on takes you exactly where it promises.

What you have learnt in these exercises are the basic procedures for making websites, interactive storybooks, informational CD ROMS and even simple games with Flash. It doesn't get much more complicated than this. As long as you remember to keep your columns of information separate, while at the same time making sure that your navigation system (your buttons) are available across all columns, you can't go wrong.

Assignment: Interactive Storybook

Use the basic Flash interactive techniques you have learnt so far to create a 6 page storybook for young children. Make sure you include the following:

- Create a title Page.
- Each page should contain a drawing and some text.
- There should be buttons on each page that allow the reader to advance to the next page or read the previous page.
- When the story is finished, give the reader a chance to return to the beginning.
- Your buttons should containing functioning “Up,” “Over,” and “Down” states.
- When you are finished, **publish** your movie as a ***windows projector*** file.
- If you choose, publish your story as a web page and upload it onto your site.
- The elementary students will be reading these stories, so make sure that your subject matter is suitable for them.

EVALUATION:

2 MARKS: For a title page and 5 pages of content (text and drawing.)
2 MARKS: For properly creating your buttons, with all states functioning properly.
2 MARKS: For programming your buttons properly to allow for adequate navigation.
2 MARKS: For an interesting and original children’s story.

TOTAL: /8

Assignment: Interactive Multimedia Project

The final project of the course is an Interactive Multimedia project designed in Flash. The project will give you a chance to utilize all the skills you have learnt during the year. It should be the most interesting and sophisticated piece of work you create, and will be worth 20% of your final grade.

You will have the choice of doing one of any 5 projects. Read the project descriptions and criteria carefully and decide which project is right for you.

All projects will be marked out of 25, based on the evaluation criteria for each project.

1. Animated Music Jukebox

Use the interactive capabilities of Flash to create an Animated Music Jukebox. A jukebox is a device that plays different songs depending on which button is pressed. The jukebox that you create should not only play songs, but present the user with a visual animation to accompany the music.

Things to Consider:

- You can decide on whatever sort of animation you would like to use: it can either be cartoon-like or be a purely abstract visual interpretation of the music with lines and shapes.
- Will your "box" look like a real machine (such as a CD Player) or can you think of a less conventional way of containing the music (such as creating a map of an island having different music playing depending on which part of the map the user clicks on.)

Criteria for Evaluation:

- You must use at least 5 songs.
- Well-designed buttons to allow the user to choose a song.
- Design a visually interesting "box" to showcase the music.
- Create interesting animations to accompany the music.
- You must publish your movie as a windows projector file and copy it onto a CD.

2. Choose-Your-Own-Adventure

Create a simple adventure game where the player has to guide an animated character, using buttons, through a series of problems and hazards to reach a goal. For example, if your character was being chased by a dragon and had reached the edge of a cliff, you could give the player the option of jumping into the river below, or fighting the dragon.

Things to Consider:

- Will your game be in “first-person” perspective (such as from the player point of view, like Quake or Half-Life) or will it be “third person” (such as Mario or Zelda, where you guide a character)?
- Always give the player more than one way of reaching his/her goal. If they fail at one point of the story, is there another way for them to succeed later?
- Make your game challenging, but not too hard – players will give up if they find it too difficult to win.
- Decide on your theme. Will it be a mystery story, a mediaeval fantasy, a science fiction adventure, or a real-world drama?
- Consider designing your game first as a *flowchart* on a big piece of drawing paper before you begin. Making a flowchart is like drawing a map of what each scene will be like, and how the scenes will connect depending on the decisions the player makes. For a complicated adventure, this step is *crucial* for your game to work properly.
- If you want to, create clues for your player to collect, and have them use those clues to come up with an answer that will help them win the game. For example, ask the player to type in the secret password to open the magically locked door to the treasure room. To do this, you will need to know a little more advanced Flash ActionScript. Don’t worry – it’s not hard – just ask your teacher to show you.

Criteria For Evalutation:

- Did you create an interesting and well-designed world.
- Is your game hard enough to be challenging but easy enough not to be frustrating?
- Is there more than one way to win, or achieve a secondary goal?
- Are your choices clear, and can the user play again if they lose?
- Publish you movie on the internet and as a windows projector file.
- You must use sound somewhere in your game.

3. Action Game

You may be surprised to know it, but with the knowledge you have, you can create a very basic action game. The trick to doing this is by making buttons that move around the screen so that the player has to chase them with the mouse. When the player clicks on a button, your movie could advance to another scene (another level) and that new level could make the game more difficult, or indicate the score. Of course, your buttons don’t have to *look* like buttons, they can be any shape you want: cars, spaceships, or anything you can think of.

Things to Consider:

- For this to work, you need to put a button into a Movie Clip Symbol and then animate that movie clip symbol on the main stage.

- When the user clicks on a button, Flash will advance to a new “labeled frame”. You can put anything you like on this new scene: it could be a new level, the score, or more moving buttons.
- Decide on the theme of your game.
- By learning a little more basic ActionScript, you can control Movie Clip Symbols in your movie by clicking on buttons. This greatly adds to the complexity and interest of your game. Ask your teacher if you would like to learn how to do this – it’s not hard.

Criteria For Evaluation:

- Was your game interesting and challenging?
- Did you have an interesting design and concept for your game?
- Did all your buttons and programming techniques work properly?
- You must use sound in your game.
- You must give the user a chance to play again.
- You must have an introductory screen that gives the user a chance to press “play” or “start” to start the game.
- You must publish your game on the internet and as a windows projector file.

4. Multimedia Web Site

Flash can do everything HTML can and more. Use the skill you have acquired to create a web-site using only Flash.

Things to Consider:

- The theme or purpose of your site.
- How will you use animation and sound?
- Be careful not to use too much sound: any sounds more than a few seconds will take too long to download.
- Make sure you have a consistent Navigation Bar.
- It is easy to add “links” to other web sites in Flash. Ask your teacher for help with this.

Criteria For Evaluation:

- Was your site visually interesting?
- Did you use animation and sound well?
- Did you use at least 5 pages of information?
- Was your navigation bar easy to use?
- Were your buttons well-designed and properly programmed?

5. Make Your Own Project:

Maybe you've been dying to do something all year and have just been waiting for the chance. Well, here's your chance. Complete the form below, discuss it with your teach, and go for it.

Name:______________________________________

Project Description:

Criteria For Evaluation:

1.____

2.____

3.____

4.____

5.____

6.____

7.____

8.____

9.____

10.____

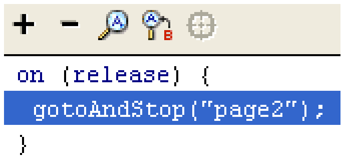
Part 2:
Introduction to ActionScript

What is ActionScript?

ActionScript is a **computer programming language**. It's a language, much like any other language that you might know, like English, Japanese or Hindi. However, you don't use it to communicate with another person; you use it to communicate with a machine – a computer. It's a language that you understand and your computer understands. You use it to tell your computer what it has to do. Fortunately, however, the computer can't use it to tell you what you have to do!

There are lots and lots of different programming languages. Some that you might have heard of before are BASIC, Pascal, C++ and Java. ActionScript is the programming language that Flash uses, and it is very closely related (and, in many ways, identical) to Java and C++. That may not mean much to you now, but keep it mind because, if you ever go on to study computer programming further, what you learn in ActionScript will provide you with a huge advantage. In terms of its flexibility, ease of use, and seamless integration with Flash, it's the best programming language that I can think of for beginners to start learning.

So what does it look like? Well, you might remember seeing something like this in earlier lessons:



```
on (release) {
  gotoAndStop("page2");
}
```

The ActionScript code is the just below the plus and minus signs. Whenever you choose options in the Flash's Action Pane window, Flash is actually converting your selections into ActionScript.

"So, if Flash is writing the ActionScript for me, why do I need to learn how to program with it myself?"

That's an excellent question. For basic actions, such as programming simple buttons or starting or stopping frames, you don't need to learn ActionScript at all. However, our final objective is to be able to create highly interactive software and games. We need a degree of control and flexibility that we can't get unless we program our ActionScript directly. There are no shortcuts. But, as you'll soon see, there don't need to be. ActionScript is easy!

But I'm Terrible At Math!

So is the person who is writing this book. One of the biggest misunderstandings non-programmers have about computer programming is that programming is like math. It's not – at all. It might look the same on the surface, and some of the syntax has been borrowed from mathematics for matters of convenience, but the whole underlying system is completely different.

That's not to say you won't be using any math in these lessons – you will. How much? Well, have you ever come across rather abstract concepts called *multiplication* and *division*? If not, I can refer you to an excellent grade 3 level textbook on the subject. That's as complex as the math gets. These lessons specifically teach programming from a non-math point of view.

That's not to say it can't get more complicated, however. It can get as complicated as you'd like it to be – but it doesn't have to. In Part 4 of this book, when we start applying our ActionScript programming to game design, some of you might be interested in creating spaceships or cars that can be rotated on the screen using the keyboard. For that, you'll need to use some fairly advanced trigonometry. However, the object of all these lessons is how to *use* the math, not necessarily in being able to *understand* it. That's what high school math classes are for! But, think of it this way: you can turn on and turn off a TV without having to know how the TV actually works. In the same way, you can use math that you may not necessarily understand to achieve certain effects – you just need to know where to insert it in the context of the programming code that you understand.

What These Lessons Are…

These lessons in ActionScript are for intermediate Flash users who have a basic grounding in creating Interactive Flash Movies; they know how to make animations and control movies with simple buttons. If you've followed Part 1 of this book, that's you! These lessons are for people who know nothing about programming, but who want to start using ActionScript to quickly create games and other sophisticated, highly interactive software with Flash. They are for people who want to start using the tools as soon as possible, without having to struggle through a 1000 page textbook on computer programming hoping that they might come across the scrap of information they need. They are for people willing to overcome their belief that they "can't program" and who are willing to trust me when I say that "*Programming is EASY – Don't be afraid of it!"*

What These Lessons Are Not…

They're not a comprehensive introduction to computer programming. Ok, so there is a place for that 1000 page textbook after all…. You will probably find that after you have completed all these lessons, you will want to learn more. The best basic introduction to ActionScript on the market is Foundation ActionScript, by Sham Bengal. Anyone who completes these lessons will find that book very easy to follow and understand. You might also want to consider taking a basic high school level course in programming. It doesn't matter which programming language you learn (the concepts are the same for all of them) just make sure it's a very basic course. You'll find that you'll be able so apply your skills immediately to Flash.

Your humble author is also working on a greatly expanded version of these lessons that teaches programming and computer science with ActionScript as the primary language, so you might also want to check keep an eye out for that in the future.

What You Will Learn

All the basic ActionScript programming concepts and skills to eventually build pretty sophisticated action games. And that's quite a bit!

Just take it slowly, carefully and, above all, don't be afraid!

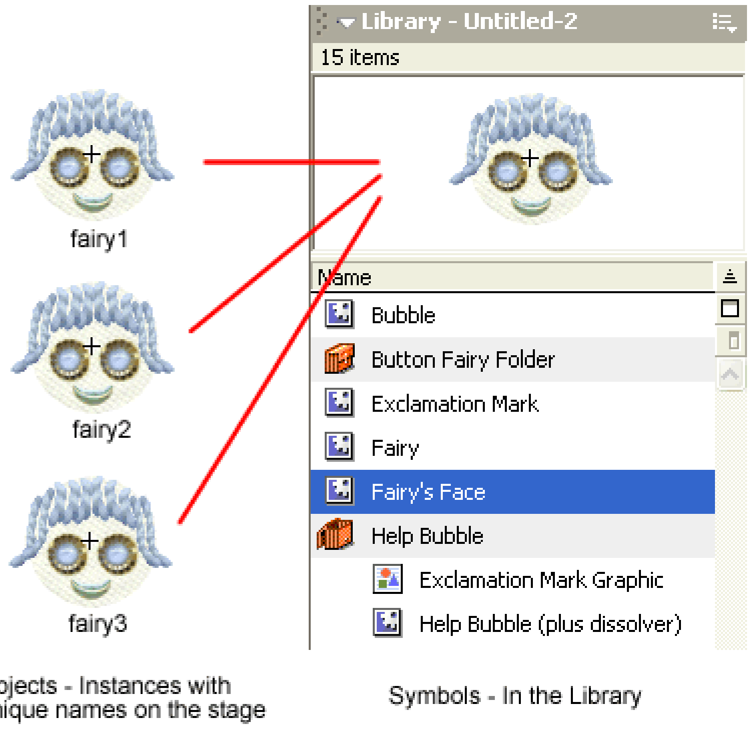
Introduction to ActionScript 1: Objects

ActionScript is an **object oriented** programming language. That means that **objects** are the main programming element.

What is an object? Every time you drag a **symbol** from the library onto the stage, the **instance** of that symbol on the stage becomes an object. (If you need a quick refresher on the difference between symbols and instances, click on the **Windows** menu in Flash and choose **Lessons**. Choose "Creating and Editing Symbols" in the new window that opens and follow the instructions.)

You can think of the symbols in the library as cookie cutters. You might spend a lot of time making the cookie cutter, but once it's finished you can make as many cookies as you like. Instances are like the cookies that your cookie cutter, the symbol, makes.

You already know how to create an object in Flash, even though you might not realize it. Whenever you create an **instance** of a **symbol** on the stage, and give it a name, you are creating an object.



In the diagram above, there are three **instances** on the stage: **fairy1**, **fairy2**, and **fairy3**. They were all made from the same **symbol** called "Fairy's Face" which lives in the library. Only **fairy1**, **fairy2**, and **fairy3** are **objects**. Why? Because they're on the stage and have unique names. Symbols in the library cannot be objects.

Why can't symbols also be objects? *Because symbols can't be controlled with ActionScript*. Only instances on the stage can be controlled with ActionScript. Think of it this way: You can eat the cookie, but you sure can't eat the cookie cutter! In the same way that you can't play with your toys until you take them out of the box.

Objects are pretty special things, and there are quite a few ways you can create and use them. The most common objects that you'll be using are **movie clip objects**. There are also lots of objects built into Flash that are not as easy to see as movie clip objects. Some of these that you might come across in the next few lessons are the **Key object** and the **Math object**.

Objects have the following features, which you should always keep in mind:

- Objects have **names**.
- Objects have **properties** that can be changed.
- Objects have **actions** that can be used to control themselves and other objects.
- Objects can have **variables** that you can create and change.

Names are the focus of this section. We'll be discussing **properties**, **actions** and **variables** in following lessons.

Dot Notation – On Earth…

Before we go any further in our discussion of objects, we need to talk a little about **dot notation.** Dot notation is the format in which object names are written. The concepts behind dot notation are crucial to understanding how to program with ActionScript. The biggest problem that beginners run into when first learning to program with ActionScript is that they don't fully understand how dot notation works.

Let's have a look at a very simple example. What's your name? My name is Rex.

```
rex
```

I Live in India

```
india.rex
```

Where is India? It's on the Earth.

```
earth.india.rex
```

(Didn't I tell you that learning how to program was much like learning a language?)

The examples above are examples of what my name would look like in dot notation. Can you see why it's called dot notation? All of the information is separated by "dots."

This should tell you a lot about how dot notation works - and it works exactly the same way in Flash as in this example.

First of all, my **name**:

```
rex
```

If I were an object (and I guess that in some strange sense I am!) this would be my object name at its simplest level.

However, I'm not the only Rex in the world. There's another Rex: my dog, who waits for me patiently in Canada. So, we have two Rex's:

```
rex
```

and

```
rex
```

How can we tell the difference? Well, one of them wags his tail and barks. I'll leave it up to you to guess which one. But, apart from that, we know that they live in **different places**. I live in India, so my proper object name might be:

```
india.rex
```

My dog lives in Canada, so his proper object name might be:

```
canada.rex
```

This is another basic feature of dot notation.   In front of the most basic bit of information, the object's name, we need to indicate where the object lives. You can add as much or as little information about where your objects live as you need to. Because both India and Canada are on the Earth, we could go one step further:

```
earth.india.rex
```

and

```
earth.canada.rex
```

The objects are still the same, even though we've described them a little more clearly. You'll always find that your objects share at least one common place – the biggest place (which, in this case, is the earth.) In Flash, the biggest place is the **main timeline**.

You can use this general model for naming objects with dot notation:

```
place.objectName
```

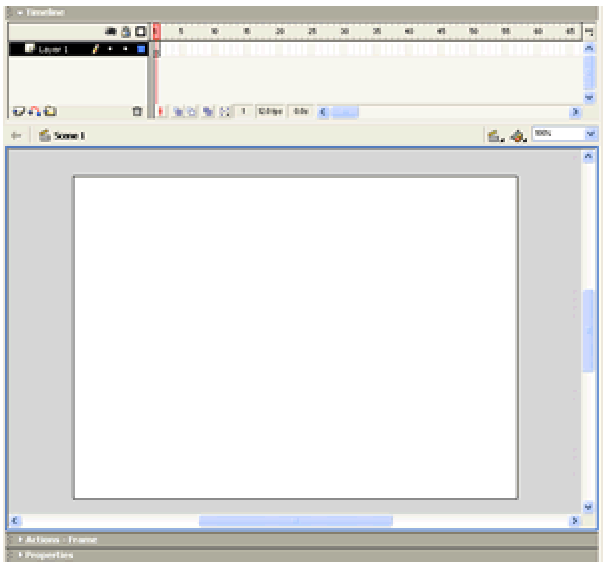
or

```
biggestPlace.smallerPlace.objectName
```

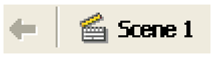
Dot Notation - in Flash…

This is how dot notation works in Flash:

Have a look at the following image.



This is the **stage** in Flash. It contains the **main timeline** (which some of you might think of as "Scene 1"), where you usually put most of your instances. If you look carefully at the top left corner of the stage, you'll see something that looks like this:



This shows us that we're on the **main timeline** – we're as far on the surface of our Flash world as we can get. If you remember the earlier example, this would the same as the "Earth" that my dog and I share.

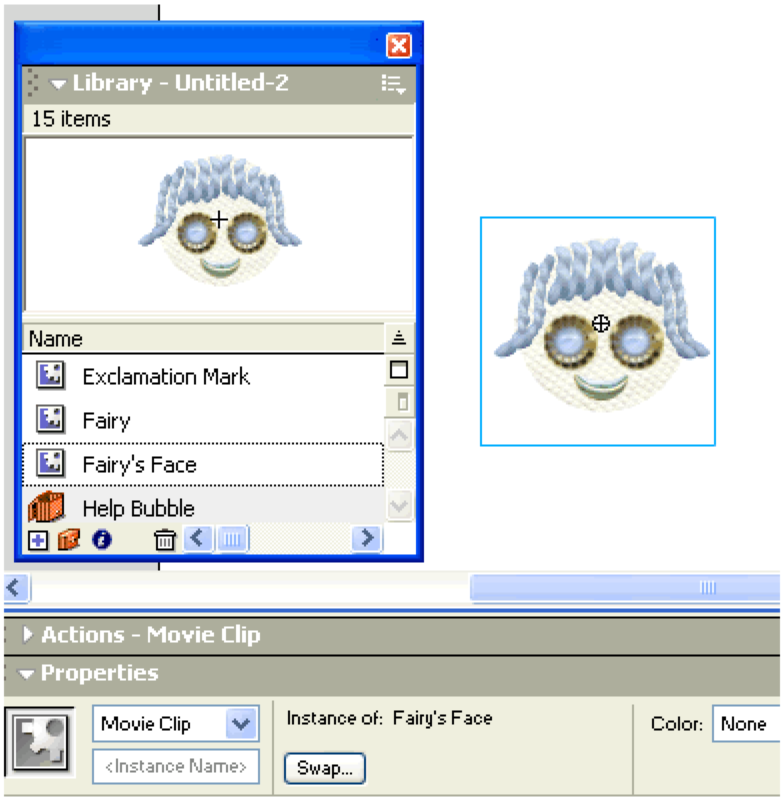
You might not know this, but the **main timeline** actually has a name. It's called:

```
        _root
```

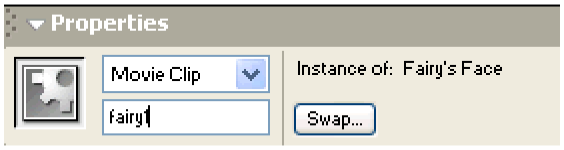
That's right. That's its actual name. Make sure you remember it; you're going to be using it a lot! Whenever you write anything in ActionScript and you use the word **`_root`**, you're actually referring to the **main timeline** and the main stage. It's extremely important to remember this.

`_root` is an **object**, and the most important one that you will need to know. It's not an object you need to create; Flash has created it for you. When you use the word **`_root`** in ActionScript, you can control what happens on the **main timeline**. You can also use it to control *objects that are on the **main timeline***.

Now, let's say that you place a new instance of the symbol "Fairy's Face" on the stage.



Select the instance (to select it, click on it once with the arrow tool) and open the **Properties Panel**. Enter "fairy1" in the field that says <`Instance Name`> *(Flash 5 Users: Open the **Instance Panel** and give the instance the name of "fairy1"):*



You have just created a new object. We can refer to the object using dot notation, which, again, simply means that we use periods ("dots") between the object names. The name of the object we have created is:

```
_root.fairy1
```

`_root` is where the fairy lives, on the main timeline, and **`fairy1`** is its name. Now, whenever you use **`_root.fairy1`** Flash will know exactly which instance you are talking about.

You can create as many instances of the "Fairy's Head" symbol that you want. You might have instances on the stage called:

```
_root.fairy1
_root.fairy2
_root.fairy3
```

They might all look the same, but as long as you give them different names, Flash regards them as completely different objects.

The Steps to Creating Objects

The steps to creating an object which can be controlled using ActionScript are as follows:

1. Create an instance of a Movie Clip Symbol.
2. Open the **Properties Panel** *(in Flash 5: the Instance Panel)* and give the instance a name.
3. The name should be lowercase, contain no spaces and follow the general naming conventions discussed in earlier lessons.

It is *extremely important* to remember the following:

1. You MUST give the instance a name.
2. If you are using Flash 5, the instance MUST be created from a Movie Clip Symbol. If you're using Flash MX, it can be a Movie Clip, Button or Textfield Symbol. Graphic Symbols can never be used to create objects (although you can use them to help you build other symbols, such as Movie Clips, that can be used as objects.)

By now you should understand that:

```
earth.rex
```

is a similar concept to

```
_root.fairy1
```

`earth` and `_root` both describe the places where the objects live. `rex` and `fairy1` are the objects' names.

Objects Inside Objects

But do you remember the problem we ran into earlier? There are two Rex's, and both live on the earth. Clearly, they can't both be called `earth.rex` otherwise we wouldn't know which one we were referring to. If you stood at your front door with a bowl of last night's leftovers calling out, "Earth.rex! Earth.rex!" you may well find yourself with two different Rex's running down the street and a terrible fight would ensue! Flash has the same problem, and if you give two objects the same name, it will get very confused.

But remember that the two Rex's are not exactly the same. Even though both live on the Earth, and both share the same name, one lives in India and the other lives in Canada. If we used the names:

```
earth.india.rex
```

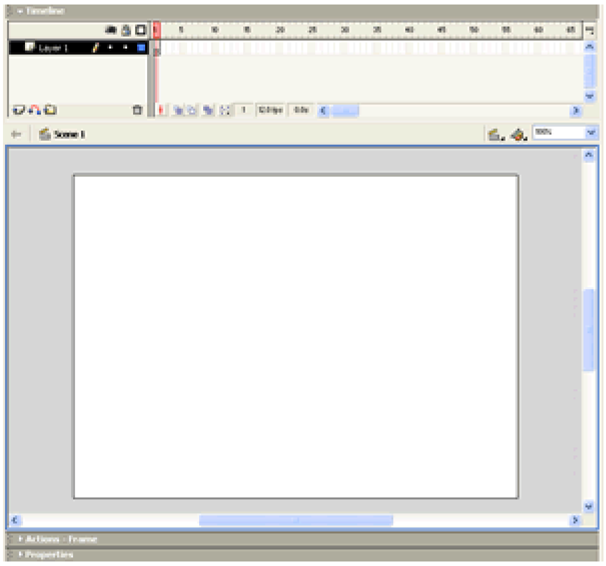
and

```
earth.canada.rex
```

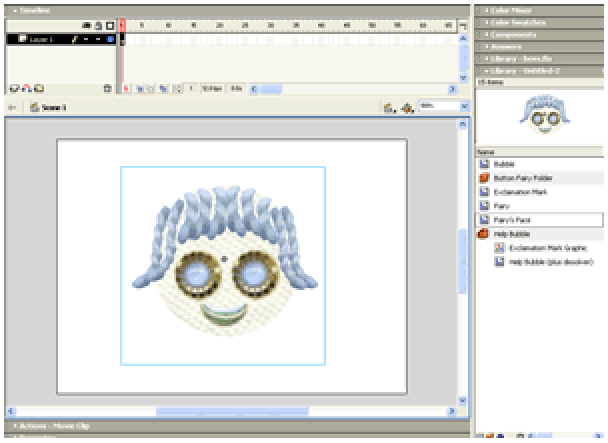
We'd know exactly which Rex's we were referring to. That's because **canada** and **india** <u>are also objects.</u> One Rex is inside an object called **india**, and another is inside an object called **canada**. If we used their full names, there would be no confusion.

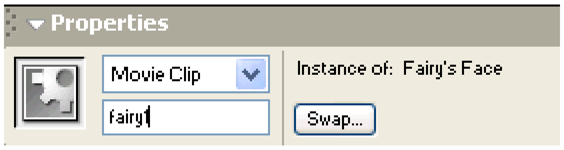
In Flash, if you want to refer to an object that's inside another object, you *have to* use its full name. Here's how it works:

Let's start with our main timeline called **`_root`**



Inside it, we create a new object called **`fairy1`**

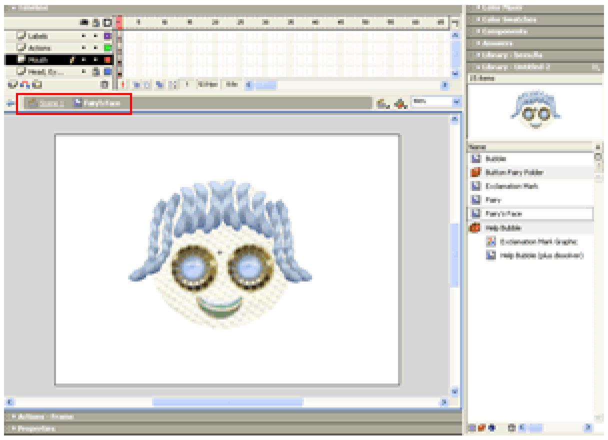




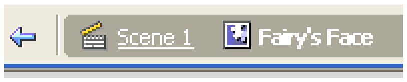
fairy1

Because she's on the main stage, **`fairy1`**'s proper dot notation name is now: **`_root.fairy1`**

Next, let's add something to our **`fairy1`** object. We'll double-click on the **`fairy1`** instance so that we enter the object's symbol editing window:

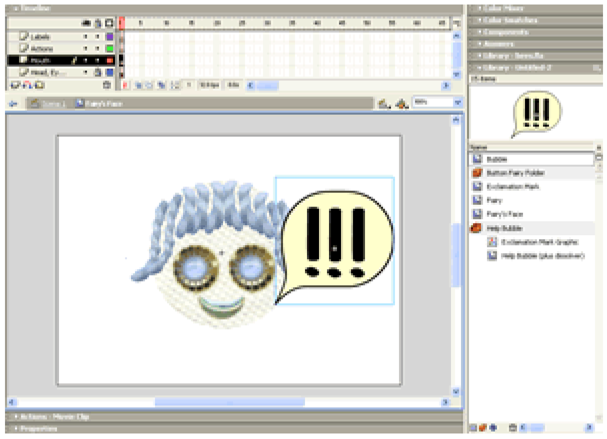


Look carefully at the top left corner of the stage:

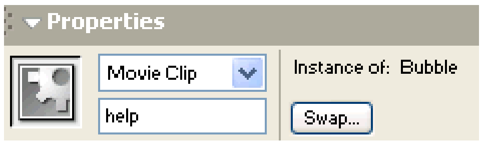


This shows us that we're no longer on the main timeline. We're inside the "Fairy's Face" symbol.

We're going to create an instance of a symbol in the library called "Bubble" by dragging a copy of it onto the stage.



Let's give our Bubble instance the instance name **`help`**.



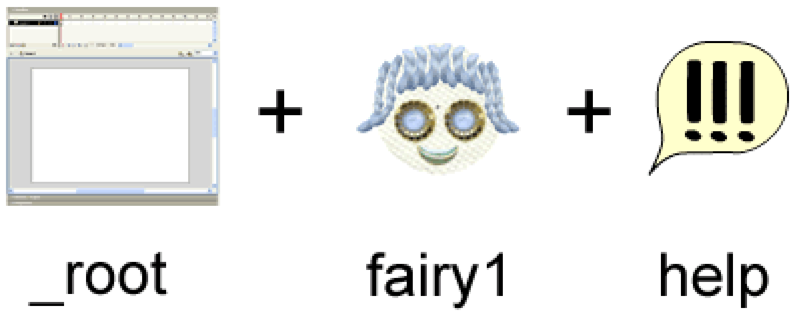
Doing this turns it into an object.

Now, just think about this for a moment. Where is **`help`**? Inside **`fairy1`**. Where is **`fairy1`**? On the **main timeline**: **`_root`**.

That means that if we want to refer to our new help bubble object, we need to use this name:

```
    _root.fairy1.help
```

Like this:



It's important to keep in mind that even though we added the **`help`** object inside the "Fairy's Head" symbol, we can only refer to it through the **`fairy1`** instance name. There is an advantage to this. If we created another instance of the

"Fairy's Head" symbol called **`fairy2`**, the **`help`** object would still be there for us to use; we wouldn't have to create it again. We'd just need to use the name **`_root.fairy2.help`**.

Understanding how dot notation works is a crucial first step to understanding how to program with ActionScript. Now that you know how to create and name objects, you can attach **actions**, **variables**, and change object **properties**. And that's when the real fun starts.

Introduction to ActionScript 2: Actions

In ActionScript, **actions** are special words that perform a pre-determined, built-in function. They "do" most of the work in your movies.

The first part of this lesson explains what actions are and how to attach them to objects. Read through this section carefully, but don't agonize about it too much if you don't understand right away or if you find it a bit confusing. The lesson is followed by a detailed exercise in which, by the end of it, using actions should be crystal clear.

How to Attach Actions to Objects

You can create actions very easily. Simply attach the action name to the end of the name of the object that you want to control. Let's look at our previous example. Do you remember this?

+
+
!!!
_root
fairy1
help

We had created an object called **`_root.fairy.help`**

Let's imagine that the first frame of the **`help`** Movie clip has a **stop action** on the first frame. You should remember how to create a stop action from our previous lessons on creating interactive movies. It looks like this:

```
stop();
```

If, however, you wanted the movie to start playing, you could assign a **play action** anywhere in your Flash animation. The play action looks like this:

```
play();
```

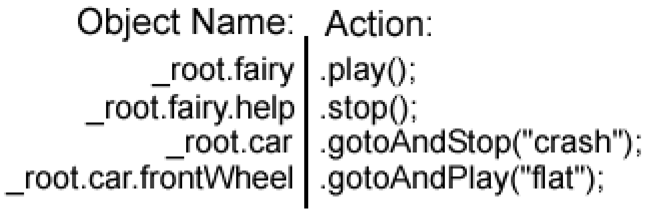
To use it, we need to *attach* it to an object. To start the **`help`** movie clip playing, we could use:

```
_root.fairy.help.play();
```

If we wanted the **`help`** movie clip to stop, we could use a stop action, like this:

```
_root.fairy.help.stop();
```

Action names are attached directly to the end of an object's name with a "dot", such as this diagram illustrates:



Object Name:	Action:
_root.fairy	.play();
_root.fairy.help	.stop();
_root.car	.gotoAndStop("crash");
_root.car.frontWheel	.gotoAndPlay("flat");

Another thing you should be aware of is that actions must always end with a semicolon:

```
;
```

This tells Flash that, after this line, the action must be performed. Most other programming languages use semicolons in the same way.

Understanding "Arguments"

Actions also require you to provide **arguments** in addition to the action name. Arguments are extra bits of information that the action needs to be able to complete the task properly. You can always recognize arguments, because they come directly after the name of the action, and are always enclosed within brackets, like this:

```
()
```

You might have noticed that the **`play();`** and **`stop();`** actions were followed by empty brackets before the semicolon. This is to show that they contain no arguments. If an action contains no arguments, you must use empty brackets to indicate this.

Let's consider our **`_root.fairy.help`** object again. If we wanted the **`help`** movie to jump to a specific labeled frame, we could use the **gotoAndStop** action or the **gotoAndPlay** action. These actions look like this:

```
    gotoAndPlay();
    gotoAndStop();
```

When you were creating interactive Flash movies in the previous lessons, you used these actions quite frequently. These two actions *must always* contain arguments – they can never be left with empty brackets.

Let's imagine that our **`_root.fairy.help`** movie has a labeled frame called "theEnd". If we wanted to create an action that caused the **`help`** movie to jump to that frame, we would write this action:

```
    _root.fairy.help.gotoAndPlay("theEnd");
```

As you can see, "theEnd" is the argument that the action uses. Without that argument, the action won't work. **gotoAndPlay** and **gotoAndStop** require that you use quotation marks inside the brackets to describe your argument. Not all actions require this – you only need quotation marks if your arguments are words or letters. If your argument is a number, such as a frame number, you don't need quotation marks.

There are lots and lots of actions that you can use in ActionScript, but these are the most basic and the most useful:

```
    play();
    stop();
    gotoAndPlay();
    gotoAndStop();
```

You should be familiar with how these work from the previous lessons on creating interactive movies. You will learn many more useful actions in the course of the lessons ahead.

By the way, to see the **`_root.fairy.help`** movie clip in action, visit this website:

```
    www.kaleidoscope-multimedia.com/trouble
```

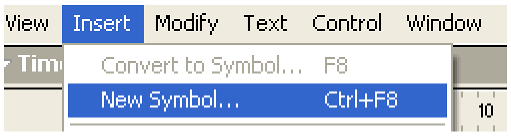
The "help" movie clip is told to play whenever the fairy is stung by a bee.

ActionScript Exercise 1: Using Objects and Actions

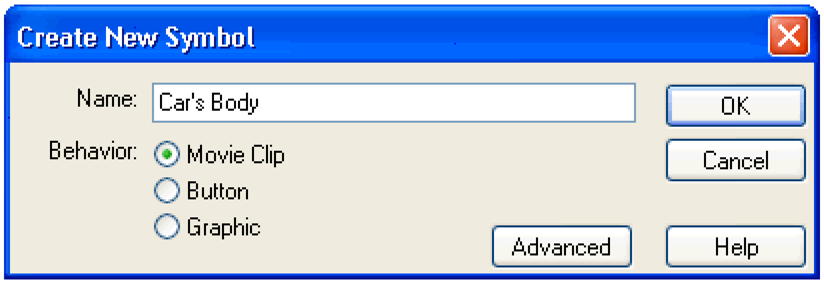
The following assignment will give you a chance to practice creating objects and assigning actions to buttons to control those objects. Make sure that you understand all of the concepts from the previous lessons before you start. Follow the directions VERY carefully.

A: Create Your Symbols

1. Create a new symbol. Click on the **Insert** menu near the top of the screen and choose **New Symbol….**

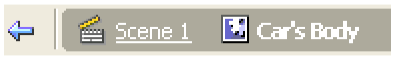


2. Give the new symbol the name "Car's Body." Make sure that **Movie Clip** is selected and click **OK**.



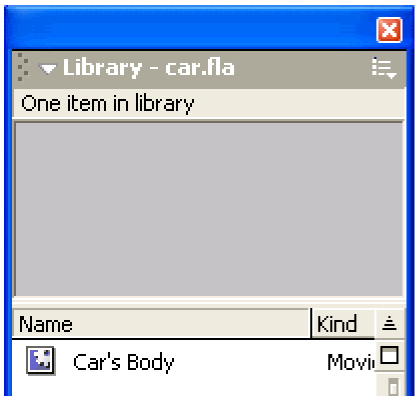
Name: Car's Body
Behavior: [✓ Movie Clip]

3. The editing window for the "Car's Body" movie clip should new open. You can confirm this by looking at the top left hand corner of the stage.

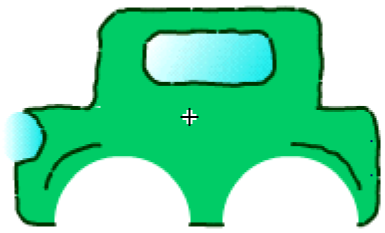


If you see something like the image above, you know you're in the right place. All the work we're going to do is now taking place *inside* the "Car's Body" movie clip symbol – not on the main timeline.

Open the library by pressing **F11** or clicking on the **Window** menu and choosing **Library.** You should see that the "Car's Body" Movie Clip Symbol has been added.

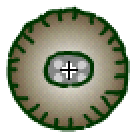


4. Use the drawing tools to create a picture of a car without wheels. Something like this:



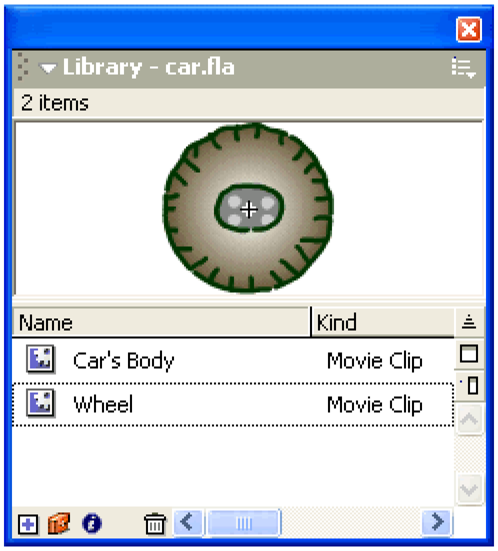
5. Create another movie clip symbol by following the same procedure you followed in steps 1 and 2. Call this symbol "Wheel".

6. In the "Wheel" symbol editing window, draw a wheel:



   Try drawing a slightly irregular shaped wheel, like this one, and add a bit of traction around the tire. The effect we're going to create will be much clearer if you do. Make sure that you draw the entire wheel on one layer.

Check your library to make sure that you now have 2 Movie Clip symbols in it.



Name	Kind
Car's Body	Movie Clip
Wheel	Movie Clip

If your library doesn't look something like this, you've made a mistake somewhere. Check the instructions above and see if you can figure out where you went wrong before you go any further. You must have 2 separate symbols, and the must be Movie Clip symbols.

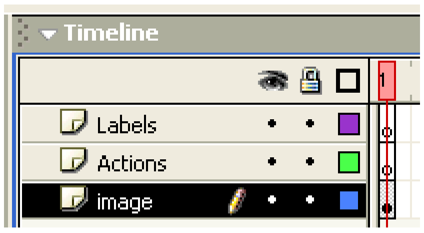
B: Set Up the "Wait" Column

There are actually a few different ways that you can create an animation that can be started and stopped with ActionScript. The method that we're going to use is not the simplest, but it's very clearly structured, which means that it's less likely that you'll make a mistake.

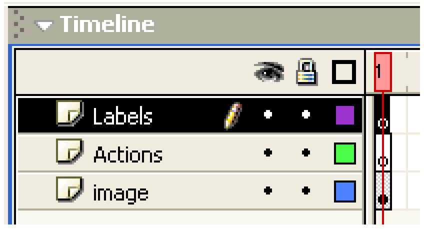
We need to turn the "Wheel" symbol into a mini interactive movie that can be controlled with ActionScript. Remember back to the interactive movie you created in earlier lessons; you'll need to apply all those skills. You'll be able to use this method again and again for all sorts of ActionScript controlled animations, including those in games.

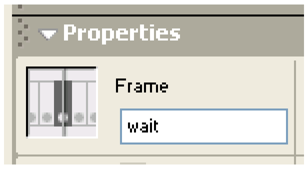
Make sure that you're still working inside the "Wheel" Movie Clip symbol, and follow these instructions

1. First, we need to correctly set up our timeline. We're going to create 2 separate columns. 1 Column will show the wheel as it is when it's stopped. The second column will be an animation loop of the wheel rotating. Create your **Labels** and **Actions** layers:

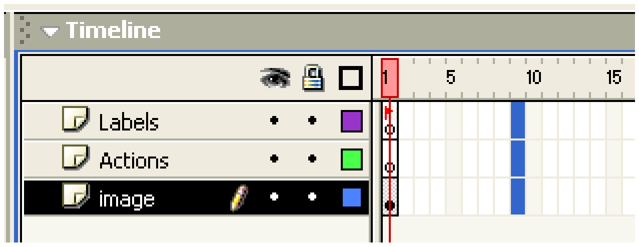


2. Select the first frame of the **Labels Layer**. In the **Properties Panel**, give the frame the label "wait".

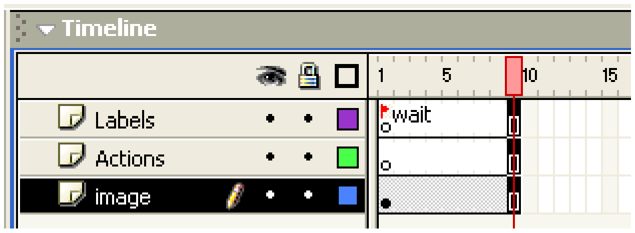




3. Before we go any further, we should define our column by adding 8 extra blank frames. Hold down the **crtl** key and, without letting it go, click on **frame 9** of the **Labels, Actions** and **image Layers.**

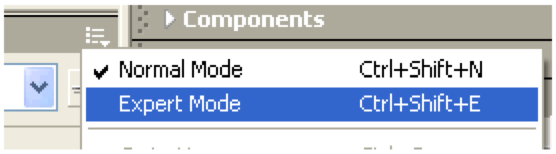


holding down **crtl** allows you to select more than one frame at a time. Next, press **F5** to add bank frames to all layers:



We've added these frames so that we can clearly see where or column is and read the "wait" label that we've created.

4. Click on the first frame of the **Actions Layer** so that it is selected. Open the **Actions Panel**. At the top right corner of the **Actions Panel** you should see what looks like a little picture of a list of items with a tiny arrow underneath. Click on this picture. Choose **Expert Mode** from the list of choices.

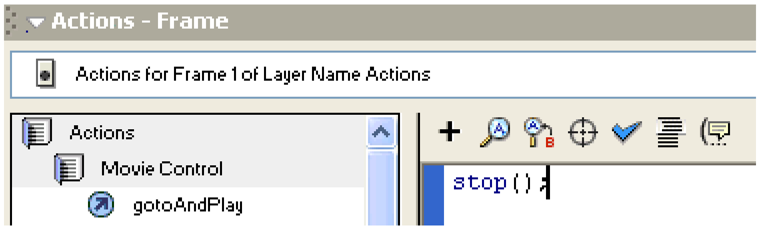


Expert Mode allows you type in ActionScript code directly into the **Actions Panel**. You need to use **Expert Mode** for any advanced ActionScript programming. From now on, you should *always* use it.

5. Type the following action into the **Action Panel**:

```
stop();
```

This tells your movie clip that it should stop at the first frame and not play the animation. The **Action Panel** should now look like this:



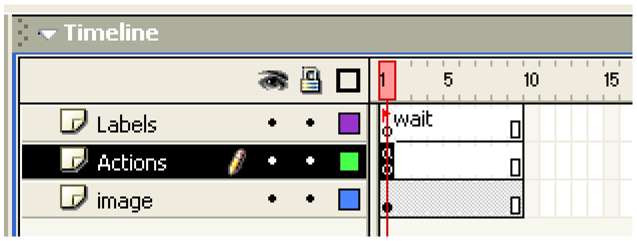
Actions for Frame 1 of Layer Name Actions

```
stop();
```

Double check to make sure that it says "*Actions for Frame 1 of Layer Name Actions*" at the top of the **Actions Panel.** If it doesn't, it means that you've entered the Action in the wrong place.

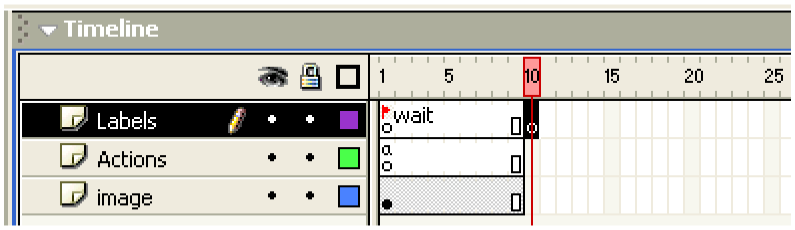
"stop" is displayed in blue letters because it is a word that Flash recognizes. Any **reserved words** in ActionScript show up in blue letters. This is useful, because if you type in an action that you think Flash should understand, and it doesn't show up as blue, you know that you've made a mistake entering it. **Reserved words** are words that Flash reserves exclusively for ActionScript.

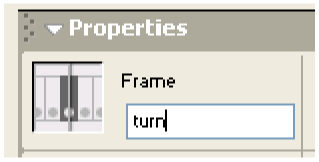
Your Wheel's timeline should now look like this:



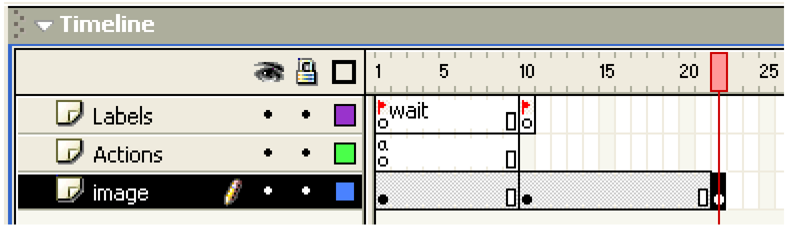
C: Set Up the Animation Loop

1. We're going to animate the wheel so that it rotates. Click on **Frame 10** of the **Labels** layer and insert a **keyframe.** In the **Properties Panel,** give the frame the label name "turn".

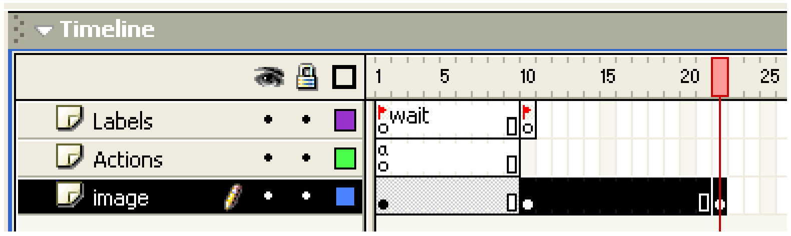




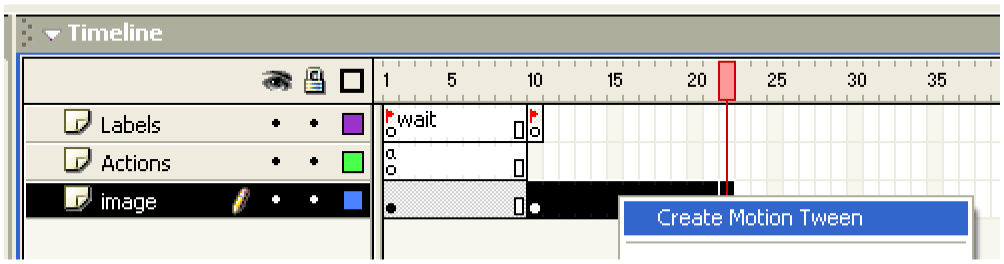
2. On the **Image Layer**, insert a keyframe on **frame 10** and **frame 22.**



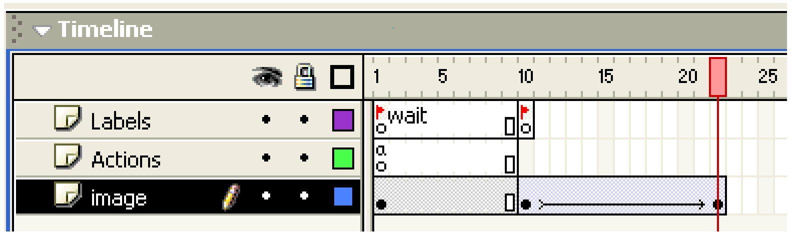
3. Highlight frames **10 to 22** on the **Image Layer**.



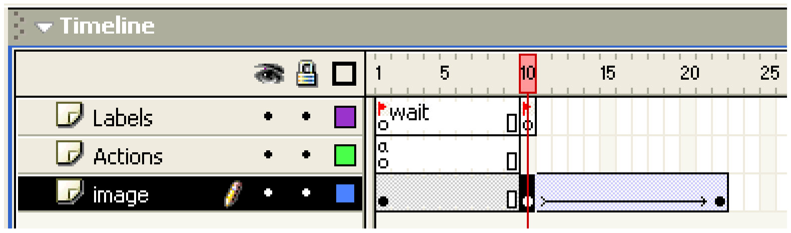
Right-click on the highlighted area and choose **Create Motion Tween** from the menu that opens.



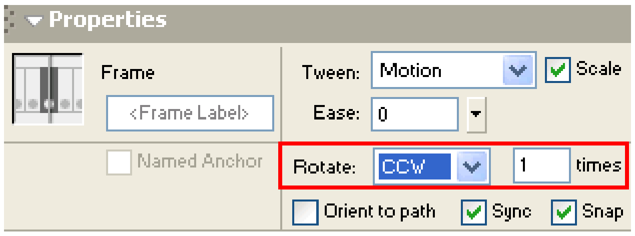
If you do this correctly, you should see a black arrow extending from frame 10 to frame 22.



4. Select **frame 10** on the **Image Layer**



and open the **Properties Panel** *(Flash 5 Users: Open the Frame Panel.)* You should see something that looks like the image below. Select "CCW" (which stands for Counter Clockwise) in the **Rotate** field. Make sure that it's set to rotate **1** time.

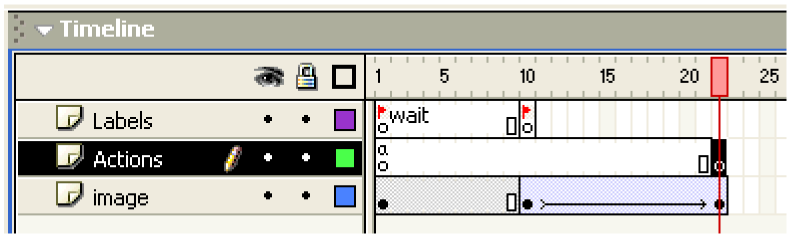


Properties
Frame
<Frame Label>
Named Anchor
Tween: Motion ✓ Scale
Ease: 0
Rotate: CCW 1 times
Orient to path ✓ Sync ✓ Snap

D: Create the Loop

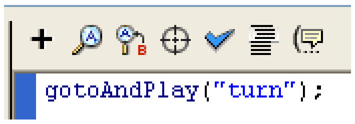
The next thing to do to is to set the animation up so that when it reaches frame 22 it loops back to frame 10 so that the wheel appears to be constantly turning.

1. Insert a **keyframe** on **frame 22** of the **Actions Layer**.



2. Open the **Actions Panel** and enter the following ActionScript code:

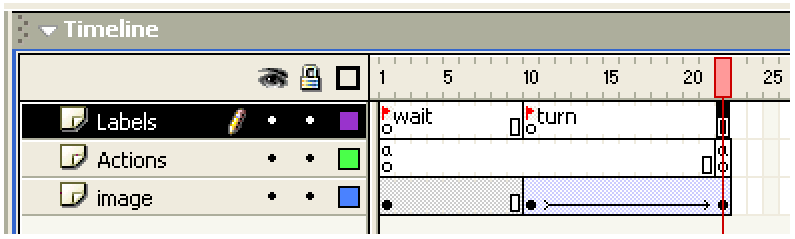
```
gotoAndPlay("turn");
```



```
gotoAndPlay("turn");
```

Every time the movie reaches this frame, it will jump to the **turn** frame, thus looping the animation. Because this action is on the same timeline that it is referring to, we don't need to include an object name.

3. Finally, we need to add a blank frame at **frame 22** of the **Labels Layer** so that we can read the label clearly.



Again, we don't really need to add this last step, but it makes it easier for you to see the columns and read the "turn" label.

We now have two clearly defined columns, each which performs a different function. The first column, "wait," is merely a static image of the wheel. The second column, "turn" is a looped animation of the wheel rotating.

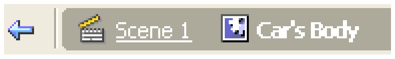
This is a very simple example, but you can create as many columns as you need for the different kinds of actions that you want your objects to perform. Whatever you put into these columns is entirely up to you, and you can have as many as you like.

For example, if you were creating a car race game, you might want a column called "flat" which displays an image of a punctured wheel when your car drives over an opponent's thumb-tack trap.

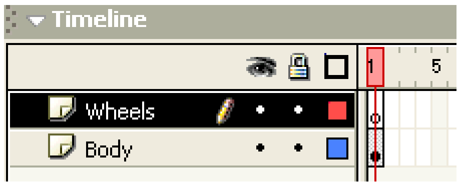
We're now ready to add wheels to our car's body.

E: Build the Wheel and Car Objects

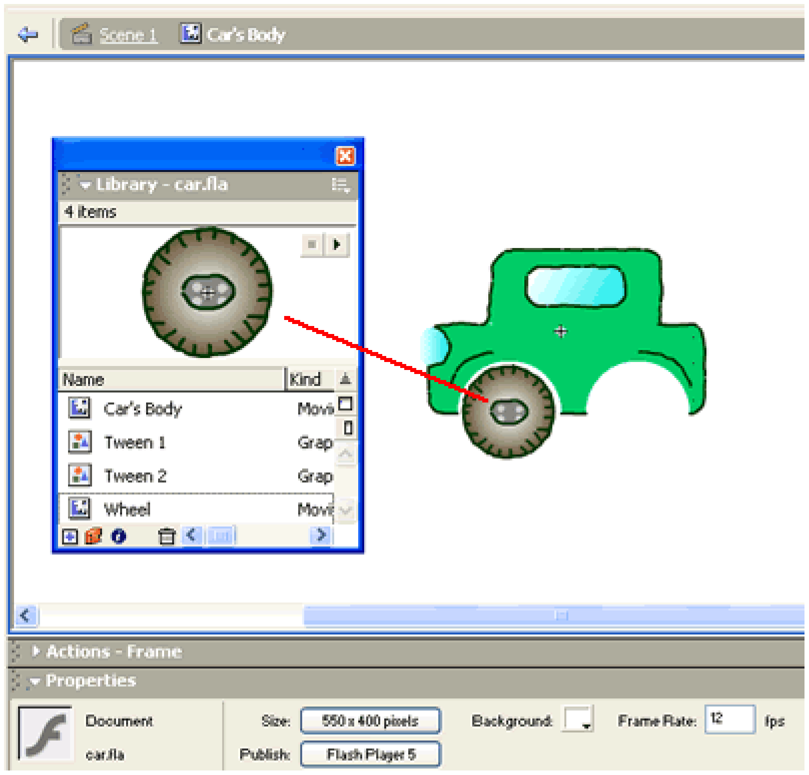
1. Double click on the "Car's Body" symbol in the library to enter its symbol editing window. Confirm that you are in the correct place:



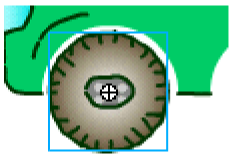
2. Create a new layer called **Wheels**.



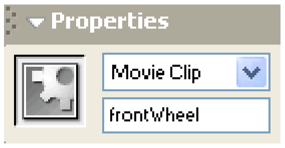
3. Make sure that the **Wheels** Layer is still selected and drag an instance of the of the "Wheel" symbol onto the image of the car's body. Position it so that it fits neatly in the front wheel space you created for it.



4. Make sure that the wheel is selected…

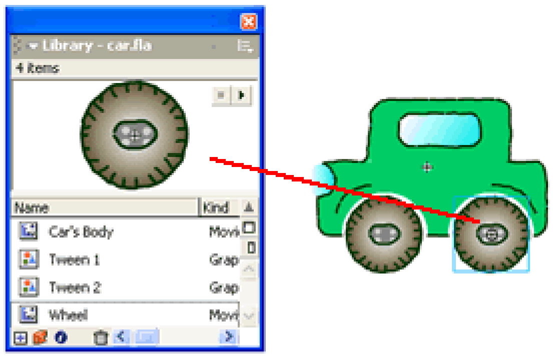


… open the **Properties Panel**, and give it the instance name **`frontWheel`**.

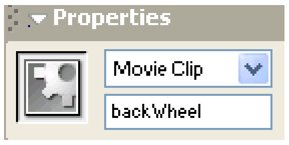


frontWheel

5. Create another instance of the “Wheel” symbol in the other empty spot.



and give it the instance name **`backWheel`**.

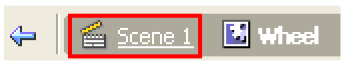


backWheel

You’ve now created 2 objects, **`backWheel`** and **`frontWheel`** inside the “Car’s Body” Movie Clip symbol. Before we can control them with ActionScript, we need to turn the “Car’s Body” itself into an object.

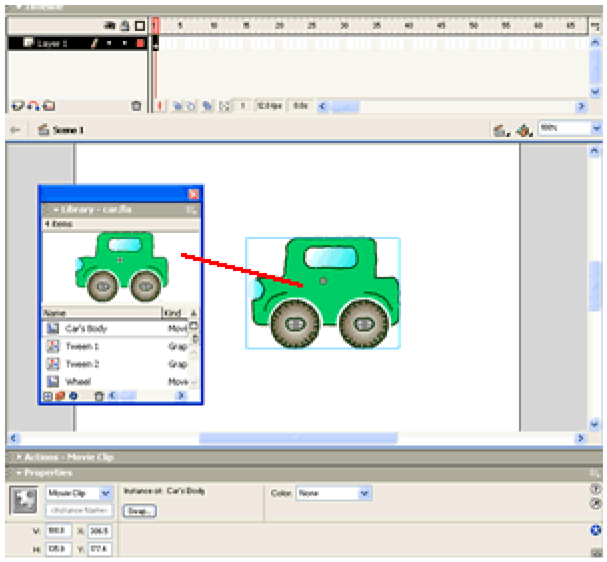
D: Create the Car Object

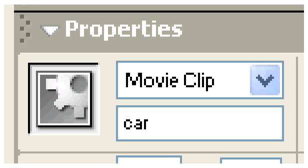
1. Click on “Scene 1” to return to the main timeline.



2. Rename **Layer1** to **Car**.

3. Drag an instance of your "Car's Body" Movie Clip symbol onto the main stage. Make sure that it's selected, and, in the **Properties Panel** give it the instance name **`car`.**





You now have three objects that you can control:

```
_root.car
_root.car.frontWheel
_root.car.backWheel
```

…Oh, you actually have four objects. You can control the main timeline directly by using the following, all by itself:

```
_root
```

It's very important to remember that you have all of these objects at your disposal. You can control any of the wheels by using ActionScript like might look like this:

```
_root.car.frontWheel.gotoAndPlay("turn");
```

This would tell the **`frontWheel`** object to start playing from the labeled frame called "start" that we created in the "Wheel" Movie Clip's timeline. Because the **`frontWheel`** object was created from the "Wheel" Movie Clip symbol, it **inherits** all of that symbol’s actions, frames and any objects that might have been inside it.

If you had included some animation in the “Car’s Body” symbol, you could control it by using ActionScript that might look like this:

```
_root.car.play();
```

If you made the car animate from the right side of the screen to the left, the wheels would move along with it because they’re inside the car object.

You could also control any labeled frames that you might have on the **main timeline** by using **`_root`** followed by an action. For example, you could tell the main timeline to start playing with the following bit of ActionScript:

```
_root.play();
```

Or, if you had a series of labeled frames, you could jump to them with a **gotoAndStop** or **gotoAndPlay** action, such as:

```
_root.gotoAndPlay(“page2”);
```

You might wondering, where do we put these actions? There are few places you can put them:

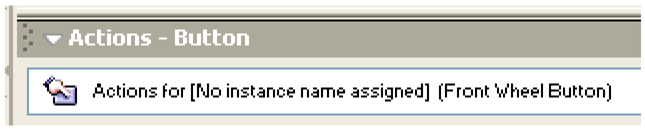
- On Buttons
- On Frames
- Directly on Objects

Over the course of these lessons, we’re going to be looking at all these methods. Often the most convenient place to put them is on buttons, because that means the user can directly control them.

E: Program Your Buttons

1. Create a simple button or use one from the Common Libraries. Create a new layer called **Buttons** and drag a copy of the button onto the main stage.

2. Select the button, and open the **Actions Panel**. Make sure that you are in **Expert Mode.** Double check to make sure that the button is selected and that you are entering the ActionScript in the correct place:

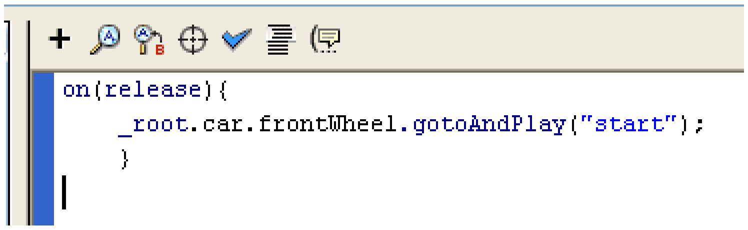


Actions for [No instance name assigned] (Front Wheel Button)

3. Type the following into the **Actions Panel**:

```
on (release) {
     _root.car.frontwheel.gotoAndPlay("start");
}
```

Your **Actions Panel** should look like this:



```
on(release){
   _root.car.frontWheel.gotoAndPlay("start");
   }
```

Test your movie and click the button. If you've done everything correctly, the front wheel should turn.

on (release) - A New Action

on (release) is a special action used by buttons. Essentially, it tells Flash, "When the mouse button is released, you should do whatever is between the curly brackets."

Curly brackets enclose the actions:

```
{ }
```

Curly Brackets are used to keep all the actions together that should be performed when the mouse button is released. Anything ActionScript code outside of the curly brackets is not performed. Curly brackets are used in exactly the same way in Java and C++. In Pascal, *begin* and *end* are used instead.

In plain English, the ActionScript code that we wrote above translates as this:

```
When the mouse button is released, do this: {
     Tell the front wheel to play;
}
```

Once you understand the concepts, it’s not difficult at all. Soon you’ll start to realize how easy, efficient and fun it is to program Flash with ActionScript.

Exercise Extension

Now that you understand the basics, add the following extras to your movie:

1) Create a button that tells the back wheel to play

2) Create a button that tells both wheels to play

3) In the Car Movie Clip’s timeline (NOT the main timeline,) animate the car so that it moves from one side of the screen to the other. Assign a stop action to the first frame to prevent it from playing. Then, create a button, on the main timeline, that makes the car move.

Introduction To ActionScript
4) Properties

Once you have created an object, you have a great deal of control as to what you can do with it. With objects, you can:

- *Attach **Actions***
- *Modify **Properties***
- *Create and alter **Variables***

*In the previous lesson, we looked at how you can attach **actions** to objects that can be controlled with buttons. In this lesson we're going to explain how to modify object **properties** and, in later lessons, how to attach and alter **variables**.*

*Once you know how to use **actions**, **properties** and **variables**, you will be able to control every aspect of you objects in a completely interactive way..*

What are Properties?

A **Property** is a characteristic of an object. A property can define how an object looks, where it is, or its size.

Properties are what you use to describe certain aspects of your object. If you think of yourself as an object, how would you describe yourself? You might list the following items:

- 150 kg
- 170 cm
- green eyes

These are your **properties**. Unfortunately, if you don't like any of these properties, you're stuck with them. You can't change or alter them in any way.

In Flash, however, you can change the all the properties of your objects. Properties can be changed whenever you need them to be changed: either with the click of a button or whenever something else important happens in your game or interactive movie.

In Flash, you can change the following properties of your objects:

- an object's transparency (called it's **alpha**)
- an object's **height**
- an object's **width**
- the **rotation** of an object

- an object’s **visibility**
- an object’s **position** on the screen (it’s **x** and **y** position)
- the **size** or **scale** of an object
- an object’s **name**

There are a few more object properties, but those above are the ones that you will use most frequently, particularly for games.

There are also some special properties that provide information on the position of the mouse on the screen, but we’ll look at those later.

How To Use Properties

The most important thing to know about properties is this: *Properties contain information (values) that describe how your objects should look or behave*.

In ActionScript, you can spot a property because its name is preceded by an *underscore* symbol (_). Here are some examples of properties:

```
_rotation
_height
_width
_x
```

To use a property, you *attach the property name at the end of an object name*. For example, if you wanted to refer to an object’s height, you might write:

```
_root.car._height
```

This is exactly the same way that you attach an action to an object.

In the above example, “`_height`” stores information about how high the object is, as measured in pixels.

Properties always “know” what their original value is. A **value** is a number that the property uses to change the way the object looks. If you created an object that was 50 pixels high, the object’s “`_height`” property would store that information. (*Pixels are the little dots that make up an image; if you look at your computer monitor closely, you should be able to see them*.)

We can easily change the **value** of any property. For example, if we wanted to change the height of an object to an exact size of 20 pixels, we could write:

```
_root.car._height = 20;
```

This would change the height of the object so that it's exactly 20 pixels. The equal ("=") sign is used to give the property its new value, and the semicolon (";") is used to activate the change.

What is important to remember is that once the value of the property has been set, it will always remain at that value until we change it again.

We can also assign one object's property value to another object's property. For example, we could write:

```
    _root.flower._height = _root.car._height;
```

This would change the "`_height`" of a flower object so that it is the same as the height of the car object.

This might seem a little confusing at first. Let's look at it this way: Suppose we initially set the car's "`_height`" property to "`50`" and the flower's "`_height`" property to "`20`". We could do that with 2 lines, like this:

```
    _root.car._height = 50;
    _root.car._height = 20;
```

… or just draw our objects on the screen with those heights.

Next, let's say we were creating a game with a monster flower. Whenever the car in our game bumps into the flower, the flower increases its size to that of the car. The easiest way to make the flower do this, is to tell the flower to become the same height as the car, and that we can do with the line of ActionScript code we looked at earlier:

```
    _root.flower._height = _root.car._height;
```

This essentially tells Flash: "*Make the flower as high as the car.*" Or, "*Assign the value of the car's _height property to the flower's _height property.*" And, the advantage here is that we don't even need to know how high the car is. Because the height of the car (50 pixels) is stored in its "`_height`" property, Flash already knows that it should be 50 pixels.

We can also assign values from different properties *in the same object*. This means that we can tell an object to be exactly as high as it is wide. Such as:

```
    _root.car._height = _root.car._width;
```

This would change the "`_height`" of the car so that it is the same as its "`_width`" - resulting in a perfect square shape. Again, we don't need to know the original values of these properties; Flash and ActionScript know what they are.

Another option is to add a mathematical calculation to the property value, such as:

```
    _root.car._height = _root.car._height / 2;
```

This would divide the current "`_height`" of the object by 2. The object would then be half as high as it was originally.

Properties give you almost limitless scope to change your objects in an interactive way. Once you start experimenting with them, you'll be amazed at how useful and fun they are to use.

You can use any of the following properties:

Property Name	**What it Does**	**Unit of Measure**
`_alpha`	Alters the transparency of an object. A value of 100 means that the object is opaque, 50 that it is translucent, 0 that it is completely transparent.	Percentage. Any number between 0 or 100
`_height`	The height of the object	Pixels
`_width`	The width of the object	Pixels
`_rotation`	Controls the angle of rotation of the object. A positive value results in a clockwise rotation, a negative value results in a counterclockwise rotation.	Any positive or negative number between 1 and 360
`_visible`	Controls object visibility. Accepts *Boolean* values: either `true` or `false`. eg. `root.car._visible = false;`	Boolean. `true` or `false`
`_x`	An object's horizontal pixel position on the stage.	Pixels
`_y`	An object's vertical pixel position on the stage.	Pixels
`_xscale`	The same as "_height" only measured as a *percentage* of the object's height.	Percentage
`_yscale`	The same as "_width" only measured as a *percentage* of the object's width.	Percentage
`_xmouse`	Tells you the current "x" position of the mouse. You can only *read* this value, not change it. For example: `_root.car._x  = _root._xmouse;` Sets the car's "_x" property to horizontal position of the mouse.	Pixels
`_ymouse`	Tells you the current "y" position of the	Pixels

	mouse. You can only read this value, not change it.	
`_name`	Tells you the name of the object, and lets you alter it if you need to. This property is rarely used, but as has some application in certain aspects of game design, so is worth remembering.	Any valid object name, such as “`car`” or “`flower`”

These are the most important object properties that you need to know. By modifying these properties, you will have an enormous amount of control over how your objects can be controlled on the screen, as we will see in the next lesson.

Introduction To ActionScript
5) Using Properties

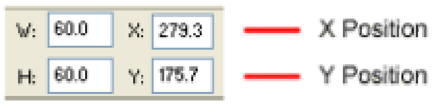
The following lesson explains how to use properties, and specifically provides an in-depth example of how to use the _x and _y properties. If you understand how the _x and _y properties work (and these are the most complex properties,) you can apply exactly the same skills to the other object properties mentioned in the previous lesson.

The _X and _Y Properties

Two particularly useful properties, especially for creating games, are the "`_x`" and "`_y`" properties. These properties control where the object is on the stage.

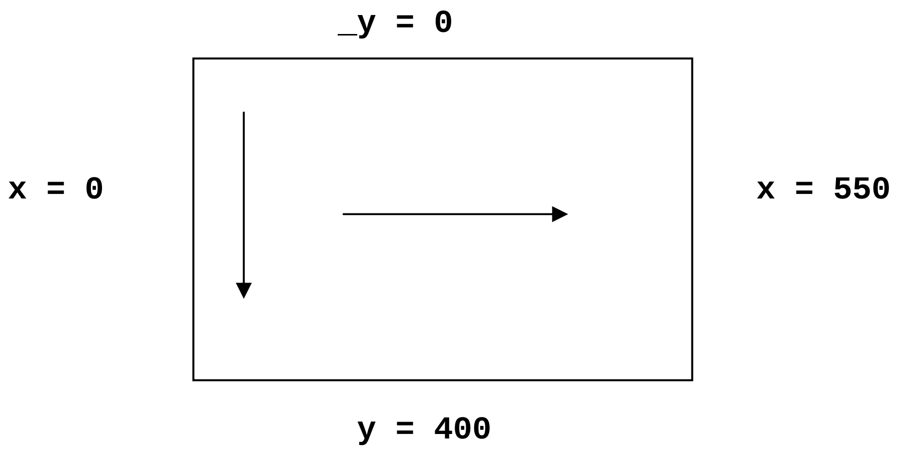
"`_x`" refers to the object's **horizontal** position and "`_y`" refers to its **vertical** position. The values that "`_x`" and "`_y`" store are **pixel values**.

Remember that pixels are the little dots that make up graphics on the screen. Each of these dots has a number, known as its **value**. Even if you don't know that, Flash does! However, you can easily see the _x and _y position of all your objects on the stage by selecting an object and opening the Properties Panel:



Using ActionScript and the "`_x`" and "`_y`" properties, we can tell our objects to go to a specific point on the screen. This is the basis for creating interactive movement and animation.

When you create a new movie in Flash, you are automatically given a stage size of 550 by 400 pixels. 550 is the width ("`_x`") and 400 is the height ("`_y`"). You can imagine this as a grid of squares, with the numbering starting at the top left hand corner of the stage.



At the top of the screen, `_y` is equal to zero. At the bottom, `_y` is equal to 400. `_x` is zero on the left side of the screen and 550 on the right. If you wanted to place an object in the exact center of the stage, you could write:

```
_root.car._x = 275;
_root.car._y = 200;
```

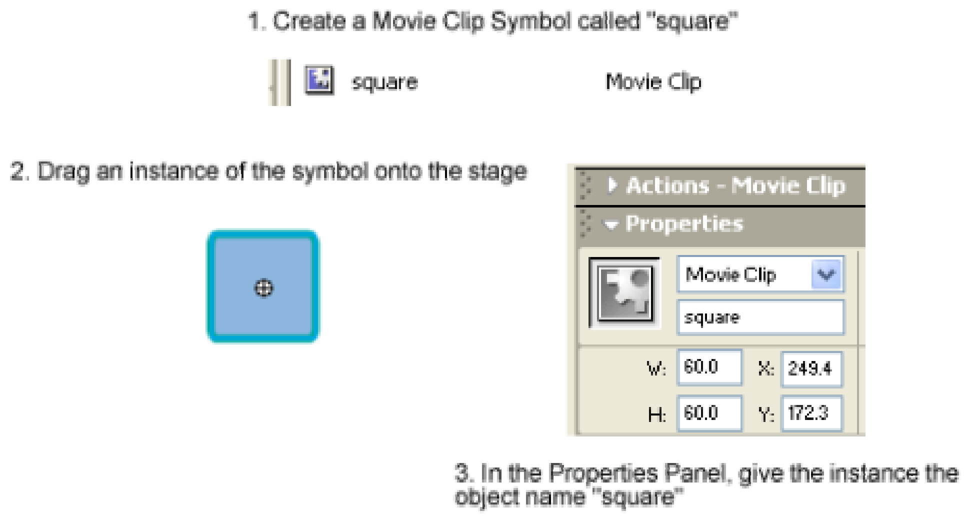
This is confusing at first, but with a little practice, you'll start to learn the logic behind it.

Using _X and _Y Properties

There are many ways to change an object's properties. You can change the property when a movie reaches a certain frame, when the user clicks on a button, or when an object bumps into another object. In the following example, you will learn how to change an object's `_x` and `_y` properties by clicking on a button.

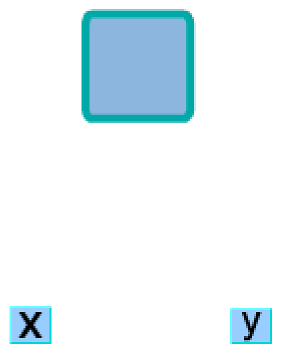
1. Create a new Flash movie.

2. Draw a square on the stage and convert it into a Movie Clip Symbol. Give it the object name "`square`".


1. Create a Movie Clip Symbol called "square"
square
Movie Clip
2. Drag an instance of the symbol onto the stage
Actions - Movie Clip
Properties
Movie Clip
square
W: 60.0
X: 249.4
H: 60.0
Y: 172.3
3. In the Properties Panel, give the instance the object name "square"

Position the square as close to the center of the stage as you can.

3. Create 2 button symbols. One should be labeled with the name "X" and the other with the name "Y". Drag instances of these buttons from the library and position them near the bottom of the stage. The stage should look something like this when you are done:



4. Click once on the "X" button to select it. Open the Actions Panel, and type in the following ActionScript code:

```
on(release){
     _root.square._x = 30;
}
```

This action is telling Flash: *When the mouse button is released, move the square's X position to 30.*

5. Click once on the "Y" button to select it. Open the Actions Panel, and type in the following actions:

```
on(release){
      _root.square._y = 50;
}
```

This action is telling Flash: *When the mouse button is released, move the square's Y position to 50.*

Now, test your movie and watch what happens when you click the buttons. If you followed the above instructions correctly, the square will be positioned at X position 30 and Y position 50 on the stage (near the top left corner.)

Next, try changing the numbers 30 and 50 to other numbers and observe how that affects the position of the square.

Changing Property Values Incrementally

Often you will want to reposition an object only slightly when a button is clicked, and have the object continue to move further along on every click. This is called **incremental movement**.

With incremental movement, you don't know exactly what the new pixel position of the object is going to be. You only know that, whenever the user clicks a button, that you want the object to move forward along the X or Y axis by, say, 5 pixels at a time.

Follow the instructions below to modify your Flash Movie so that the buttons you created in the previous steps move the square *incrementally* on each click

1. Click once on the X button to select it and open the Actions Panel. Change the ActionScript code so that it looks like this:

```
on(release){
   _root.square._x = _root.square._x + 5;
}
```

2. Next, select the Y button and change its ActionScript code so that it looks like this:

```
on(release){
   _root.square._y = _root.square._y + 5;
}
```

Test your movie, and see what happens. You should notice that every time you click one of the buttons, the position of the square is advanced 5 pixels along either the X or Y axis.

How It Works

Let's look at the X Button actions:

```
on(release){
   _root.square._x = _root.square._x + 5;
}
```

In plain English, we are telling Flash:

"When the mouse button is released, the circle's X position should be the same as its current X position, plus 5 pixels"

The most important line is this one, and deserves a slightly closer look:

```
_root.square._x = _root.square._x + 5;
```

What this line is doing is telling Flash to add 5 to the ***current*** value of "`_root.circle_x`". And the current value of "`_root.circle._x`" is ***already stored*** in "`_root.circle._x`"

A lot of people new to computer programming find this confusing. Well, it is!

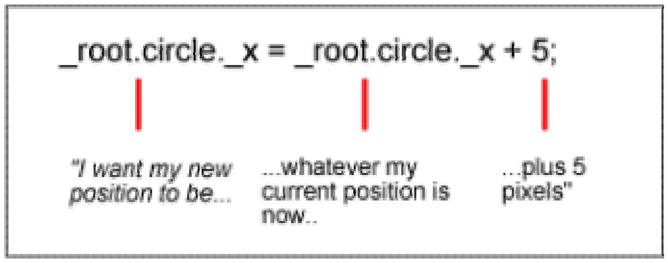
If we had just written:

```
_root.circle._x = 5;
```

… the circle would have jumped to the left hand side of the screen, just 5 pixels to the left of its edge, and stayed there, no matter how many times we clicked the button.

What we had to do was tell the circle to *add* 5 pixels to where it already is. That means that the first time we click the button, the circle moves 5 pixels to the right. The second time we click the button, it is already in a new position, so the 5 pixels are added to whatever that new position is.

Ok, did you have to view that last sentence though a mirror and stand on your head to make sense of it? Here's another way of looking at it:


_root.circle._x = _root.circle._x + 5;
"I want my new position to be...
...whatever my current position is now..
...plus 5 pixels"

If you're still having problems, cut the above diagram out, tape it to your computer monitor, and follow it blindly whenever you need to move an object. Don't worry, with a little practice, you'll get it!

This technique is called **incrementation**. If we were subtracting a value, we would call it **decrementation**. (We could easily decrement the above value by replacing the plus sign with a minus sign – try it in your movie!)

Modifying Other Properties

You can apply these same techniques to any property that accepts numbers as a value. This includes `_rotation`, `_height`, `_width`, `_xscale`, `_yscale` and `_alpha` (transparency.) Changing any of these properties is just as easy as changing the `_x` and `_y` properties.

For example, if you wanted to change the `_rotation` property of your square, you could change one of your button actions so that it looked like this:

```
on(release){
   _root.square._rotation = _root.square._rotation + 5;
}
```

It's exactly the same technique we used earlier; only the property name has been altered.

Remember, if you use a minus sign, you will be able to alter your object properties in an opposite way. For example, if you wanted your square to gradually become transparent, you could use this ActionScript code:

```
on(release){
   _root.square._alpha = _root.square._alpha - 5;
}
```

And, remember that you can don't need to always use the number "5"! Larger numbers, such as 10, will cause the effect to be more obvious, because the change from the old value to the new value will be more drastic. Lower numbers, such as 2, will alter the property more gradually.

Changing values incrementally is at the heart of creating truly interactive animations. With a little practice, you'll find it easy.

Hint:

(*If you're feeling a little over saturated with new information, skip this next section. But any brave souls who think they can tarry further, read on….)*

Because incrementation is such a common procedure, there is a special operator you can use:

+=

This operator tells the property to add the new value to itself. So, you could write the above example that we were looking at as:

```
_root.circle._x += 5;
```

… and it will work in exactly the same way! It's exactly the same as writing

```
_root.square._x = _root.square._x + 5;
```

… but involves less typing.

To decrement a value, use: **-=**.

ActionScript Assignment 1: Controlling Properties

The following assignment practices the techniques you have learnt in ActionScript so far: controlling objects, assigning button actions, changing property values and using incremental and decremental values. Follow the instructions carefully:

1. Create a movie clip symbol, drag it onto the stage and give it a name.
2. Copy 18 instances of a button symbol onto the stage. Make sure that they're small, and organize them into 2 neat columns on the left or right hand side of the screen.
3. Program the buttons so that they do the following:

   a) Increase and decrease the object's _x position.
   b) Increase and decrease the object's _y position.
   c) Increase and decrease the object's _height.
   d) Increase and decrease the object's _width.
   e) Increase and decrease the object's _rotation.
   f) Increase and decrease the object's _xscale.
   g) Increase and decrease the object's _yscale.
   h) Set the object's visibility on.
   i) Set the object's visibility off.

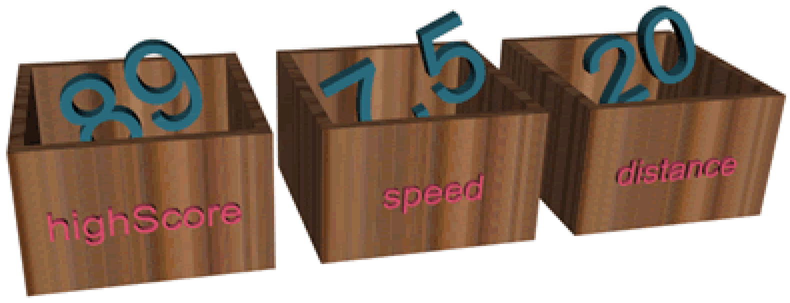
4. Label all your buttons so that you know what they do and make sure they're organized in a neat and logical way on the screen.

Evaluation:
4 Marks: For properly programming the buttons, as described above.
1 Mark: For clearly labeling your buttons
1 Mark: For a neat organization of the all the elements on the stage.

TOTAL: /6

Introduction To ActionScript
6) Variables

You can think of **variables** as boxes that store information:



If you need to use the information inside the box, you just need to refer to the box's name. You can also empty the box and put new information in at any time.

Variables act like an object's memory. They contain information that an object needs to perform whatever task you have set it.

Unlike **actions** and **properties**, variables are *not predefined*. Its up to *you* to decide what kind of variables you will need, what kind of information they will store, and then create them.

You create variables by attaching words, any words you like, to the end of an object name. The words can represent numbers, letters or **Boolean** (true/false) values. The following are all examples of variables:

```
_root.car.speed
_root.car.distance
_root.highScore
```

None of these variables have been created by Flash. They were decided on by the person who was creating the movie. And, it's up to you to decide what kind of information they store.

Initializing Variables

To create a variable, you need to do the following:

a) Give your variable a **descriptive name**.
b) **initialize** the variable.

Initializing a variable means giving the variable its first **value**. You can always change that value later, but a variable must always have an *initial* value.

Let's imagine that you're creating a game, and you want a variable that will keep track of the score. First, you would decide on a name for your variable. "`score`" would seem to be a good name. Then, you would have to decide on an initial setting for the score. When you start a game, what is the score? Well, in most games it's usually zero. So, we can initialize our variable by writing the following:

```
    _root.score = 0;
```

The most common place to initialize a variable is on the first frame of the **actions** layer of the main timeline. And, actually, if you initialized it there, or anywhere on the main timeline, you would simply have had to write:

```
    score = 0;
```

The variable already *knows* that it's on the main timeline, so you don't really need to add "`_root`". The main timeline *is* the `_root`.

If we think of our variable as a box, the above line would have put the number "`0`" into a box called "`score`":

Assigning New Values to Variables

After you have assigned an initial value, you can change a variable's value at a later time. In fact, that's why it's called a "variable" because its value can "vary."

Let's imagine that we have a very simple race game. It's so simple, in fact, that the score can be either 0 or 1. The player would get a score of 1 if he or she beat the computer to the end of the racetrack. Let's imagine that after a long, difficult race, the player reaches end of the track, and wins by a hair's breadth.

At the end of the race we would update the score to show that the player has won. We would then simply add:

```
_root.score = 1;
```

This would now fill our score box with the number 1:

Easy, isn't it?

You can also, *at any time*, change the value of "`_root.score`" if you need to by simply giving it a new value. The following are some examples:

```
_root.score = 23;
_root.score = _root.car._height;
_root.score = (_root.playerOneScore - _root.playerTwoScore);
_root.score = "You Lost!";
```

Whatever information you tell the variable to store, it will remember. You can then use that information whenever you need it, simply by referring to the variable's name. And, like property values, a variable value can be the result of a mathematical equation.

Now let's imagine that we've got a slightly more complex game, where the player has to shoot down invading aliens. The player starts with a score of 0, but every time he or she hits an alien, 1 is added to the total score. In other words, the score is **incremented**. You've heard that word before somewhere…remember? In this case, we would write the following whenever an alien spaceship is hit:

```
_root.score = _root.score + 1;
```

This means that 1 is added to our old score, 0, resulting in the new value: 1. The next time an alien is hit, 1 would be added to "`_root.score`" again, resulting a new value: 2. On the third hit, 1 would be added to "`_root.score`" again, so that it would become 3.

The logic behind this is the same as incrementing property values.

Local and Global Variables

Variables can either be **local** or **global.**

Global variables can be used anywhere in the Flash movie, at any time. They are always preceded by a "`_root`" prefix, without an object name. The following are examples of global variables:

```
_root.score = 30;
_root.speedLimit = 40;
_root.maximumBullets = 10;
```

They're not attached to any objects.

Local variables can only be used *inside the object they were created in.* They only exist for as long as that object exists.

Let's imagine that we created 3 variables *inside* our "car" Movie Clip. We only want to use these variables to control the car's animation, and no other objects.

```
_root.car.speed = 15;
_root.car.distanceCovered = 30;
_root.car.tankIsEmpty = false;
```

Because we don't need to use these variables anywhere else, we've made them local variables. They're attached directly to the car object.

Local and Global variables can share information. Let's say that we've got a local variable, that stores the car's speed, and a global variable, that keeps track of the speed limit for all the cars in the game:

```
_root.car.speed = 15;
_root.speedLimit = 25;
```

If we wanted to set the car's speed to the global speed limit, we could write:

```
_root.car.speed = _root.speedLimit;
```

Now, "`_root.car.speed`" would equal 40, exactly the same as "`_root.speedLimit`". "`_root.speedLimit`" hasn't changed, but it has shared its value with "`_root.car.speedLimit`".

It also works the other way around. For example, we might want to determine the score of our game by subtracting the car's speed from the speed limit.

```
_root.score = _root.car.speed - _root.speedLimit
```

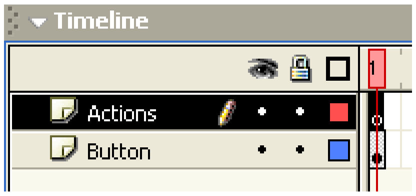
Which are better to use, local variables or global variables? In most cases, I would recommend that you use global variables. Although local variables can be very useful, especially in a complex project, you are far less likely to make mistakes if you stick with global variables - while you are learning, at least.

Using "Dynamic Text" to Display Variable Values

Often you will want to display the value of a variable on the screen. The following exercise will show you how to use **Dynamic Text** to do this. It will also give you a very concrete example of how variables actually work.

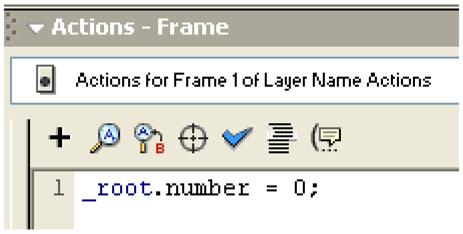
The first thing we need to do is create a global variable called "`number`", and initialize it.

1. Create a new layer and name it "Actions".



2. Make sure that Frame 1 of the Actions Layer is selected and open the Actions Panel. Enter the following into the Actions Panel:

```
_root.number = 0;
```

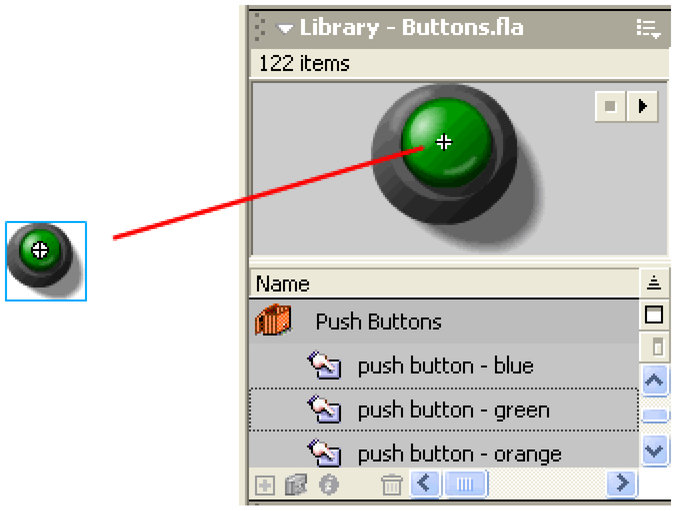


```
_root.number = 0;
```

This is how we **initialize** the variable. We're giving it its first, *initial*, value. This is almost always done in Frame 1, because Frame 1 is the first frame that Flash loads when the movie starts.

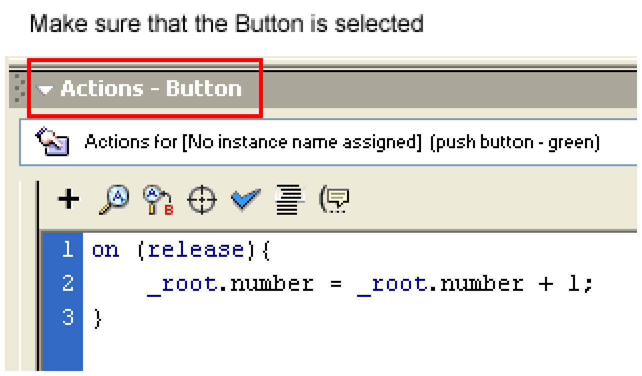
Next we need to add a button.

3. Open the **Buttons** library from the **Common Libraries**. Choose any button that you like and drag it onto the main stage.



4. Make sure that the button is selected, and open the Actions Panel. Enter the following button object actions:

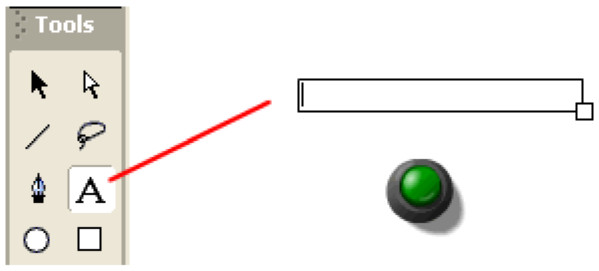
```
on (release){
              _root.number = _root.number + 1;
   }
```


Make sure that the Button is selected

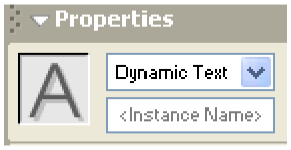
Actions for [No instance name assigned] (push button - green)

```
on (release){
    _root.number = _root.number + 1;
}
```

5. Choose the **Text** tool and use it to create an empty text field on the main stage.



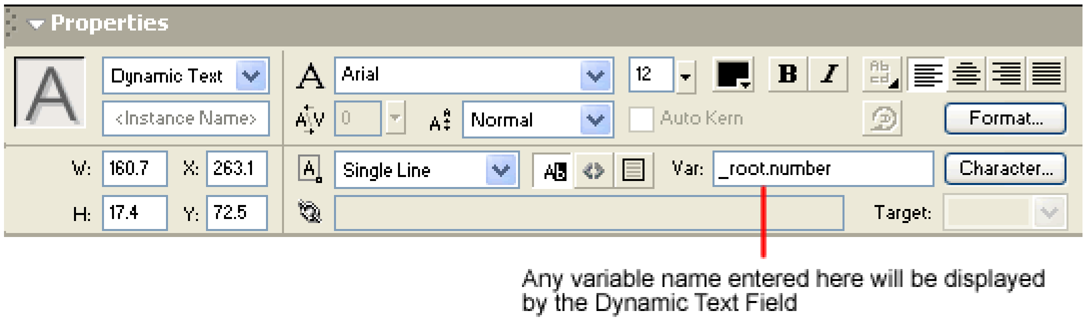
6. Make sure the Text Field is still selected, and open the **Properties Panel**. Choose **Dynamic Text** from the drop-down menu on the far left-hand side.



Dynamic Text is text that can be changed according to the value of a variable.

7. With the **Properties** panel still open, find the field called "Var:" and enter the following:

```
_root.number
```



This is the same variable we initialized earlier. Assigning a variable here will cause the value of that variable to be displayed in the Dynamic Text Field.

Test you movie… what happens???

Exercise Extension:

1. Create another button to *subtract* 1 from `_root.number` each time it is clicked. What happens when `_root.number` goes below zero?

2. Initialize `_root.number` to "2." (Remember, you initialized `_root.number` on the first frame of the Actions layer on the main timeline.) Create another button that *multiplies* `_root.number` by 2 each time it is pressed. How high can you go? (*Remember, use an asterisk, *, to perform multiplication.*)

3. Create another button, and, in the `on(release)` action, enter the following:

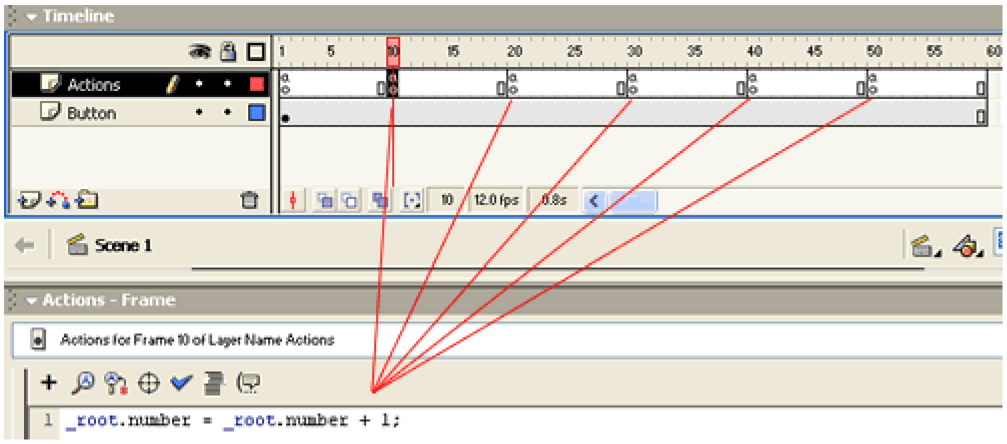
```
_root.number = _root._xmouse;
```

Click on the button and see what happens. What information is it giving you?

Reposition the button on the stage, and then test your movie again. Is there a change in the value of `_root.number`? If so, why?

4. On the Actions Layer of the main timeline, add 10 keyframes spaced 15 frames apart. In each new keyframe, enter the following:

```
_root.number = _root.number + 1;
```



```
_root.number = _root.number + 1;
```

Add this action to each keyframe, spaced 10 frames apart

Test your movie. What happens?
This is an easy way to make a time-lapse counter in Flash. Usually, it would be created as a self-contained Movie Clip animation, and dragged onto the stage wherever you need it.

Introduction To ActionScript
7) Variable Types and Strings

"Type" refers to the kind of information that a variable stores. *Words* are a different **type** of information than *numbers*. When you are using variables, you must be aware of what **type** of information they contain, as this can be very important to how your variables are used.

In ActionScript, as in other computer programming languages, there are *4 major **types***:

1. **Integer** (whole number):
   ```
   _root.age = 12;
   ```

2. **Real** (decimal number):
   ```
   _root.total = 2.4455623;
   ```

3. **String** (letters and words, surrounded by quotes):
   ```
   _root.message = "You Won!"
   ```

4. **Boolean** (true or false):
   ```
   _root.car.empty = false;
   ```

Unlike many other programming languages, however, ActionScript "*knows*" what **type** of information variables store. You don't need to worry about declaring integer, real or string types, as you would if you were programming with Java or C++. Also, ActionScript is usually not picky about the difference between **integers** and **real numbers**. If it sees a number with a decimal place, it assumes it is a **real number**. Likewise, numbers without decimal places are assumed to be **integers**. **Integers** and **real numbers** can be combined freely.

There are a few rules that you need to remember when creating new variables from combinations of other variables:

- You can combine any number variables (integer and real numbers.)
- You can combine any string (word) variables.
- You *<u>can't combine</u>* Boolean (true/false) variables with anything, even other Boolean variables.
- You *<u>can't combine</u>* string (word) and number variables if the result is a number.
- You can combine string (word) and number variables *if the result is a new string of words*.

Combining Variable Types: Examples

The following examples demonstrate how variable **types** can (and can't) be combined

1. Let's imagine that we have **initialized** the following variable values. To **initialize** a variable means setting it to the value we want it to start out at:

   ```
   _root.newTotal = 0;
   _root.age = 12;
   _root.total = 20;
   _root.message = "You Lost!"
   ```

   The following combination would be acceptable:

   ```
   _root.newTotal = _root.age + _root.total;
   ```

   Both `_root.age`, `_root.total` and `_root.newTotal` are all number variables, so they can be combined without any problem.

   The following combination would *not* be acceptable:

   ```
   _root.newTotal = _root.age + _root.message;
   ```

   Can you see why? `_root.age` and `_root.newTotal` are number variables and `_root.message` is a **string** (word) variable. These different types can't be combined in this way.

   How does `_root.newTotal` know that it is a number variable? By the last type of information that was stored in it, which was "0".

2. You can, however, combine any **string** variables to form a *new* string variable. For example, let's say we've initialized these variables:

   ```
   _root.name = "James";
   _root.greeting = "Hello, my name is ";
   _root.message = "";
   _root.age = 10;
   ```

   (*Notice that `_root.message` was initialized to "". When you're dealing with words, this is equivalent to setting a number variable to 0. It means that the variable is blank.*)

   We could then add this line of ActionScript to combine the string variables together:

```
     _root.message = _root.greeting + _root.name;
```

This would result in:

```
     Hello, my name is James
```

Because all the variables are strings, it works.

In computer programming, when strings are combined in this way it is called **concatenation**.

3. You can combine strings and number variables *if the final result is a string*. For example:

```
_root.message = _root.greeting + _root.name +
     ". I am " + _root.age + " years old.";
```

(The above would be all on one line.) This would result in:

```
     Hello, my name is James.  I am 10 years old.
```

Combining strings in this way can produce very interesting results.

Notice that spaces also need to be added as part of your strings, if you want your words to be spaced evenly. ActionScript doesn't know where you want to include spaces and punctuation, so you have to make sure that you carefully consider where you want them to appear and add them at the correct places yourself.

Introduction To ActionScript
8) Data Entry Exercise

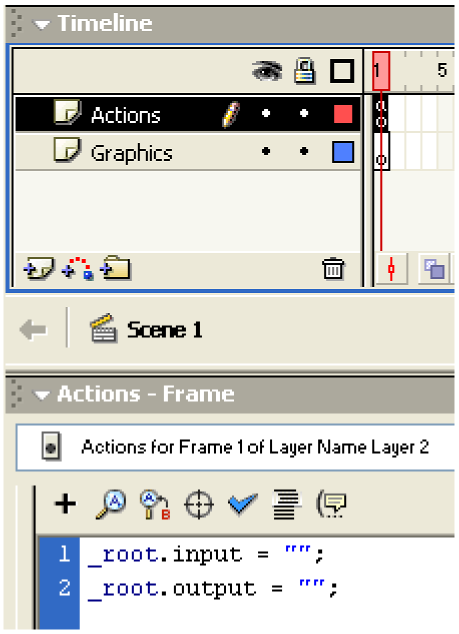
In the following exercise you will learn how to enter words and numbers into Flash with the "Input Text" feature. You will then learn how to output that text on the screen, and combine it with additional text. You will also learn how to use these techniques to perform simple mathematical calculations.

A) Initialize Your Variables

The first thing you need to do is **initialize** your variables, so that Flash knows what values they should be set to first.

Start a new movie. Create an Actions Layer and, on the first frame of that layer, enter the following actions:

```
_root.input = "";
_root.output = "";
```



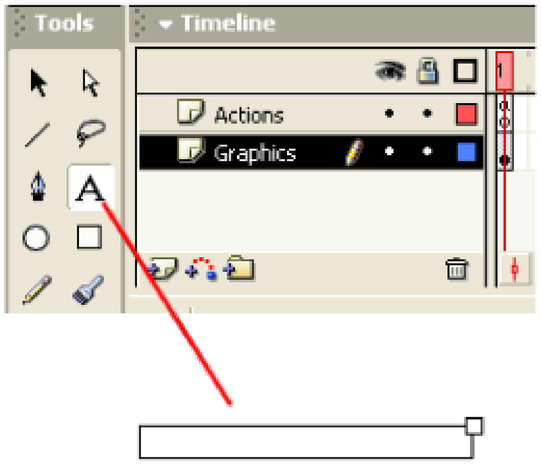
```
_root.input = "";
_root.output = "";
```

Variables are almost always initialized on Frame One of the Actions Layer.

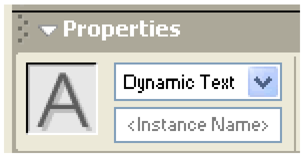
B) Set Up an Output Field

Next, you need to create **a Dynamic Text Field** to display the variables on the screen. (By the way, a **text field** is a box in which you can enter or display words.)

1. Click on the Text Tool and draw a blank text field on the stage. Make sure that you *don't* draw it on the Actions Layer.



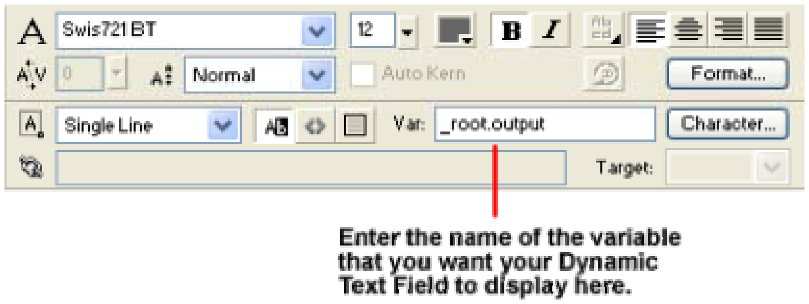
2. Make sure that the new Text Field is still selected. Open the **Properties** panel and choose **Dynamic Text**.



Dynamic Text allows you to display the values of variables.

3. In the **Var:** field, enter the following:

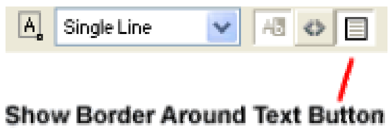
```
_root.output
```



Enter the name of the variable that you want your Dynamic Text Field to display here.

This tells Flash that this **Dynamic Text** field should display whatever the value of `_root.output` is. It will display the value of any variable you enter here.

To make sure that your text field is clearly visible when you test your movie, click the **Show Border Around Text** button.



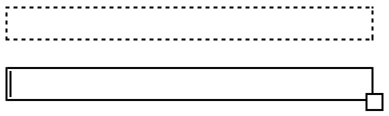
Once you've tested the functionality of your text field, you may want to make the border invisible again and draw your own custom border, using Flash's drawing tools, on another layer beneath the one that your text field is on.

Optionally, you may at another time want to change the **Single Line** option to **Multiline**, if you need to display a lot of information.

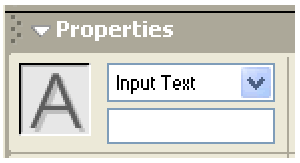
C) Set Up an Input Field

You now need to set up a text field to *enter* words or numbers.

1. Click on the Text Tool and draw a blank text field onto the stage, just below the previous one you created.



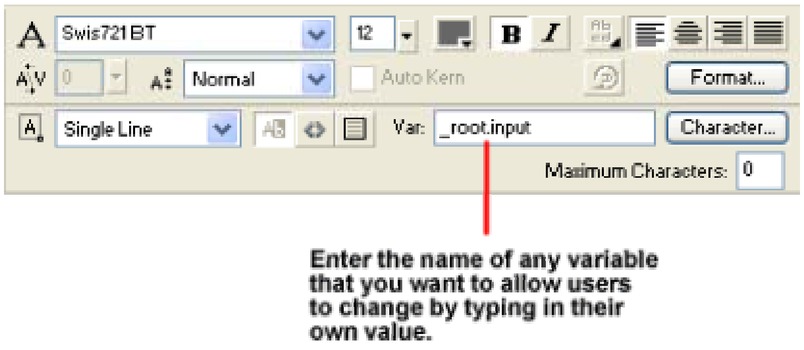
2. Open the **Properties** panel and choose **Input Text.**



Input Text allows you to enter information (numbers or words) into the text field.

3. In the **Var:** field, enter the following:

```
_root.input
```

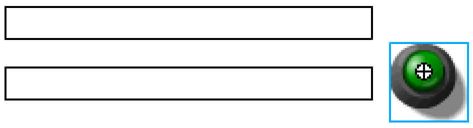

Swis721 BT
12
Normal
Auto Kern
Format...
Single Line
Var: _root.input
Character...
Maximum Characters: 0
Enter the name of any variable that you want to allow users to change by typing in their own value.

This tells Flash that whatever is entered into this field becomes the value of `_root.input`. You could, of course, use choose any name for your variable.

D) Create a Button To Activate the Input Text Entry

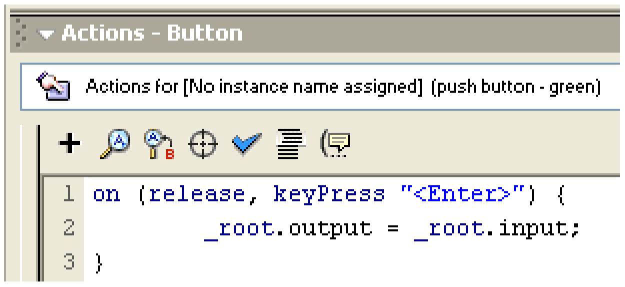
The next step is to add a button to activate the user's text entry.

1. Drag one of the buttons from the Common Library onto the stage. Position it next to the Input Text field.



2. Add the following to the button's **Object Actions**:

```
on (release, keyPress "<Enter>") {
     _root.output = _root.input;
}
```


Actions - Button
Actions for [No instance name assigned] (push button - green)

```
on (release, keyPress "<Enter>") {
        _root.output = _root.input;
}
```

The additional "`keyPress "<Enter>"`" argument for the "`on`" action tells flash to accept the change if the user presses the button OR the "Enter" key on the keyboard.

Test your movie. Enter some text in the Input Text field, and click the button. You should see the same text you entered displayed in the Dynamic Text field.

E) Clearing the Input Text Field

You might have noticed that after you type in some words and press Enter, that the Input Text field does not become blank. If you want to enter new text, you need to erase what you have written first. To clear the Text Input field automatically, so that the user can enter some more text, add the following to your Button Object Actions:

```
_root.input = “”;
```

Your button object actions should now look like this:

```
on (release, keyPress "<Enter>") {
     _root.output = _root.input;
     _root.input = "";
}
```

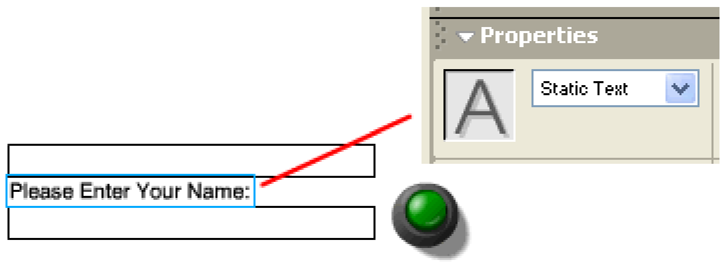
Test your movie and observe the effect. Can you see why this works?

D) Performing Simple Text Manipulation

Ok, that’s all fine and well, but what can you do with the kind of simple input/output system we set up above? In the following example you will see how you can use this technique to create a very basic simulation of artificial intelligence.

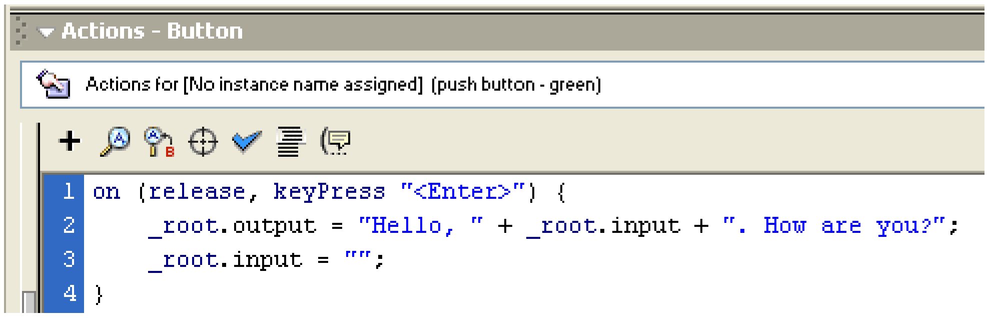
1. Create a *Static Text* Field just above the Input Text Field. A Static Text Field is one that does not contain any variables, and cannot be changed or updated. This is the kind of text that you have been used to using in Flash in the past. Type in the following:

   ```
   Please Enter Your Name:
   ```



2. Now, change the Button Object Actions, so that it looks like this:

```
on (release, keyPress "<Enter>") {
    _root.output = "Hello, " + _root.input + ". How are you?";
    _root.input = "";
}
```


Actions - Button
Actions for [No instance name assigned] (push button - green)

```
on (release, keyPress "<Enter>") {
    _root.output = "Hello, " + _root.input + ". How are you?";
    _root.input = "";
}
```

The only change is on the second line. What we've done, using simple plus symbols, is combine "`Hello, `" and "`. How are you?`" with the `_root.input` variable. In computer programming, combining strings together like this is called **concatenation**.

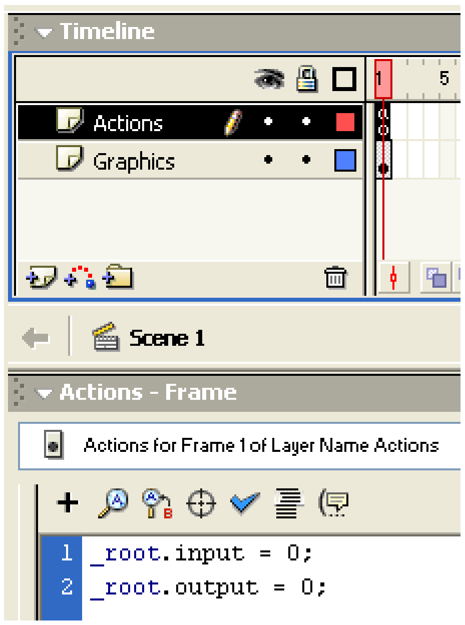
Test your movie, and see what happens now.

E) Performing Simple Mathematical Calculations

You can use these same techniques to do mathematical calculations.

1. In the first frame of the Actions Layer, change your variable values so that they're initialized to 0.

```
_root.input = 0;
_root.output = 0;
```



```
_root.input = 0;
_root.output = 0;
```

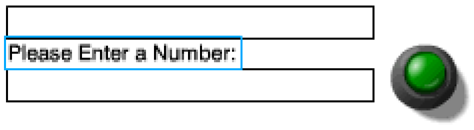
This tells Flash that we're now dealing with numbers rather than strings.

2. Next, change the Button Object Actions so that they look like this:

```
on (release, keyPress "<Enter>") {
     _root.output = _root.input * 10;
     _root.input = 0;
}
```

This tells Flash that `_root.output` should be `_root.input` multiplied by 10. In ActionScript, as in all other programming languages, the asterisk ("*") symbol is used to represent multiplication.

3. Change the Static Text so that it reads, "`Please Enter a Number:`".



Test the movie and observe the result. The Dynamic Text field will display any number you entered, multiplied by 10.

Experiment with a few other mathematical operators and see what happens. The operators you could use are:

/ Division
- Subtraction
+ Addition

* Multiplicaion

You might notice an unexpected result when you use the addition operator. Why do you think this might be occurring?

Performing Addition with Input Text Field Entries

When you enter text into an Input Text field, ActionScript automatically assumes that what you've entered is a string. If you enter a number, it converts the number into a string. This is a real problem if you want to add a number from an Input Text field to another variable.

To see this in action, change your Button Object Actions so that they look like this:

```
on (release, keyPress "<Enter>") {
          _root.output = _root.input + 10;
          _root.input = 0;
}
```

Test the movie, and try entering "`23`" into the Input Text field. After you press the button, you should see this:

```
2310
```

Of course, 23 plus ten is not 2310! What is happening is that ActionScript is interpreting the number you entered in the Input Text field as a **string**, and **concatenates** (combines) it with the number 10 instead of adding the two numbers together.

To perform the addition, you need to *force* ActionScript to interpret the entry from the Input Text field as a number. To do this, you need to use a special built in function called **Number**.

Change your ActionScript code so that it looks like this (the code you need to change is highlighted in bold text):

```
on (release, keyPress "<Enter>") {
          _root.output = Number(_root.input) + 10;
          _root.input = 0;
}
```

We've done two things:

- Surrounded `_root.input` in brackets

- Added a special ActionScript word called **Number** in front of the brackets.

Number forces everything inside the brackets to be interpreted as a number by ActionScript. Very importantly, **Number** is spelt with a capital "N." You'll know that you've entered it correctly if it appears in dark blue in your Actions Panel. Flash colour codes all **reserved words** (special words that ActionScript uses) as dark blue. This is useful to know, as it is an easy way of confirming whether you have entered any ActionScript reserved words correctly.

Test your movie now. The addition should work properly.

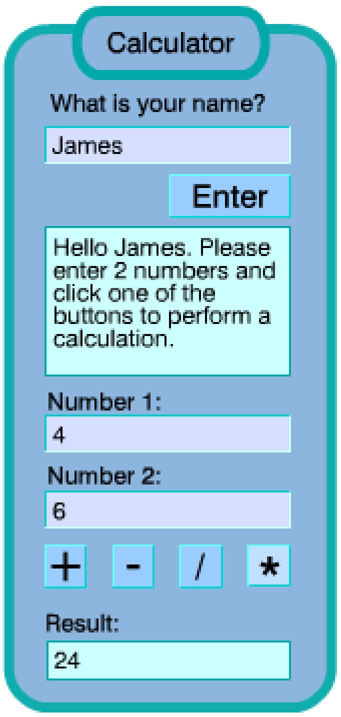
Keep this technique in mind whenever you need to add numbers obtained from Input Text fields. You do not need to use **Number** when adding together number variables from any other source.

Calculator Assignment

You should now have all the skills you need to build a very simple calculator in Flash. Your calculator should do the following:

- Ask the user to enter his or her name.
- Greet the user with a welcome message, after the user presses an “Enter” button.
- Ask the user to enter 2 numbers. You will need to create 2 Input Text fields to allow this.
- Provide the user with 4 choices as to how they might wish to calculate the numbers: Addition, Subtraction, Division or Multiplication. You need to create 4 buttons to allow for this.
- Display the result of the calculation.

When you are finished, your calculator should look something like this:



Hello James. Please enter 2 numbers and click one of the buttons to perform a calculation.

Can you think of a design to make your calculator look like a “real” calculator – something someone might be able to put in their pocket?

Hints:

- You will need at least 5 buttons

- You will need at least 3 Input Text fields
- You will need at least 2 Dynamic Text fields
- You will need to use the following variables (you can create your own names for these variables if you like):

  ```
  _root.name
  _root.greeting
  _root.number1
  _root.number2
  _root.result
  ```

Remember to ***initialize*** these variables on the first frame of your Actions layer.

Evaluation:

4 Marks: For properly providing 4 calculator operations
2 Marks: For correctly setting up and using Dynamic, Input and Static Text
1 Mark: For a properly formatted greeting
1 Mark: For properly initializing your variables
2 Marks: For an interesting well thought out layout/design for your calculator.

TOTAL: /10

Part 3:
Introduction to Game Design

Video Game Design: An introduction

You'll be pleased to know that the most difficult part of the journey is over. Now that you have a basic understanding of how ActionScript works and how you can use it to make highly interactive software, you can now extend that knowledge into the realm of video game design. Flash also happens to be the best tool currently available for making 2D games – by far. The next best software you could use, *Director*, just doesn't come close when it comes to ease of use and a fully integrated programming environment. There's just no easier and more versatile way to make games than with Flash.

The first thing that I should point out is that all the programming techniques you will be learning are very well defined and very specific to game design. Once you learn these basic techniques, you'll find that you use them over and over again. And, you'll find that once you've worked through these exercises, you can simply copy and paste most of the ActionScript code you've written, with minor modifications, to create your own completely unique games.

We'll be covering the following topics, step by step, in the chapters and sections ahead:

Part 3: Introduction to Game Design

- How to move a player object with the keyboard
- How to prevent your player from moving beyond the edges of the screen
- How to check to see whether your player is bumping in to other objects, such as enemies, and then take some kind of action, such as updating a score.
- How to create a maze environment for the player to move through

Part 4: Advanced Game Design

- How to create objects that move by themselves
- How to have your player fire bullets
- How to create a vehicle with a rotating gun turret
- How to create a "running and jumping" game

And, throughout the sections, we'll also look at four specific case studies that show you, step-by-step, how you can use these techniques together:

- Dungeon Maze Adventure
- Car Racing Game
- Space Shooter
- Running/Jumping Game (also known as a Platform Game)

- Drag-and-Drop Game

The final chapter of this book explains how to create a simple drag-and-drop matching game for young children. There are just a few new techniques to learn which you'll find when combined with the other techniques you already know, give you almost limitless scope to pursue any type of game you might think of.

It's important to point out, however, what our limitations are. Flash only allows us to create 2 Dimensional games – it just does not yet have the technical capabilities to handle 3D. This is not a bad thing, however, because it is exactly 3 times more difficult to make 3D games than it is to make 2D games with the technology currently available. And, as all the basic techniques for creating 3D games are the same for 2D, you're far better off getting a good grounding in 2D games first. When you do enter the fascinating realm of 3D, you'll find that you've already got all the groundwork covered and, with your background in Flash game design, you'll find it just that much easier.

The other area of game design that is not covered is word/number games (like hangman.) or text based adventure games (like the classic Zork or Kings Quest series.) The reason for that is that the programming concepts involved are quite heavy, as you need to able to create and search a database for information. The techniques you would be required to learn begin to fall into the realm of computer science, which is just beyond the scope of this book. For those of you interested in pursuing this area, *Foundation Action Script* by Sham Bhangal contains an excellent chapter on the subject of word games.

Finally, there is no mention made here of on-line multiplayer games. There are a few reasons for this. Firstly, the techniques involved are highly specialized and require knowledge of other software, technologies and programming languages outside of Flash and ActionScript. Secondly, the technology is changing so rapidly that it is likely to be out of date by the time you are reading this. So, it may amount to a lot of ultimately wasted effort in the long run, which is not something I want to burden you with while you are still learning the basics. For multi-user Flash environments the technology *seems* to be moving in the direction of the **Flash Communication Server** – but the verdict is still out. For those of you who feel you have a good grasp of the techniques covered in this book and would like to extend those skills into the multiplayer arena, the best book on the subject is currently *Flash Games Studio* published by Friends of Ed.

Based on these limitations, however, you'll find that what you will actually be able to produce by the time you have worked through these exercises and examples is quite extraordinary. There is no 2D video game from the 80 and 90 that you *can't* reproduce with Flash if you want to, and any possible game you can dream up you'll have the technical capabilities to achieve. Have fun, and send me your games when you're done!!!

Introduction to Game Design
1) Keyboard Control

Control Structures and the "If…Else" Statement

All computer programming languages need a way to figure out the order in which actions should happen, and what kinds of actions should happen based on certain conditions.

For example, if you were creating a computer game where the player controls a car, the computer might need to know that the car should only move forward ***if*** it has enough petrol or ***if*** it hasn't crashed into a wall. Normal cars know this by themselves, but computer generated cars have to be told exactly what to do.

Computer programming languages use things called **control structures** to figure out these sorts of problems. Control structures organize what happens when.

One of the most useful control structures is the **If Statement**. As in the example of the car game mentioned above, an "If" statement performs an action *if* a certain condition is true. If the condition is not true, the action doesn't happen.

The following is an example, in plain English, of an "If" statement in action. (Plain English examples of computer programming code are called **pseudocode**, ("fake code") by the way, and can be useful way of working out complex logic):

```
If the car's tank is full, move the car.
If the car's tank is empty, stop the car.
```

Not too difficult, is it? We can modify our pseudocode so it uses the correct ActionScript syntax (the addition of brackets, braces and semi-colons,) like this:

```
if (the car's tank is full){
     move the car;
}

if (the car's tank is empty){
     stop the car;
}
```

As you can see, the keyword here is **If**. Right after the "If" are a pair of brackets which enclose a **conditional statement**. If the conditional statement is true (such as "the car's tank is full") the action surrounded by the curly brackets {…} activates. If it is not true, the action is skipped and nothing happens.

Of course, "the car's tank is full" or "move the car" is English, not ActionScript programming code. We need to translate our English into ActionScript. To do this, our conditional statement needs to check some kind of values. These values are usually in the form of variables.

Let's imagine that we've created a **Boolean** (true/false) variable called "`_root.car.tankIsEmpty`" that keeps track of the car's petrol. And let's say we've created a labelled frame on the "car" movie clip called "Go" and "Stop". Our Action Script code might look like this:

```
if (_root.car.tankIsEmpty == false){
     _root.car.gotoAndPlay("Go");
}

if (_root.car.tankIsEmpty == true){
     _root.car.gotoAndStop("Stop");
}
```

The conditional statement uses a **double equal sign** (==) to check to see if the statement is true. A single equal sign won't work.

"If" statements are often (but not always) followed by an **Else Statement**. "Else" statements tell the computer what to do if the "If" statement above it was not true. We could therefore have written the above ActionScript code as:

```
if (_root.car.tankIsEmpty == false){
     _root.car.gotoAndPlay("Go");
}else{
     root.car.gotoAndStop("Stop");
}
```

This code does exactly the same thing as the first example but the code that comes after the "Else" statement will *always* be activated if the first "If" statement isn't. With a bit of practise, you will learn when you need to use just a series of "If" statements or whether you need to combine them with an "Else" statement.

Comparison Operators

A **comparison operator** is a symbol, like the **double equal sign** (==) that helps the computer decide when it should do what. Comparison operators are always used in **conditional statements**. Conditional statements are what are between the brackets of an "If" Statement, such as "(`_root.car.tankIsEmpty == false)`".

The following is a list of the most useful comparison operators you can use:

==	is equal to
!=	not equal to (you can also use **<>**)
<	less than
>	greater than
||	Or
&&	And
<=	less than or equal to
>=	greater than or equal to

By combining these simple operators, you can create remarkably complex "artificial intelligence."

A few examples of how conditional operators might be used appear below:

Greater Than and Less Than…

Let's imagine that our car's petrol tank can be not just full or empty, but, like a real car, can become gradually less full as it goes along. Our imaginary car is full when the fuel tank reads "10" and empty when it reads "0" Let's also imagine that our fuel is represented by a variable called "`_root.car.fuel`" and that we've initialized it to "10" at the very start of the game. As the game progresses, perhaps every few kilometres, the car's fuel could drop by 1.

We can use a **greater than** or **less than** comparison operator to check to see whether the fuel is more or less than zero.

```
if (_root.car.fuel < 1){
     _root.car.gotoAndPlay("Stop");
}
```

In this example, if the car's fuel drops below 1 (becomes zero) the car will stop.

We can also check to see if a value is **less than or equal to** a certain value, such as this example:

```
if (_root.car.fuel <= 3){
     _root.car.warningLight.gotoAndPlay("Blink");
}
```

In this example, if the car's fuel is equal to or less than 3, the car's warning light will blink.

OR and AND…

You can also **combine** tests for certain conditions. For example, we might want to flash the car's warning light when the car's fuel drops below 3 **or** if the engine becomes too hot. In that case, we might use a comparison operator called **Or.**

```
if ((_root.car.fuel <= 3) || (_root.car.engineTooHot == true)){
            _root.car.warningLight.gotoAndPlay("Blink");
      }
```

"Or" is represented by 2 vertical lines: **||**. (This is sometimes called the *pipe symbol* and is just above the *backward slash* key on the keyboard.) The above "If" statement will activate if *either* of these conditions are true. Both conditions have to be within their own pair of brackets, and both sets of brackets must be inside the "If" statement's own pair of brackets.

The **And** comparison operator is another useful one, and checks to see whether two or more different conditions are true. The "And" operator is represented by a double "And" symbol: **&&**. In the following example, the warning light only blinks if the engine is too hot <u>*and*</u> the fuel is low:

```
if ((_root.car.fuel <= 3) && (_root.car.engineTooHot == true)){
      _root.car.warningLight.gotoAndPlay("Blink");
      }
```

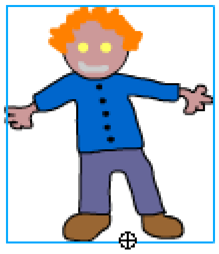
Using If Statements to Build a Simple Keyboard Control

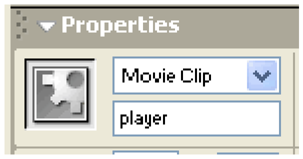
Follow the exercise below to create a simple object that can be controlled using the arrow keys on the keyboard:

1. Create a new movie. Set the Frame Rate (Frames Per Second) to **30**.

2. Think of a game that you would like to create, and create a new **Movie Clip Symbol** that represents the player's character. It might be a spaceship, a car or a person.

3. Copy an instance of this Movie Clip symbol onto the stage and give it the object name: **player**





4. Open the **Movie Clip Actions** of your **player** instance. Make sure that **Actions – Movie Clip** appears at the top of the Actions Panel. (If it doesn't click on the player Movie Clip instance you created above.) Copy the following ActionScript code into the Actions Panel:

```
onClipEvent(enterFrame) {

     if (key.isDown(39)) {
          this._x = this._x + 5;
     }

     if (key.isDown(37)) {
          this._x = this._x - 5;
     }

     if (key.isDown(40)) {
          this._y = this._y + 5;
     }

     if (key.isDown(38)) {
          this._y = this._y - 5;
     }

}
```

Don't forget the final curly bracket! Test your movie and see if it works.

How It Works:

The above bit of ActionScript introduces a few new actions. The first one is:

```
onClipEvent(enterFrame) {
                    }
```

onClipEvent (enterFrame) tells the computer that whatever is betweens its curly brackets, { } , should happen ***all the time***. This is a **loop**.

The ActionScript code that is inside this action repeats for as long as the object is on the stage. That means that the "If" statements inside the loop are checked 30 times every second. In games, things can change in a split instant, so it is important that your object is constantly looking for new information. As far as we're concerned, it loops forever.

And what is the computer looking for? 30 times every second it is looking for which keys the player is pressing so that it knows which direction to move the object in. That brings us to the next few lines:

```
                    if (key.isDown(39)) {
                         this._x = this._x + 5;
                    }
```

First, we have an "If" statement. The "If" statement checks to see if the Right Arrow is being pressed on the keyboard. "39" is the code for the right arrow. "`Key.isDown`" is a special action that listens for key presses. (*If you want to see all the keyboard codes, for all the keys, press "F1" in Flash to open the help file, click on "Action Script Reference" and then "Keyboard Key Code Values."*)

If the right arrow key is being pressed, the next line, "`this._x = this._x + 5;`" tells the computer to move *this* object's "x" position 5 pixels to the right. Remember, "_x" is the object's horizontal position.

What's "`this`"???

You could easily have used the following code, and the movie would have worked exactly the same way:

```
     if (key.isDown(39)) {
          _root.player._x = _root.player._x + 5;
     }
```

So why are we using an object name called "`this`"?

Because this ActionScript code were are using is *right inside* the object, we don't need to use the object's full name ("`_root.player`") *The object knows its own* *name*. We would only need to use its full name if we were trying to control its position from another object. We could use its full name if we wanted to, but in this case we don't need to.

Using "`this`" as the object name when your ActionScript is inside the object itself is very efficient. It has the following advantages:

- It saves you some typing – you don't have to write out the full object name.
- You can copy exactly the same code into another object and it will work exactly the same way, no matter what the object's name is. This means that you could create a whole library of useful code and copy it onto whichever object you would like to exhibit a particular behaviour. Because you've used "`this`" instead of the object name, the code will work for any object in exactly the same way.

The rest of the ActionScript code works in exactly the same way as the first "If" statement except that each following statement checks for another key being pressed. If no keys are being pressed, none of the "If" statements are activated and the object doesn't move.

Changing Direction

Very often you will want your objects to change the direction they are facing in when you press one of the arrow keys. For example, you might want your player character to face right when you press the right arrow key, and left when you press the left arrow key.

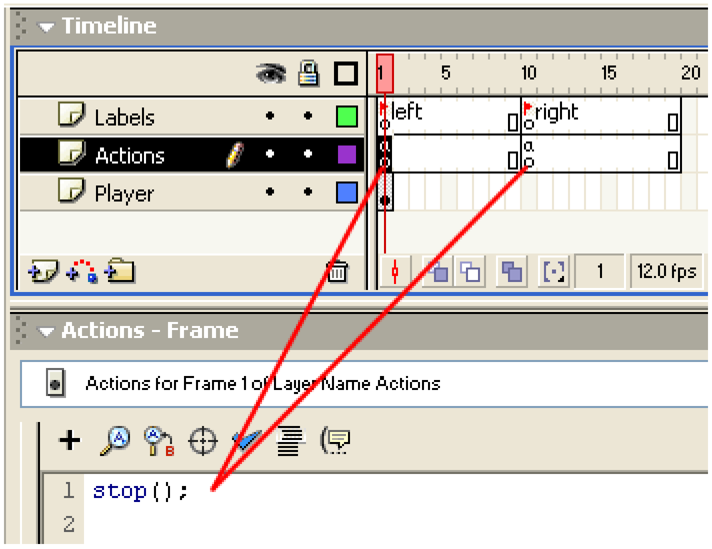
For this to work, you first need to add some labelled frames to you player's Movie Clip Symbol. You could call them "Right", "Left", "Up" and "Down". On each labelled frame, change the direction in which your object is pointing.

Next, you need to add some extra ActionScript to our example. Your ActionScript needs to tell Flash to jump to the correct labelled frame when an arrow key is pressed.

The following exercise shows you how you can easily add "right" and "left" directions to your player object.

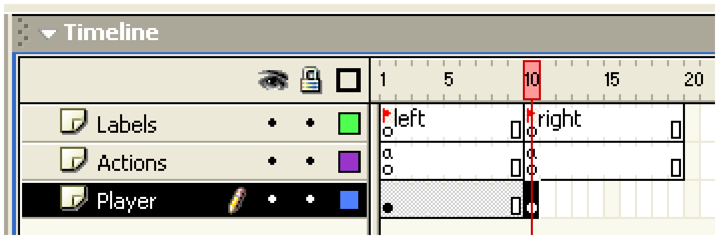
1. Open your player movie clip symbol. Add two layers: "`Labels`" and "`Actions`". Create two new labels, "`left`" and "`right`" spaced 10 frames apart. Add `stop()` actions to the Actions layers just below these labels.

   Your timeline should look something like this:

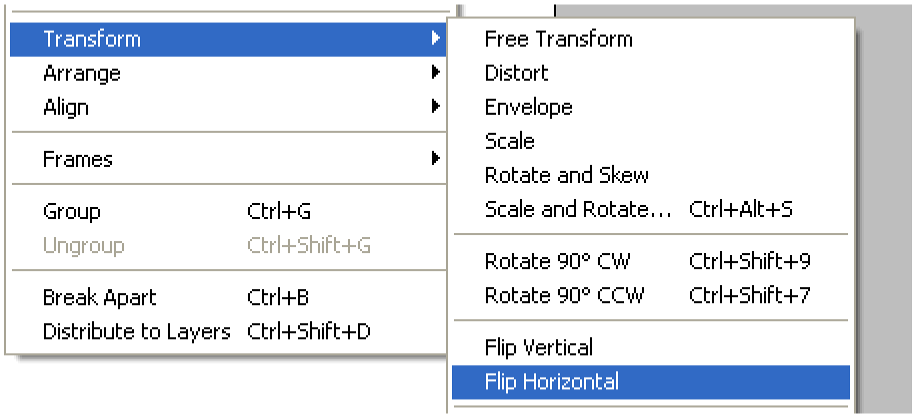


```
stop();
```

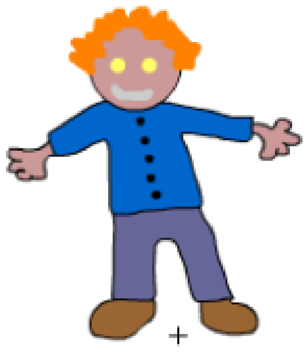
2. On the layer containing your player graphic, add a keyframe just below the “`right`” label.

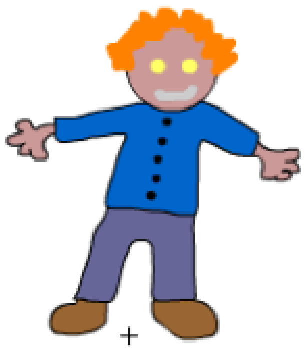


3. Select your player graphic. Click on the **Modify** menu, choose, **Transform** and then **Flip Horizontal**.



You should notice your player graphic change the direction in which it is facing.





Go back to the main stage, and open up the player's Object Actions. Change the code so that it looks like this (the modified sections are highlighted):

```
onClipEvent (enterFrame) {

     if (key.isDown(39)) {
          this._x = this._x + 5;
          this.gotoAndStop("right");
     }

     if (key.isDown(37)) {
          this._x = this._x - 5;
          this.gotoAndStop("left");
     }

     if (key.isDown(40)) {
          this._y = this._y + 5;
     }

     if (key.isDown(38)) {
          this._y = this._y - 5;
     }

}
```

Whenever the left and right arrow keys are pressed, the code tells the movie clip to go to and stop at the frames labelled either "`left`" or "`right`".

You've used the technique of jumping to labelled frames many times in the past, but it may not have occurred to you to use it in this context to achieve what is a very compelling effect. Knowing when to combine simple actions in this way is 90% of the challenge to creating games… and it doesn't get much more difficult than this. Think carefully about what you want to achieve and, more often than not, the solution will be a simple one.

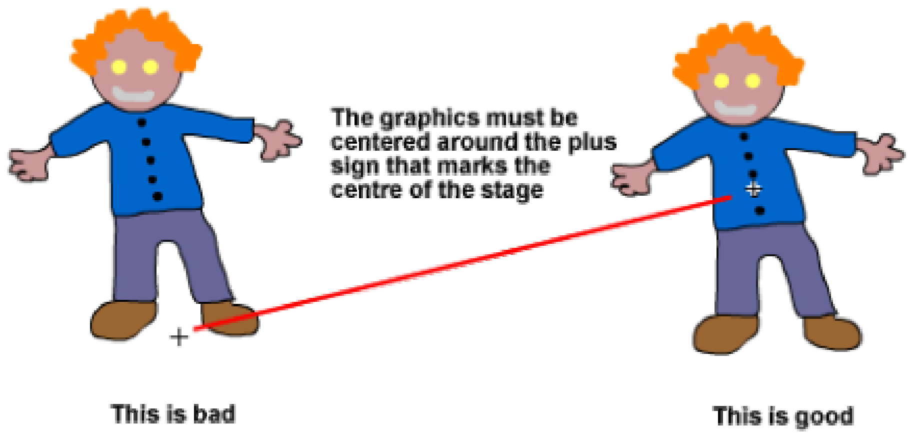
Introduction to Game Design
2) Screen Boundaries

You might have noticed that the object you created in the previous lesson disappears off the edge of the screen if you move it too far. It doesn't know where the boundaries of the screen are. Computers, and the things that we create with computers, only know as much as we tell them, so the next step is to tell our object where the edges of the screen are.

In computer games, there are usually two ways that objects react when they reach the edge of the screen. The first is, not surprisingly, they stop. The second, which is very common in space games especially, is that they *wrap* to the opposite side of the screen. We'll look at both of these one at a time.

1: Stopping At the Edges of the Screen

1. Make sure that your object is at the very centre of its own stage. Open up the player object's Movie Clip Symbol. Make sure that the graphics are centered right over the plus symbol (+) that marks the centre.


The graphics must be centered around the plus sign that marks the centre of the stage
This is bad
This is good

None of the ActionScript below will work the way you'd expect it to if the object is not completely centered. (*If you used "left" and "right" labeled frames from the previous lesson, you must make sure that the graphics from both frames are in the same place. Use the Properties Panel to help you with this.*)

2. <u>Add</u> the following highlighted ActionScript code <u>*inside*</u> the **onClipEvent (enterFrame)** action you created in the previous assignment:

```
onClipEvent(enterFrame) {

     if (key.isDown(39)) {
          this._x = this._x + 5;
          this.gotoAndStop("right");
     }

     if (key.isDown(37)) {
          this._x = this._x - 5;
          this.gotoAndStop("left");
     }

     if (key.isDown(40)) {
          this._y = this._y + 5;
     }

     if (key.isDown(38)) {
          this._y = this._y - 5;
     }

     //Check for Screen Boundaries

     if (this._y <= 0){
          this._y = 0;
     }

     if (this._y >= 400){
          this._y = 400;
     }

     if (this._x <= 0){
          this._x = 0;
     }

     if (this._x >= 550){
          this._x = 550;
     }
}
```

Remember… don't forget that final curly bracket! Test the movie and see what happens.

Let's look at the first "If" Statement:

```
     if (this._y <= 0){
          this._y = 0;
     }
```

The way this works if quite simple. If the _y position of the object is less than or equal to "0", it means that it must be at or beyond the top of the stage. Remember, the top of the stage is "0" and the bottom is "400". If the object's _y position is at or less than "0", its position should be forced back to the top of the stage. The line "`_y = 0`;" is what forces it back.

If we translated the first "If" statement of our ActionScript code into English, it might look something like this:

```
If ( I am at or beyond the top of the stage ){
     Set me back to the top;
}
```

This prevents the object from ever going past the top of the screen.

The three remaining "If" statements check to see if the object is beyond the right, left and bottom edges of the stage, and force it back in the same way.

You might be wondering about this line:

```
//Check for Screen Boundaries
```

This is **comment**. Comments are notes to yourself that you can write inside your code. Comments are preceded by two forward slashes: //. Anything you write after the forward slashes will not be processed as ActionScript. Comments are a useful way of organizing large, complex pieces of code, because they allow you to give each section a title.

You can also use comments to temporarily disable bits of code without deleting it. This is useful for debugging; if you want to see how your movie will run without a particular section of code but still keep it around in case you need to use it later.

To write longer comments that extend for more than one line, or to disable large sections of code, you can use this format:

```
/*
Anything between
these 2 symbols will
not be processed by
ActionScript
*/
```

2: Screen Wrapping

Screen Wrapping means making the object jump to the opposite edge of the screen when it reaches the screen boundaries. This is a way of making a kind of “infinite” screen, and is used in many types of games. Follow the directions below so that your object wraps around the edges of the stage:

1. Delete the four new “If” statements that you entered above in your Movie Clip Object Actions. (You can’t have both at the same time. If you want to temporarily disable the code, however, without deleting it, add `/*` at the beginning of the section of code and `*/` at the end of it.)

2. Add the following ActionScript code:

```
if (this._y < 0){
     this._y = 400;
}

if (this._y > 400){
     this._y = 0;
}

if (this._x < 0){
     this._x = 550;
}

if (this._x > 550){
     this._x = 0;
}
```

This works in a very similar way to the first example but this time, when the object reaches the edge of the screen, *it is forced to the opposite side*. Tricky, huh? Can you see how it is working?

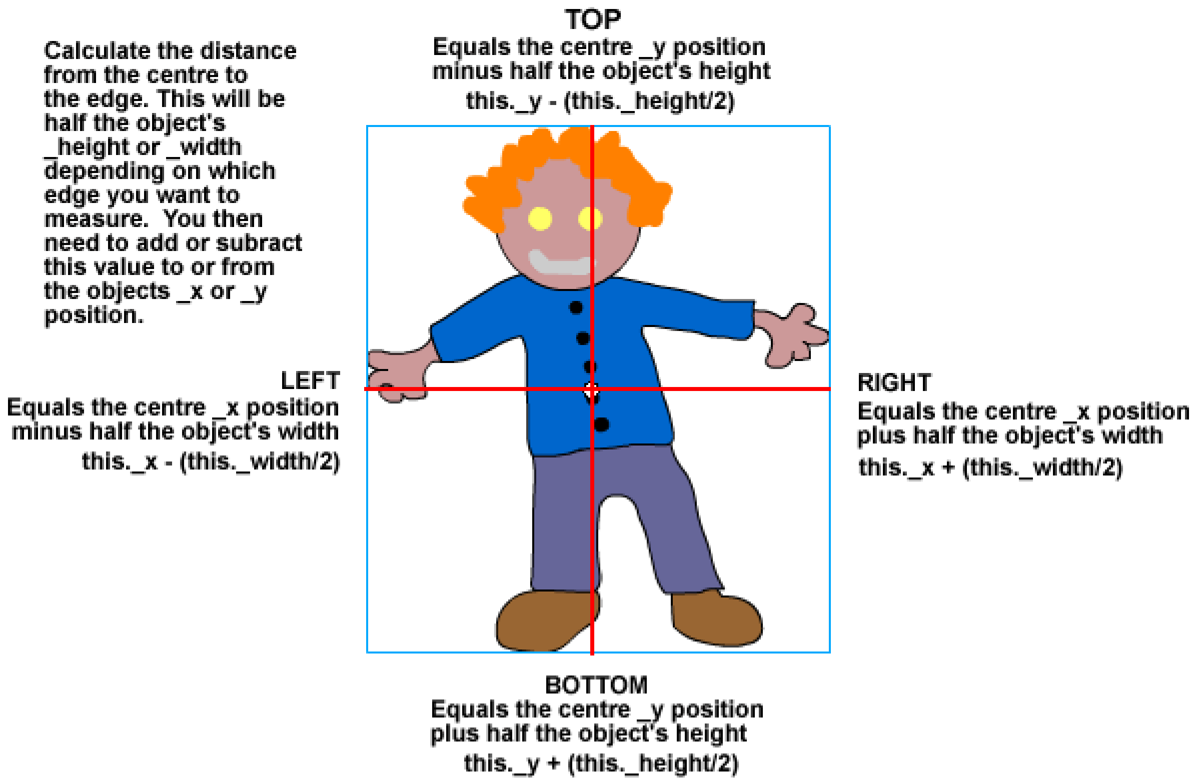
Advanced Screen Boundaries: Detecting Player Boundaries

You might have noticed a small problem with the above two examples. The object only stops or wraps when its *centre* goes past the edge of the screen. What if we want the object to stop when only its top edge hits the top of the screen, or wrap only when it has completely disappeared?

To do this, the object needs to know how big it is: it needs to know its height and width. Fortunately, ActionScript gives us very convenient properties to figure this out: “`_height`” and “`_width`” of course!

Now, let's think about how we can use "`_height`" and "`_width`" to figure this out:

The top position of our object equals the center _y position *minus* half of its height. The bottom position of our object equals the center _y position *plus* half of its height. Similarly, the right and left edges of the object are its _x position, plus or minus half its width.


Calculate the distance from the centre to the edge. This will be half the object's _height or _width depending on which edge you want to measure. You then need to add or subract this value to or from the objects _x or _y position.
TOP
Equals the centre _y position minus half the object's height
this._y - (this._height/2)
LEFT
Equals the centre _x position minus half the object's width
this._x - (this._width/2)
RIGHT
Equals the centre _x position plus half the object's width
this._x + (this._width/2)
BOTTOM
Equals the centre _y position plus half the object's height
this._y + (this._height/2)

Remember, when you move from left to right, the _x position increases. When you move from top to bottom, the _y position increases.

I know, this sounds a bit confusing. It is, until you actually see it in action, and think about how it is actually working.

To use this information to stop the edges of your player character from disappearing past the edges of the stage, you can use this code (replace the "If" statements you used earlier):

```
//Check for Screen Boundaries

if ((this._y - (this._height/2)) <= 0){
     this._y = 0 + (this._height/2);
}
```

```
if ((this._y + (this._height/2)) >= 400){
     this._y = 400 - (this._height/2);
}

if ((this._x + (this._width/2)) >= 550){
     this._x = 550 - (this._width/2);
}

if ((this._x - (this._width/2)) <= 0){
     this._x = 0 + (this._width/2);
}
```

Your final ActionScript code should now look like this:

```
onClipEvent(enterFrame) {

     //Keyboard Control

     if (key.isDown(39)) {
          this._x = this._x + 5;
          this.gotoAndStop("right");
     }

     if (key.isDown(37)) {
          this._x = this._x - 5;
          this.gotoAndStop("left");
     }

     if (key.isDown(40)) {
          this._y = this._y + 5;
     }

     if (key.isDown(38)) {
          this._y = this._y - 5;
     }

     //Check for Screen Boundaries

     if ((this._y - (this._height/2)) <= 0){
          this._y = 0 + (this._height/2);
     }

     if ((this._y + (this._height/2)) >= 400){
          this._y = 400 - (this._height/2);
     }
```

```
          if ((this._x + (this._width/2)) >= 550){
               this._x = 550 - (this._width/2);
          }

          if ((this._x - (this._width/2)) <= 0){
               this._x = 0 + (this._width/2);
          }
     }
```

This is one example of how surprisingly useful properties can be.

That should be enough to get you started, and enough to help you solve the following problem:

Modify your ActionScript so that the object wraps to the opposite side of the screen only after it has *completely disappeared.* This will make your screen wrapping look really believable.

Introduction to Game Design
3) “Bumping Into Things”: Collision Detection

What makes most computer games fun to play is that they are, in their essence, a simplified simulation of the real world. And, like the real word, they contain objects that you can interact with in some way. These objects might be walls that block your movement, friends that help you, or enemies that could harm you.

To create these sorts of interactive objects, you first need a way of finding out whether one object is touching another object. In computer game programming, this is called **collision detection**. Flash has a very simple method of detecting collisions between objects.

The hitTest Action:

hitTest is an action that checks to see if any two objects have bumped into one another. Let’s say that you have a Movie Clip object called “`car`” that the player can control. You also have a Movie Clip object called “`wall`”. In your game, if the player’s car hits the wall, it should crash.

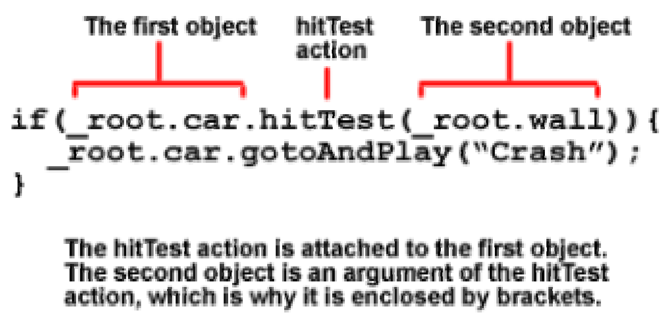
In plain English, we would want to write some computer code that looks something like this:

```
if (the car hits the wall){
     The car must crash;
}
```

We can translate this into ActionScript like this:

```
if(_root.car.hitTest(_root.wall)){
     _root.car.gotoAndPlay(“Crash”);
}
```

The **hitTest** action is attached to the end of the “`_root.car`” object. It has an argument, “`(_root.wall)`” which contains the name of the object that you want to check for a collision.



```
if(_root.car.hitTest(_root.wall)){
   _root.car.gotoAndPlay("Crash");
}
```

The hitTest action is attached to the first object. The second object is an argument of the hitTest action, which is why it is enclosed by brackets.

Usually, you would use the `hitTest` action inside the **conditional statement** of an **If Statement**. If the `hitTest` is *true*, the actions inside the If Statement activate. If it is false (if the objects are not touching) the actions inside the If Statement don't activate.

The examples on the following pages will show you how to use `hitTest` to:

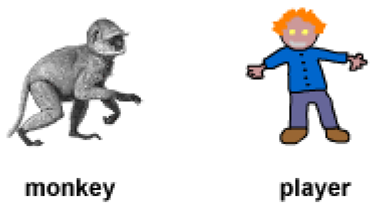
- Change a Dynamic Text Field
- Trigger an animation
- Block movement
- Update a score
- Reduce a "health meter"

Once you able to use these very simple techniques, you will be able to, with a little imagination, to produce a richly varied number of different kinds of games. All your hard work is about to pay off.

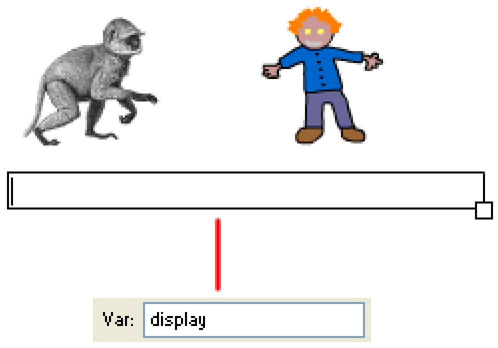
Exercise 1: Changing a Dynamic Text Field

The following example will show you how to use `hitTest` to change the text of a Dynamic Text field.

1. Open the example file that you were working on in the previous lessons. You should have a "player" object that you can move around the screen. Create a new object called "`monkey`"



2. Create a **dynamic text field** and assign it the variable name "`display`".



3. Open the **Actions** for the "`player`" object that you have been controlling with the keyboard. Enter the following ActionScript code *inside* the **onClipEvent(enterFrame)**, *after* your last If Statement, but *before* the last curly bracket. Only enter the code that is in **bold text** - the rest is just there to show you exactly where in your code to place it.

```
onClipEvent(enterFrame){

     …your previous "If" statements will be  here…

     //Collision Detection
     if(this.hitTest(_root.monkey)){
          _root.display = "You Hit the Monkey!";
     }

     }
```

Enter only this new code

4. Test the movie. You should notice the words "You Hit the Monkey!" appear when the player bumps into the monkey object.

Because the **hitTest** action is inside the player's **Movie Clip Actions**, you don't need to include the name of the object that is doing the hit-testing. The object knows its own name, and that's why we've used "`this`". You could, however, also use "`_root.player.hitTest`" (if your object is called "`player`") and that would work just as well.

You might have noticed that the message in the Dynamic Text field appears the first time the player bumps into the monkey, and then remains on the stage the whole time: even when the player is not touching the monkey. You can prevent this from happening by adding an "Else" statement to your ActionScript code:

```
if (this.hitTest(_root.monkey)) {
        _root.display = "You Hit the Monkey!";
   } else {
        _root.display = "You Have Not Hit the Monkey.";
   }
```

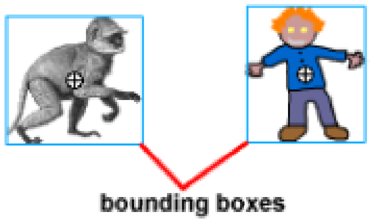
This is useful because whenever you have ***not*** hit the monkey, you get a message telling you so.

Exercise Extensions:

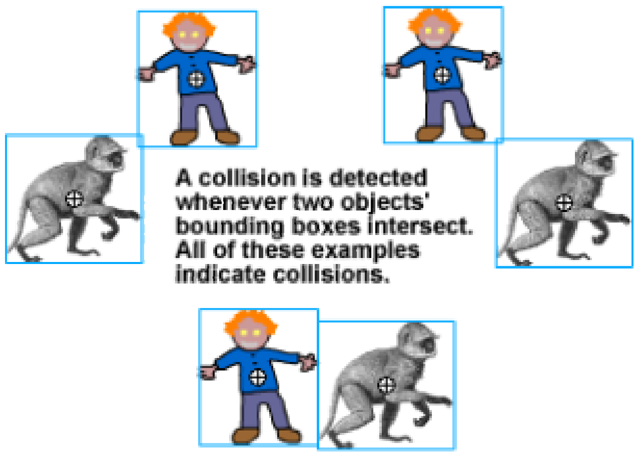
1. Animate the "monkey" Movie Clip so that it moves around the stage in an interesting way. This is an easy way to create a moving target.
2. Modify your ActionScript code so that the monkey becomes invisible when it is hit. Use the `_visible` property to accomplish this.
3. Modify your code so that the monkey moves to a different place on the stage when it is hit. Use the `_x` and `_y` properties to accomplish this

Bounding Boxes

Flash detects a collision between two objects when the **bounding boxes** of those objects overlap. Bounding boxes are invisible boxes that surround an object. When you select an object on the stage with the mouse, you can see the bounding box as a blue box that surrounds the object.



A collision is detected whenever any portion of the bounding box intersects with any portion of another object's bounding box. The following are some examples:


A collision is detected whenever two objects' bounding boxes intersect. All of these examples indicate collisions.

You should notice a problem with this right away. Collisions can be detected even though the actual graphics of the objects themselves are not intersecting.

This is actually not a flaw with Flash, but a technical problem that all game designers face. The reason it exists is that modern computers are just not fast enough to be able to calculate intersections of exact shapes fast enough to update the screen so that the game is smoothly playable. Using a bounding box is the most efficient way to check for collisions because the computer only has to calculate the shape of a rectangle. This is mathematically very easy to do, and therefore doesn't put much strain on the computer's processor. If the computer was forced to check every pixel of every object to see whether or not it was intersecting with every pixel of every other object, the game would be unplayably slow.

Of course, this can be a problem if, as in the above examples, a collision is being detected when none seems, visibly, to be occurring. There are 3 ways around this problem:

- Create squarish or rectangular shaped objects in your games. This is the simplest solution, and is the one that most game designers rely on. Even if the collision detection doesn't appear to be exact, it's unlikely that the player will notice if the objects in the game are moving quickly. If you look carefully at most 2D games, you'll notice that player and enemy characters are usually rounded or squarish in shape. The fact that you may not have noticed this before is a tribute to the game designer's skill at working around this technical limitation.

- Detect for collisions using **sub-objects**. For example, let's say your player object has a sub-object inside it called "`head`". Instead of checking for a collision between the monkey and the player, you could check for a collision between the monkey and the player's head. To do this, you might use a line of ActionScript code that looks like this:

  ```
  if (_root.player.head.hitTest(_root.monkey)) {
       … actions …
  }
  ```

  In this example, a collision would only be detected when the player's head (which is a movie-clip sub-object inside the player object) collides with the monkey. You could set up similar collision detection code for the player's hands and feet to create very accurate detection.

  However, be careful not to use too many sub-objects. The more sub-objects you use, the more the computer's processor has to work, and more you risk slowing down the smooth play of your game.

- Check for the intersection of an object's shape and a specific point on another object. This is an advanced technique that is built into `hitTest` and requires the use of special built-in variable called **shapeFlag**. We will cover the use of **shapeFlag** in a later lesson.

Exercise 2: Triggering An Animation

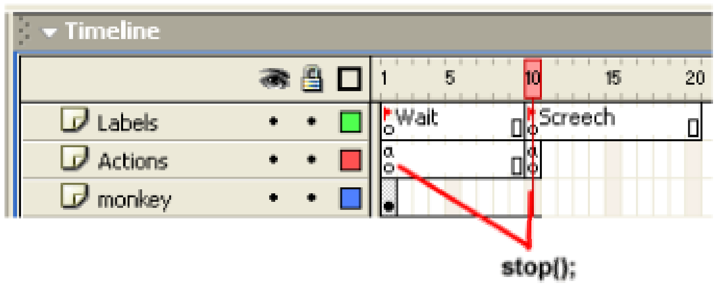
When 2 objects collide in a game, you will often want to trigger some kind of animation, such as an explosion. This is a very easy effect to create. All you need to do is create a labeled frame ***inside*** one of your Movie Clips and tell Flash to go to that frame when a collision occurs.

The following exercise walks you though this process:

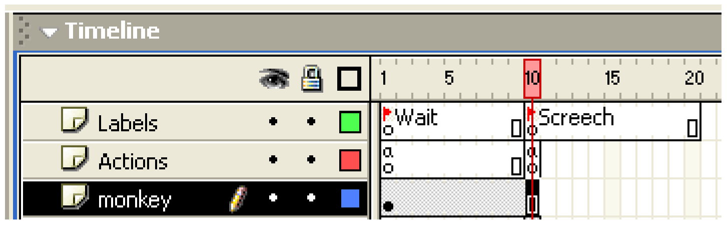
1. Open up the "`monkey`" Movie Clip and, on the **Actions Layer**, add **stop()** actions at frame **1** and frame **10**.

2. On frame **1** of your **Labels Layer**, create a label called "Wait". On frame **10** of your **Labels Layer** create a labeled frame called "Screech". Insert a blank frame (F5) on frame 20 so that you can easily read the label.

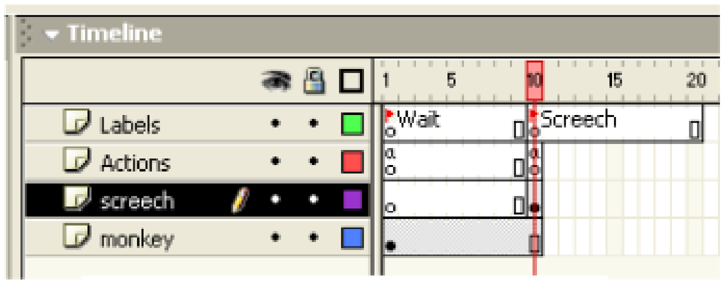
Your layers should now look like this:



3. On the layer containing the monkey graphic, insert a blank frame (F5) on frame 10, so that the image is extended into that frame.



4. Create a new layer called "bubble" and insert a keyframe (F6) on frame 10. Draw a speech bubble with the word "Eeek!" inside it:



5. Back on the **main stage**, open the player's Movie Clip Actions and add the following code to the If Statement that checks for the collision with the monkey:

```
_root.monkey.gotoAndStop("Screech");
```

Your ActionScript code should now look like this:

```
if (this.hitTest(_root.monkey)) {
     _root.display = "You Hit the Monkey!";
     _root.monkey.gotoAndStop("Screech");
} else {
     _root.display = "You Have Not Hit the Monkey.";
}
```

6. Test your movie.

You can easily use this technique to create explosions or other effects in your games.

Exercise Extension:

Modify the exercise so that the monkey's speech bubble disappears when the player is not touching it.

Exercise 3: Blocking Movement

A very useful thing to be able to do in games is to block a player, or another object, from moving into a certain area. The following code shows you a simple trick you can use to do this:

```
if(this.hitTest(_root.monkey)){
     this._x = oldXPosition;
     this._y = oldYPosition;
     }

oldXPosition = this._x;
oldYPosition = this._y;
```

(Remember, all this *must be inside the onClipEvent(enterFrame) action*)

How It Works:

Two new variables are used, "`oldXPosition`" and "`oldYPosition`". They keep track of the *previous* _x and _y positions of the player. When the player hits the square, the player's _x and _y properties are told to go back to what they were a split moment before. And this stops the object from moving further.

Remember that everything inside an **onClipEvent(enterFrame)** action loops *continuously*. If you have set your frame rate to 30 frames per second, all the actions inside **onClipEvent(enterFrame**) will be activated *30 times each second - forever*. And, what's really important, in this case, is that the actions are scanned from top to bottom. This means that actions at the top are activated before actions at the bottom. And, if you change the value of a variable at the bottom, the same variable at the top will still be using the old value - until the next scan.

So, what this means is that, on the first scan of the code, "`oldXPosition`" and "`oldYPosition`" store the *current* value of _x and _y. But on the second scan, the next frame, if the object has moved, they would be storing the *previous* values. That means that we can use those previous values to force the object back to where it was just before the current frame.

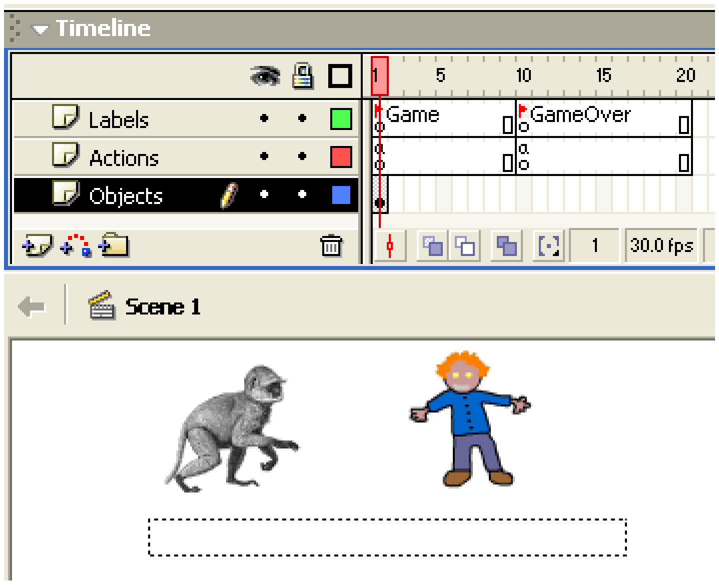
Don't worry if you don't understand this!!! It *is* complicated. The most important thing is that you know you can use this trick if you need to, and that it works.

Exercise 4: Updating a Score

Most games keep track of whether a player has won or lost by updating a score, based on how well the player is performing. The following exercise will show you how you can update a score and end the game when a certain score has been reached.

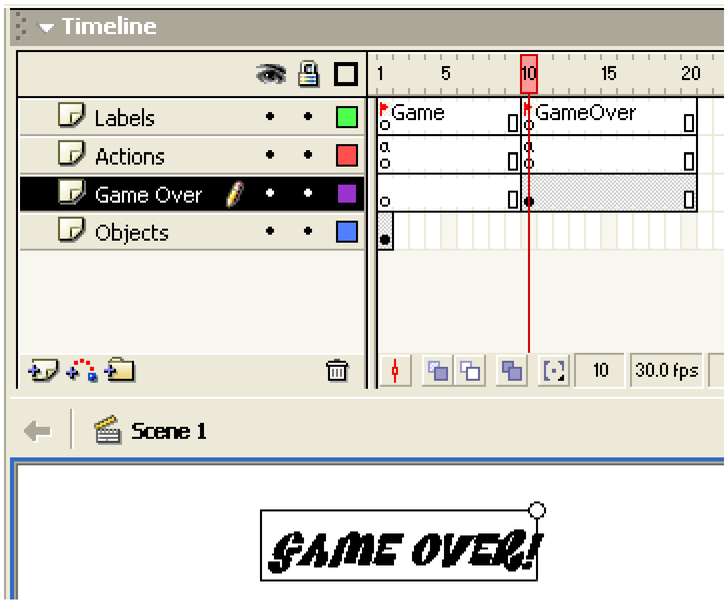
The first thing you need to do is to set up a labeled frame on the **main timeline** that tells the player that they have won the game:

1. Set up your main timeline so that it looks like the image below:

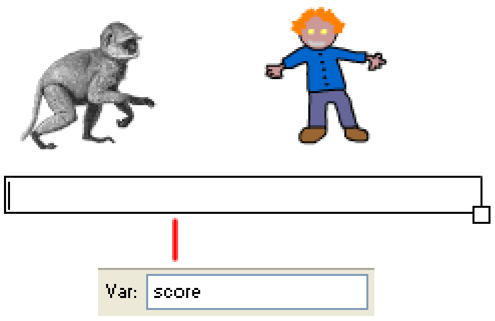


Create 2 new layers, "Labels" and "Actions". On the Labels layer, add the label name "Game" on frame 1 and "GameOver" on frame 10. On the Actions layer, add stop actions on frames 1 and 10.

2. Create a new layer, and on it insert a keyframe on frame 10. Write the words "Game Over!" on the stage with the text tool:



3. Finally, on frame one, change the variable name of your Dynamic Text field to "`score`".



4. The next step is to add the ActionScript. Change the code on your player object that checks for collision detection so that it looks like this:

```
if (this.hitTest(_root.monkey)) {
     _root.score = _root.score + 1;
}
if (_root.score >= 100) {
     _root.gotoAndStop("GameOver");
}
```

Test your movie and see what happens. You should notice that while the player is touching the monkey, the score increases. As soon as the score reaches 100, "Game Over!" is displayed on the screen.

There is nothing really new here. You've seen these techniques before, but it might not have occurred to you to use them in this way to control the flow of a game.

There is one detail that should be pointed out, however:

```
_root.gotoAndStop("GameOver");
```

In the above line, we are telling the _root timeline, *not the object*, to go to a labeled frame called "`GameOver`". Remember that all this code is *inside* our player object. If we want to control something that happens on the main timeline or inside another object, we have to use **_root.** and then the name of that object. The main timeline is called "_root", however, so if want to trigger actions on it we simply use **_root.** and then the action name.

Remember, you can also use the **increment operator** (a double plus sign) to increase the value of your "`score`" variable by one:

```
if (this.hitTest(_root.monkey)) {
            _root.score++;
      }
```

This is a slightly more efficient way of writing your code.

Fine-Tuning Your Score Keeping

In the previous example you would have noticed that the score continuously updates while the two objects are colliding. Very often, you will only want the score to update once, on the first collision, and not every moment that the objects are touching.

To accomplish this you need some way of figuring out when the objects have collided for the first time, and then prevent the score from updating until the objects collide again later. The easiest way to do this is by using a new variable to keep track of the state of the collision. In the example below, we will use a variable called "`iAmNotColliding`" to do this, although you could use any other variable name that you choose.

Beware, however: you should only attempt the exercise below if you have a firm grip of how If Statements work. You may whish to come back to this exercise later after you have had a little more experience programming with ActionScript. It's not difficult as such, but the logic involved is slightly on the mind-numbing side. Proceed with caution….

The first step is to **initialize** the new variable that we will use to keep track of the state of the collision. To do this, we will use a new action called **onClipEvent(load)**.

Enter the following *before* the onClipEvent(enterFrame) action:

```
onClipEvent(load){
     iAmNotColliding = true;
     }
```

Your ActionScript code should now look something like this:

```
onClipEvent(load){
     iAmNotColliding = true;
     }

onClipEvent(enterFrame) {
     …check for key presses
     …check for screen boundaries
     …collision detection
}
```

Any actions that you enter inside an **onClipEvent(load)** action *only run once: the very first time Flash loads the object onto the stage*. You should always use **onClipEvent(load)** to initialize variables that your objects will need to use. The **onClipEvent(load)** action should *always be added above* the `onClipEvent(enterFrame)` action.

We've initialized the "`iAmNotColliding`" variable to "`true`" so that our object knows that, when it is first loaded onto the stage, it is not colliding with anything.

Next, change your collision detection code so that it looks like this:

```
if ((this.hitTest(_root.monkey)) && (iAmNotColliding == true)) {
      _root.score = _root.score + 1;
      iAmNotColliding = false;
}
if ((!this.hitTest(_root.monkey)) && (iAmNotColliding == false)) {
      iAmNotColliding = true;
}
if (_root.score >= 10) {
      _root.gotoAndStop("GameOver");
}
```

Here's where the mind-numbing logic comes in. Let's walk through it in plain English:

If the player is colliding with the monkey, and has not collided before, then we should increase the score by one, and tell the player that it is currently colliding.

If the player is not colliding with the monkey, but has collided before, then we should tell the player that it is no longer colliding, so that it is free to collide again in the future.

If the score is greater than or equal to ten, go to the labeled frame on the main timeline called "GameOver"

Yes, I know this is pretty tricky! This is a fairly complex use of logical operators and If Statements. Don't feel discouraged if you don't understand it right away or not are able to write similarly complex code yourself any time soon. Look it over a few times, think about it while lying in bed a night, come back to it a few days later and it will gradually start to make sense. Do, however, feel free to use it whenever you need to use this kind of effect. A crucial aspect to learning how to program is to see how other people have solved problems and then to use those solutions to solve your own problems. This is how everyone learns to program.

Although all of the basic mechanics behind the above code have been covered in previous lessons, there is one line that is worth clarifying, as it involves quite an unexpected use of the **Not** operator, an exclamation mark: **!**. It's this line here:

```
if ((!this.hitTest(_root.monkey)) && (iAmNotColliding == false))
```

The **Not** operator is used to tell Flash when it should be on the lookout for things that are *not* happening. Often, this can be as important as knowing when things *are* happening.

We can translate the section of code…

```
if ((!this.hitTest(_root.monkey))
```

…as…

*if this object is **not** colliding with the monkey.*

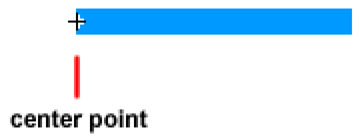
This is useful to keep in mind if you ever need to check something similar in the future.

Exercise 5: Implementing a Health Meter

Many games employ the use of a "Health Meter" to determine when the game is over. When the player bumps into bad things, like enemies, the health meter gradually shrinks in size. When the health meter disappears, the game ends.

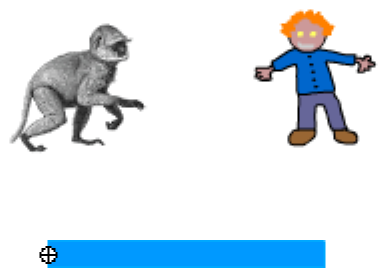
Implementing a health meter is very easy. It makes clever use of the Movie Clip object's `_xscale` property. The following example demonstrates how to create one.

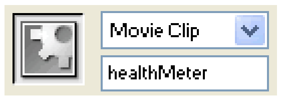
1. In the same file that you were working on previously, create a new Movie Clip Symbol called "Heath Meter". In the Symbol Editing window, draw a long bar on the stage:

   Make sure you draw the heath meter so that the Movie Clip's center point is on the very left hand side.

2. Drag an instance of the health meter onto the stage and give it the object name "`healthMeter`"

3. Modify your player object's collision detection code so that it looks like this:

```
if (this.hitTest(_root.monkey)) {
    _root.healthMeter._xscale--;
}
if (_root.healthMeter._xscale <= 0) {
    _root.gotoAndStop("GameOver");
}
```

Test the movie. You will notice that while the player is touching the monkey, the health meter gradually shrinks. When it reaches zero, the "Game Over!" screen is visible.

In the following line we used the **decrement operator** (double minus sign):

```
if (this.hitTest(_root.monkey)) {
     _root.healthMeter._xscale--;
}
```

You could, however, write the line like this:

```
if (this.hitTest(_root.monkey)) {
      _root.healthMeter._xscale = root.healthMeter._xscale - 1;
}
```

It's more typing, but it will work exactly the same way.

Also important to keep in mind is that you can use the values of object properties inside If Statement, to check any condition of the game, as in this line:

```
if (_root.healthMeter._xscale <= 0) {
     _root.gotoAndStop("GameOver");
}
```

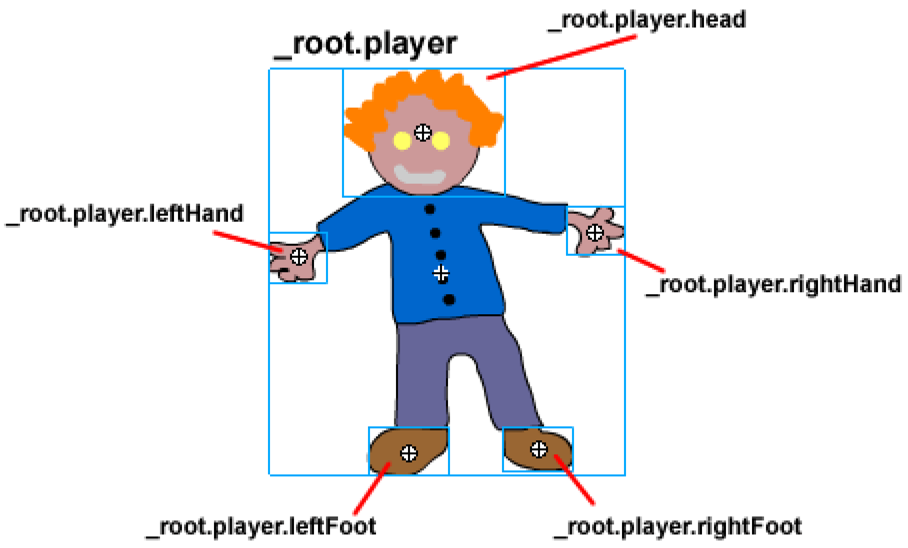
We are using the `_xscale` property of the "`healthMeter`" Movie Clip object itself, not a variable, to determine when the game should end. You can use any object properties in this way – and this is could be an extremely useful tool when you are creating a game. Keep it in mind.

Introduction to Game Design
4) Advanced Collision Detection

There may be many instances were collision detection using a Movie Clip object's bounding box will just not be accurate enough for the kinds of games you will want to create. To increase collision detection accuracy, you can set up detection using **sub-objects** or **points**.

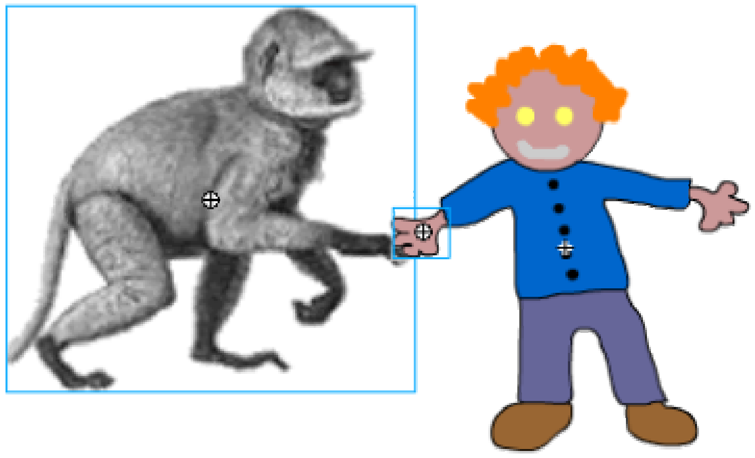
Using Sub-Objects

You can easily make your collision detection more accurate by creating smaller objects inside the main objects to check for collision. For example:



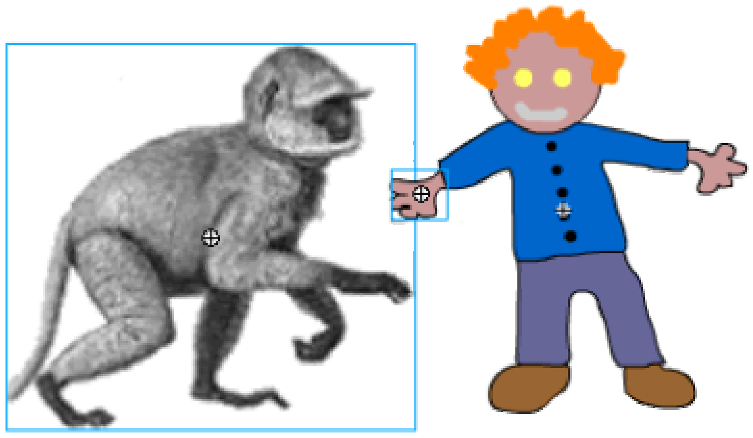
Inside the `_root.player` object are other movie clip symbols. Instead of using the main `_root.player` object to check for collisions, you can use the sub-objects, like this:

```
if (_root.player.leftHand.hitTest(_root.monkey)) {
     … collision actions …
     }
```

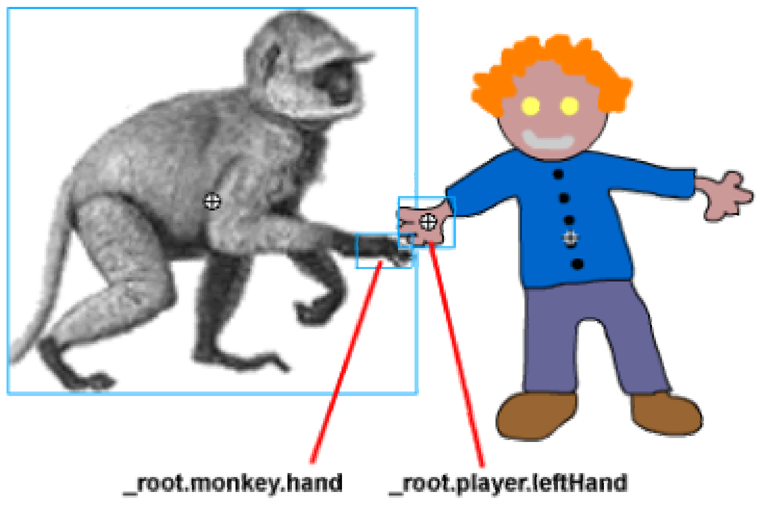


In this example a collision will only be detected if the player's left hand is intersecting with the monkey's bounding box.

Although this is more accurate, a collision will still be detected if even if the player's left hand intersects with an empty area of the monkey's bounding box, such as in this example:



You can increase the accuracy of this technique by creating sub-objects *in the other object* as well:



```
if (_root.player.leftHand.hitTest(_root.monkey.hand)) {
      … collision actions …
      }
```

In this example a `hand` object was created inside the `monkey` object, and a collision set to be detected only if it intersects with the player's `leftHand`.

(*By the way, the easiest way to create a sub-object inside an object that already exists is to select an area with the arrow tool and press F8 to convert it into a Movie Clip Symbol. This is useful because it means that you don't have to decide on what your sub-objects will be until after you've created your main object.*)

If you create enough sub-objects at strategic spots inside your main objects, you can have very accurate collision detection. You will, of course, also need very many If Statements to check for all these collisions. In addition to vastly increasing the amount of typing and debugging that you will need to do, your game many also start to slow down due to the extra processing required to do all this collision detection.

Careful use of this technique, however, will help you solve most of the accuracy problems that you are likely to run into.

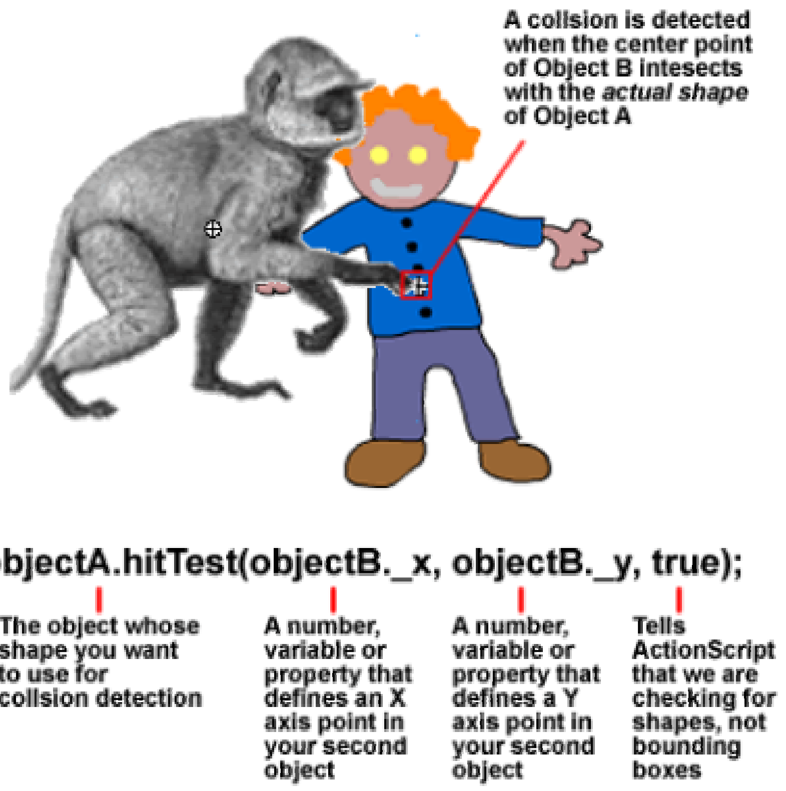
Detecting Shapes

There is another version of **hitTest** that looks like this:

```
objectA.hitTest(objectB.pointX, objectB.pointY, true);
```

This allows you to check to see if a *single point* inside "objectB" is touching the actual shape of "objectA." If it is, a collision is detected.

Let's have a look at how this would work with our example:


A collision is detected when the center point of Object B intesects with the *actual shape* of Object A
objectA.hitTest(objectB._x, objectB._y, true);
The object whose shape you want to use for collsion detection
A number, variable or property that defines an X axis point in your second object
A number, variable or property that defines a Y axis point in your second object
Tells ActionScript that we are checking for shapes, not bounding boxes

```
if (_root.monkey.hitTest(_root.player._x, _root.player._y, true)) {
          … collision actions …
     }
```

In this example we are checking to see whether the center `_x` and `_y` position of the player is intersecting with the actual shape of the monkey. If it is, a collision is detected.

"`true`" is simply used to tell ActionScript that we are checking for a collision between this point and the other object's shape. If we change "`true`" to "`false`", the objects' bounding boxes will be checked for a collision instead.

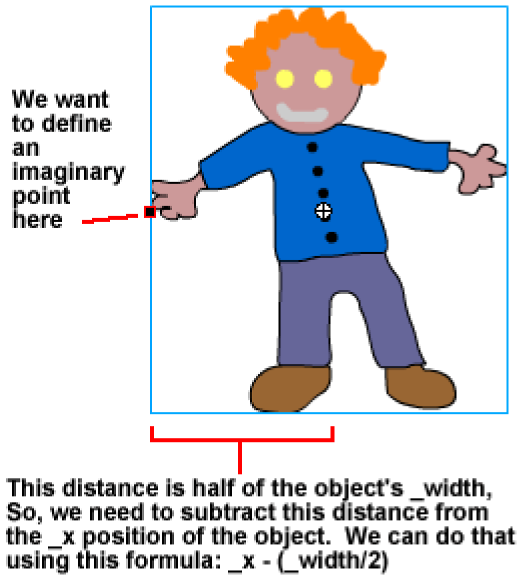
You don't just need to use the center point of the object, however. *You can use any point*, even one that is defined mathematically. For example, you could use code that looks like this:

```
pointX = _root.player._x;
pointY = _root.player._y;

if (_root.monkey.hitTest(pointX, pointY, true)) {
          … collision actions …
```

```
      }
```

In this example, the center `_x` and `_y` position of the player were defined as variables, and these variables were used in the collision detection code. The advantage of this is that we can change the points we want to use simply by changing how the variables are defined.

Let's say we wanted to define a point that was on the left side of the player:


We want to define an imaginary point here
This distance is half of the object's _width, So, we need to subtract this distance from the _x position of the object. We can do that using this formula: _x - (_width/2)

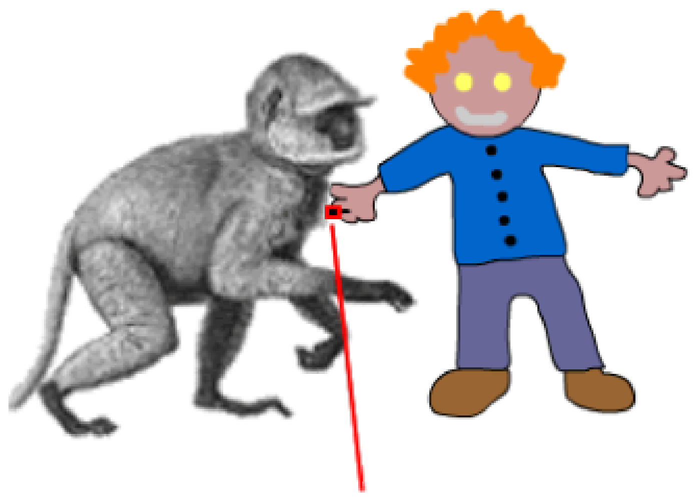
We define our variables so that the X and Y position of this point is defined:

```
pointX = _root.player._x - (_root.player._width / 2);
pointY = _root.player._y;
```

We can then use the same collision detection code:

```
if (_root.monkey.hitTest(pointX, pointY, true)) {
            … collision actions …
      }
```

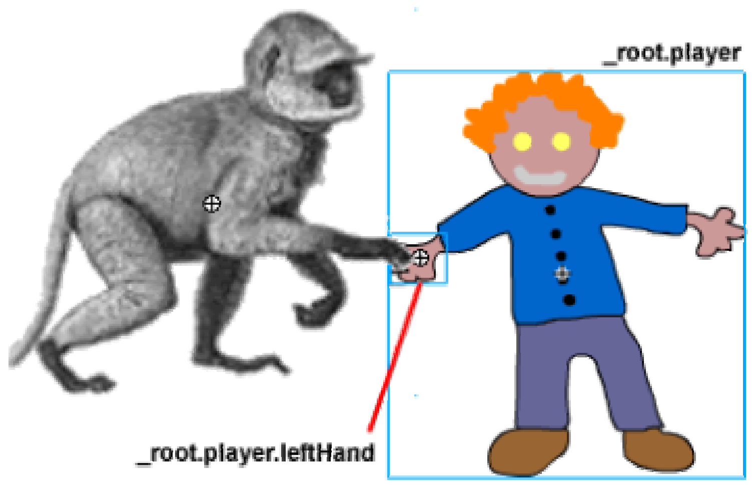
Now, a collision will be detected when the newly defined point intersects with the shape of the monkey:



A collision is detected when the newly defined point intersects with the monkey's shape

You can define as many points like this as you need to for your objects, and set up individual collision detection code for each of them. By carefully choosing the points on your object that are most likely to intersect with the shapes of other objects, you can create very accurate collision detection.

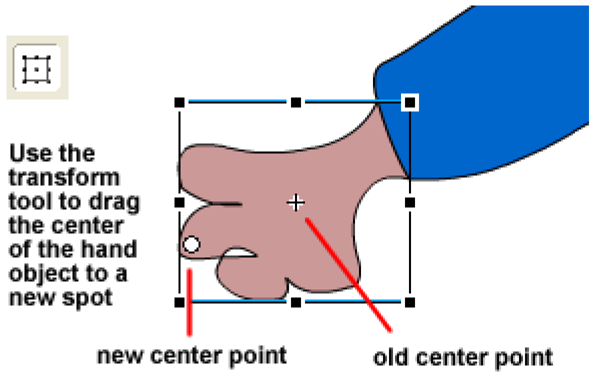
You can define points within sub-objects just as easily. And, doing so is often more convenient, such as in this example:



```
pointX = _root.player.leftHand._x;
pointY = _root.player.rightHand._y;

if (_root.monkey.hitTest(pointX, pointY, true)) {
          … collision actions …
     }
```

In this case, the center point of the `leftHand` sub-object is being tested against the shape of the monkey. This is convenient because it saves us from having to mathematically calculate the position of the point.

To increase accuracy, you can position the center point of the `leftHand` sub-object to the edge of the hand using the Transform tool.


Use the transform tool to drag the center of the hand object to a new spot
new center point
old center point

This method will give you very accurate collision detection without having to do any math… and that has to be good!

But Why Isn't There A Simpler Way???

That's a good question. Why can't Flash just detect the entire shape of an object straight from the beginning? Why does it need to use inaccurate bounding boxes or complicated individual points?

The reason for this has to do with what's going on "behind the scenes" in our computers. When we create a game in Flash, Flash is actually translating our ActionScript code into *binary machine language* that our computer can understand. As it turns out, describing to a computer exactly where a shape begins and ends is an *extremely* complex and processor-intensive thing to do. If your computer had to do these calculations, it would make your games so slow that they would be unplayable. To accurately define the shape of an object, Flash would have to calculate the position of hundreds of tiny points around the edge of the object, check those against the points of other objects (which are probably moving) *30 times per second!* Modern computers are just not fast enough to do this.

Don't feel too discouraged, however: *all* computer games, not just ones created in Flash, currently only use bounding boxes and individual points for collision detection. And, actually, they're really all you need.

If you look at your own objects carefully, you'll notice that there are usually only very few points that that will *ever* come into contact with another object. Find these points on your objects, and you'll have a very accurate collision detection system using as few points as possible.

VERY Advanced Collision Detection…Proceed With Caution…!!!

The above examples should be more than enough to get you started using ***hitTest*** *for very accurate collision detection. However, as your games become more complex, you'll find that you need to fine-tune your collision detection even further. Until that time, however, feel free to skip this next section as it may just confuse you. Also, you'll understand it more once you've had some practice working on your own projects.*

If you're using a lot of individual points or sub-objects to check for collisions, you'll soon start to notice that your game becomes slower and slower to play. That's because your computer has to spend more and more processing power checking to see if all these objects and points are colliding with one another. The more objects you have, the slower your game will become.

To fix this, you need to make sure that you don't check for a collision *until you absolutely have to*. There are lots of ways to do this, but here's a particularly useful one:

First of all, set up your collision detection so that that it *only* checks for a simple collision between bounding boxes, like this:

```
if(objectA.hitTest(objectB)){
     …a collision has occurred…
}
```

This will tell you that the two objects are within a general proximity of one another.

Next, check for a collision between your points or sub-objects *inside* the first **hitTest**:

```
if(objectA.hitTest(objectB)){
     if(objectA.top.hitTest(objectB.body))
     || if(objectA.bottom.hitTest(objectB.body))
     || if(objectA.left.hitTest(objectB.body))
     || if(objectA.right.hitTest(objectB.body)))
     {
          …a collision has occurred…
     }
}
```

This saves the computer from having to process 4 additional If Statements until the two objects' bounding boxes are touching. If you use this technique, you'll notice a significant improvement in the speed and responsiveness of your game. And in video games, responsiveness and speed (often called *performance*) are the most important factors.

If you have a little knowledge of basic computer programming, and you feel comfortable with the above information, you can make your ActionScript code more efficient by looping through the objects you want to test using a **for loop**:

```
if(objectA.hitTest(objectB)){
     for(i= 1; i>=4; i++;){
          if(objectA.hitTest([“objectB.subObject” + i]){
               …a collision has occurred…
          }
     }
}
```

For this to work, “objectA” would need to have 4 sub-objects called “subObject1”, “subObject2”, “subObject3” and “subObject4”.

The **for loop** loops through the second If Statement 4 times, and each time checks a different sub-object.

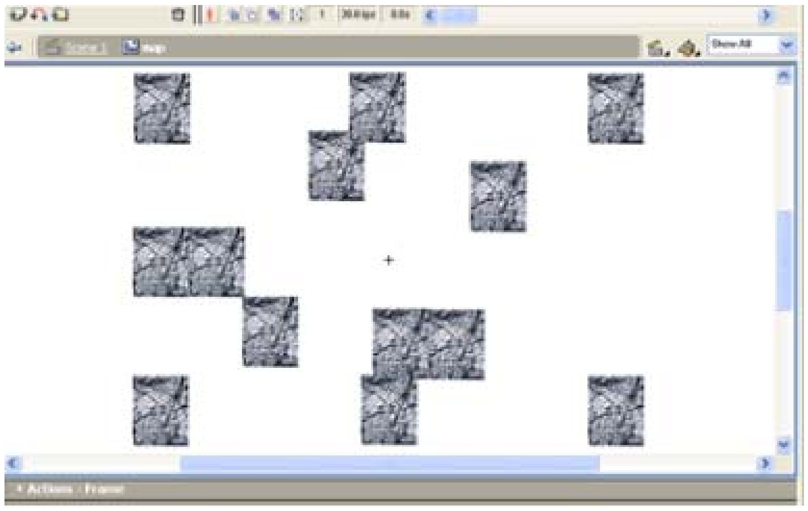
Introduction to Game Design
5) Creating a Maze

The following exercise shows you how you can create the basis of a simple maze game using the Advanced Collision Detection techniques explained in the previous lesson. Please make sure that you understand those techniques before you proceed.

In order to create a game in which the player needs to move through a complex maze, you need to know how to use the shape collision detection technique. The following example demonstrates how:

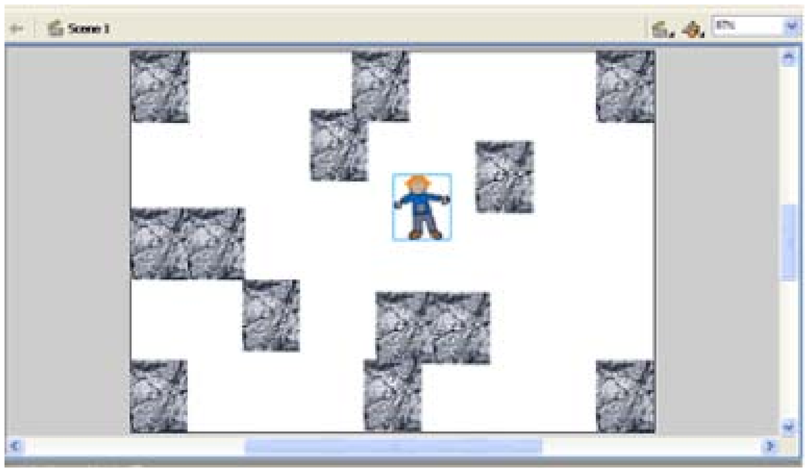
1. Create an object on the stage called "`player`", or use the same "`player`" object that you've been using in the previous lessons. Make sure that you can move the object around the stage with the arrow keys.

2. Create a new Movie Clip Symbol called "`map`". In the map symbol, design your maze or environment.


Design your maze inside a Movie Clip Symbol. Or, create it on the main stage, select all the maze elements together, and convert it into a Movie Clip Symbol.

   You might find it easier to actually design your map on the main stage, and then convert it into a symbol. This way you can see exactly where the stage boundaries and your other objects are. You can select multiple objects by holding down the Shift key. Press F8 to convert the selected objects into a symbol.

3. Drag an instance of the map symbol onto the stage and give it the object name "`map`".



4. Enter the following in the **onClipEvent(enterFrame)** action of your “`player`” object:

```
//Collision Detection
if (_root.map.hitTest(this._x, this._y, true)) {
     this._x = oldXPosition;
     this._y = oldYPosition;
}
oldYPosition = this._y;
oldXPosition = this._x;
```

Test your movie and see what happens. Whenever the player’s center _x or _y point touches the shape of the map, the player is blocked.

The entire ActionScript code attached to you player, including keyboard control and screen boundaries, should now look like this (the new code is highlighted.):

```
onClipEvent (enterFrame) {
     //Keyboard Control
     if (key.isDown(39)) {
          this._x = this._x + 5;
     }
     if (key.isDown(37)) {
          this._x = this._x - 5;
     }
     if (key.isDown(40)) {
          this._y = this._y + 5;
     }
     if (key.isDown(38)) {
          this._y = this._y - 5;
     }
     //Check for Screen Boundaries
     if ((this._y - (this._height / 2)) <= 0) {
          this._y = 0 + (this._height / 2);
     }
     if ((this._y + (this._height / 2)) >= 400) {
          this._y = 400 - (this._height / 2);
     }
     if ((this._x + (this._width / 2)) >= 550) {
          this._x = 550 - (this._width / 2);
     }
     if ((this._x - (this._width / 2)) <= 0) {
          this._x = 0 + (this._width / 2);
     }
     //Collision Detection
     if (_root.map.hitTest(this._x, this._y, true)) {
          this._x = oldXPosition;
          this._y = oldYPosition;
     }
     oldYPosition = this._y;
     oldXPosition = this._x;
}
```

Again, don't forget that final brace!

This is essentially a combination of shape based collision detection and the technique used in earlier lessons to block movement.

Unfortunately, we're only halfway there. You probably don't want to create a maze game where the player's center point intersects with the maze, but when its *edge* does.

To do this, you first need define the top, right and left edges for the player. The easiest way to do this is to add this bit of ActionScript code (*which should* ***replace*** *your previous code*:)

```
     //Collision Detection

     //Define Player Boundaries:
     top = this._y - (this._height / 2);
     bottom = this._y + (this._height / 2);
     right = this._x + (this._width / 2);
     left = this._x - (this._width / 2);

     //Check for Collision with Map:
     if ((_root.map.hitTest(left, this._y, true))
     || (_root.map.hitTest(right, this._y, true))
     || (_root.map.hitTest(this._x, top, true))
```

```
        || (_root.map.hitTest(this._x, bottom, true)))
        {
        //Block Movement:
        this._x = oldXPosition;
        this._y = oldYPosition;
}

oldYPosition = this._y;
oldXPosition = this._x;
```

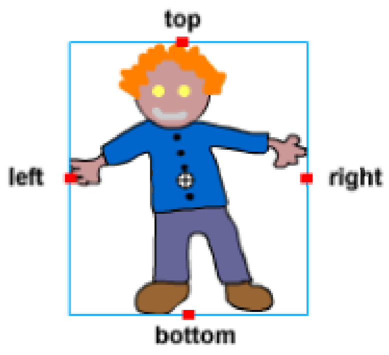
The coded is complicated, so make sure that you've added the right number of brackets in the right place, otherwise it won't work and Flash will give you error messages in the output window.

Test the movie now, and you'll notice that the player is stopped when the top, bottom, left and right edges come into contact with the maze.

The first part of the code looks like this:

```
        //Define Player Boundaries:
        top = this._y - (this._height / 2);
        bottom = this._y + (this._height / 2);
        right = this._x + (this._width / 2);
        left = this._x - (this._width / 2);
```

What this does is to actually create 4 imaginary points on your object, something like this:



These points are stored as variables (`top`, `bottom`, `left` and `right`) so that they will be easy to use later.

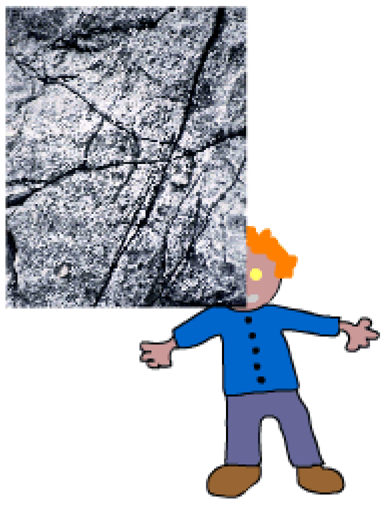
The next part of the code uses those points to check to see whether they are intersecting with the shape of the map object:

```
//Check for Collision with Map:
if ((_root.map.hitTest(left, this._y, true))
|| (_root.map.hitTest(right, this._y, true))
|| (_root.map.hitTest(this._x, top, true))
|| (_root.map.hitTest(this._x, bottom, true)))
{
```

This is one big if statement that checks to see if any of the points are intersecting with the map. The "||" symbol means "Or". The collision detection will be activated if any of these four conditional statements are true.

If any of these points are touching the shape of the "map" object, the player is prevented from moving, using the same blocking technique we used in the previous lesson.

You'll notice, however, that you can 'squeeze' through these points if you try.



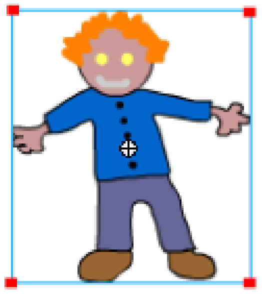
There are a few possible solutions to this problem, and the one you choose will depend on the specific design of your game.

The first possible solution is to define the points on the *corners* of your object, instead of the middle of the edges. This is very easy to do, using the variables that we've already defined. Change your code so that it looks like this (the modified code is highlighted):

```
//Define Player Boundaries:
top = this._y - (this._height / 2);
bottom = this._y + (this._height / 2);
right = this._x + (this._width / 2);
```

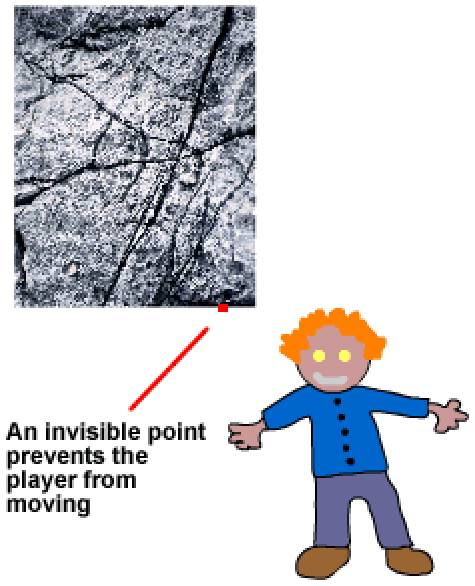
```
left = this._x - (this._width / 2);

//Check for Collision with Map:
if ((_root.map.hitTest(left, top, true))
|| (_root.map.hitTest(right, top, true))
|| (_root.map.hitTest(left, bottom, true))
|| (_root.map.hitTest(right, bottom, true)))
{
```

This checks for a collision between the map and corners of the player's bounding box, like this:



This will work fine as long as the walls of your maze are wider than your player, as they are in the example. If the walls of the maze are thinner, then you'll have to define additional points.

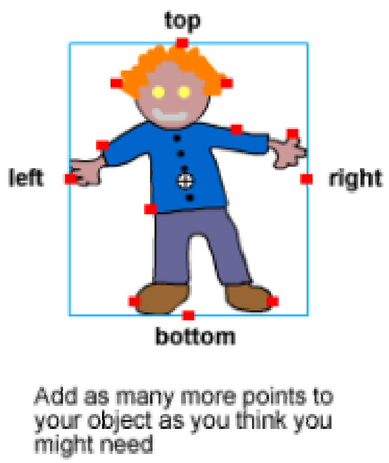
You may also run into another problem, which is the opposite of the one we faced previously:



The player might be blocked by one of the points we've defined, even though it looks, visually, like nothing should prevent it from moving.

The easiest way to solve this problem is to design a squarish shaped player character. That way the corners of the player's bounding box will always be near the corners of the actual image itself. Another solution is to design a maze that doesn't contain protruding edges.

If neither of those solutions will work for your particular game, then you will need to add more points on any part of your object that might come into contact with the edge of your maze:


top
left
right
bottom
Add as many more points to your object as you think you might need

It could be very difficult to define these points mathematically, so you might want to use small, invisible, sub-objects instead. You then need to add more conditional statements to your If Statement to check for a collision with these new points.

Doing accurate collision detection like this takes time, patience and concentration. However, it is the key to making a very professional game. If you put the time into it, your players will thank you for it!

ASM3O - Media Arts

Dodge Game Assignment

You should now have all the skills you need to create a fun computer game. In this assignment, you will create a **Dodge Game.** A dodge game is a game in which you have to **avoid** certain **obstacles** to reach a goal or destination. Obstacles could be enemy spaceships, cars, trees or monsters. You could also include helpful objects (called "pickups") such as medicine kits, shields, or treasure chests.

Your game should include the following:

- A start screen with the **game title** and a button that starts the game.
- Your game screen, which should include the following:
    - A player object which is controlled using the keyboard
    - "Enemies" that the player must avoid
    - A start position for the player
    - A goal that the player should reach
    - A background scene where your game takes place
- You need to include some way of finding out whether the player has won or lost the game. You could do this by creating a score (with a dynamic text field), creating a goal that the player should reach, or by creating a time limit.
- You need an end screen that tells the player whether they have won or lost, and then give them the option to play again.

Project Duration: 8 periods.

Game Ideas:

The following are some ideas that you *could* use to create your game. There are many more that I'm sure you could come up with yourself:

Space Game
The player's spaceship must avoid a field of meteors and enemy spaceships to reach its base or home planet. The player would win the game if he/she reaches the base without being hit. You could make this game more interesting by creating an additional goal that the player would have to fulfill, such as rescuing friends who have been stranded in space, or collecting fuel cells. The player could gain a point for each item collected, and, if he/she returns to the base with them, wins the game.

Street Crossing Game
A very popular early computer game called "Frogger" involved helping a frog cross a street of busy traffic to reach his pond on the other side. You could easily create a similar game with the skills you have.

Skiing Game
Create a game where the player skis down a hill, avoiding trees, in order to reach the finish line at the bottom. To make this game work, the player would only need to move right and left, not up and down. You could create the *illusion* of downward movement by just animating the trees so that they move up the screen. You could add an extra layer of interest to the game by requiring players to ski over markers, and you could give them a point for each marker that they cross.

Maze Dungeon
Create a game where the player needs to move through a dungeon maze. In each room of the dungeon, there might be a treasure guarded by a monster. If the player avoids or kills the monster and is able to pick up the treasure, he/she would win. Decide on how many rooms, treasures and monsters you will have. Also, think of where the dungeon will start and where it will end. You could make this game more complex by allowing players to kill monsters only if they have certain weapons or items. If you want to make a very complex maze, however, you need to understand Advanced Collision Detection using the **shapeFlag** variable, because the simple form of the **hitTest** action does not work well for complex shapes.

Evaluation:

A) Game Play:

- Did you create a detailed/interesting player object? (2 Marks)
- Did you create a detailed/interesting background? (2 Marks)
- Did you create detailed/interesting "enemies"? (2 Marks)
- Was the theme and idea of the game clear and well developed? (2 Marks)
- Was the game fun to play (not too easy, not too hard)? (2 Marks)?

B) Technical Features:

- Did you have a proper title screen with a "Start" button? (2 Marks)
- Did have a proper finish screen that gave the player the option of playing again? (2 Marks)
- Did you use the **hitTest** action properly? (2 Marks)
- Was your game able to tell the player whether he/she won or lost? (2 Marks)?

- Did you use some sort of scoring system with a Dynamic Text Field?

Bonus: Games are always more fun to play if there is a personal element. Can you think of a creative use for *text input* (1 Mark)?
TOTAL: /20

Additional Resources

The following are some on-line listings of additional resources that you might want to use in your games.

Images

"Dingbats" are fonts that use pictures instead of letters. There are thousands of different dingbat fonts available on the internet, and many of these contain images that are perfect as characters or objects in computer games. You can download dingbat fonts here:

```
http://www.dingbatpages.com/
http://www.fontfreak.com/
```

To use Dingbats in Flash:

- copy the font into the "Font" folder in the C: drive of your computer.
- Select the font in Flash, and, using the text tool, find the image that you want to use by pressing the correct key.
- Select the image with the arrow tool.
- Open the **modify** menu and choose **break apart**
- You now have a basic vector graphic that you can use for your game.

And what about 3D graphics? Don't forget, you can create all your images in Bryce, import them as bitmap graphics into Flash, and create a very impressive looking 3D-style game.

Sound

You are not required to use sound in this project, but if you want to, you can find music loops and sound effects from these sites:

```
http://www.partnersinrhyme.com/
http://www.flashkit.com/soundfx/index.shtml
http://www.basementarcade.com/arcade/sounds/sounds.html
http://www.vgmusic.com/
```

Game Creation Tutorials:

If you feel quite confident in the techniques you have learnt so far, you can learn how to create scrolling backgrounds, independently moving enemies and bullets in this very easy-to-follow tutorial. The tutorial is in 3 parts - start with the first one:

```
1) http://www.flashkit.com/tutorials/Games/Building-David_Do-598/index.shtml
2) http://www.flashkit.com/tutorials/Games/Building-David_Do-610/index.shtml
3) http://www.flashkit.com/tutorials/Games/Building-David_Do-611/index.shtml
```

For a *very advanced* tutorial on how to create a 3D space game in Flash, visit this link:

http://www.flashkit.com/tutorials/Games/Programm-Ian_Rose-632/index.shtml

Have fun!

Part 4:
Advanced Game Design

Advanced Game Design

1) Advanced Keyboard Control – Natural Motion

You might have noticed that the objects you control with your keyboard start moving suddenly, stop moving suddenly, and move around the screen in a jittery way. In real life, things don't move like that. That's because, in real life, objects have *mass* and *inertia*. It takes time for them to pick up speed, and takes time for them to slow down. In the following exercises you will learn how to create objects that behave like they have mass. This is basis for creating very compelling and fun games.

It's All in the Variables!

Very conveniently, all we need to do to simulate natural motion is to create a few variables inside our objects that store certain imaginary physical properties. Here are the basic physical properties that we'll need to use:

- Horizontal Speed (`xSpeed`)
- Vertical Speed (`ySpeed`)
- Friction
- Acceleration
- Speed Limit (to make sure that the object doesn't move faster than the laws of nature allow.)
- … and maybe Gravity

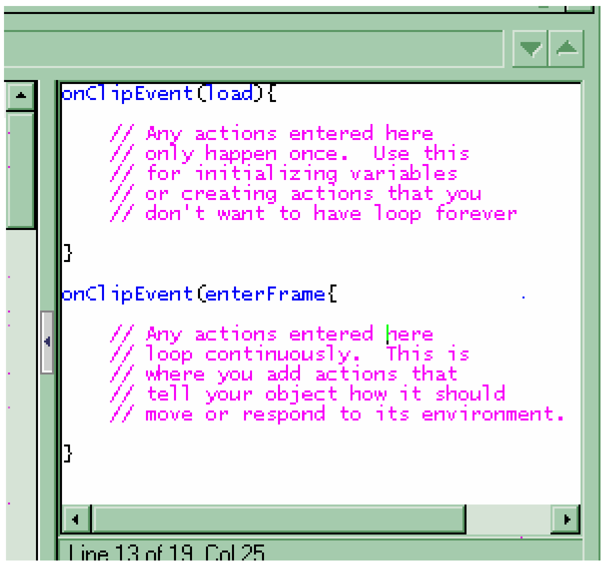
All we need to do is initialize these variables, apply some *very simple* calculations, and we'll start creating objects that move as though they have mass.

OnClipEvent(load) - A New Action

OnClipEvent(load) is an Movie Clip Action that is used to initialize variables inside objects. It is almost always used along with **onClipEvent(enterFrame)**. The two actions are usually used together.

The difference between them is simple. Any actions inside **onClipEvent(load)** only activate once. Actions inside **onClipEvent(enterFrame)** loop forever.

Usually you use **onClipEvent(load)** first to initialize your variables, then you use **onClipEvent(enterFrame)** to move and control the object. Here is what your Object Actions should look like when you use the two together:



```
onClipEvent(load){

    // Any actions entered here
    // only happen once.  Use this
    // for initializing variables
    // or creating actions that you
    // don't want to have loop forever

}

onClipEvent(enterFrame{

    // Any actions entered here
    // loop continuously.  This is
    // where you add actions that
    // tell your object how it should
    // move or respond to its environment.

}
```

Line 13 of 19, Col 25

(By the way, a **double forward slash: //** tells the computer to ignore any information that comes after it on the same line. It is used for writing notes to yourself about what your ActionScript does. It is also for temporarily disabling actions so that you can see what happens without them - rather than deleting them.)

So, let's create some variables. Create an object, and enter the following in its object actions:

```
onClipEvent(load) {

     xSpeed = 0;
     ySpeed = 0;
     acceleration = 0.7
     friction = 0.97;
     speedLimit = 10;

}
```

If you want to change the way your object moves, all you need to do is change the values for **acceleration, friction** and **speedLimit**. (for a realistic effect, a value between 0.88 and 0.99 works well for **friction**.)
Now that we have our variables, all we need to do is make our object move.

Make it Move!

We're going to use the arrow keys to make our object move. Instead of the arrow keys changing the _x and _y position of the object, they are now going to change its **xSpeed** and **ySpeed**, and add our **acceleration** value every time they are pressed. The 2 lines at the very end of the list of actions update the object's position based on its speed. Also, we are going to make sure that the object only increases its speed if it is within the speed limit (this prevents the object from flying away at a fantastic speed):

```
onClipEvent (enterFrame) {

     //Right Arrow
     if (key.isDown(39) && xSpeed < speedLimit) {
          xSpeed = xSpeed + acceleration;
     }

     //Left Arrow
     if (key.isDown(37) && xSpeed > -speedLimit) {
          xSpeed = xSpeed - acceleration;
     }

     //Down Arrow
     if (key.isDown(40) && ySpeed < speedLimit) {
          ySpeed = ySpeed + acceleration;
     }

     //Up Arrow
     if (key.isDown(38) && ySpeed > -speedLimit) {
          ySpeed = ySpeed - acceleration;
     }

     // Reduce the object's speed if no keys are pressed:
     xSpeed = xSpeed*friction;
     ySpeed = ySpeed*friction;

     //Move the Object:
     this._x = this._x + xSpeed;
     this._y = this._y + ySpeed;
}
```

Test your movie and see what happens!

Creating a "Speed Trap"

If you watch your object carefully as it slows down, you'll notice that it sometimes doesn't come to a complete stop. This is because the xSpeed and ySpeed continue to reduce their values even after they reach zero. What you need to do is force your object to stop completely at zero. You can do this by adding the following code just before the end of your **onClipEvent(enterFrame)** action:

```
     if (xSpeed < 0.1 && xSpeed > -0.1) {
          xSpeed = 0;
          }

     if (ySpeed < 0.1 && ySpeed > -0.1) {
          ySpeed = 0;
          }
```

Now your object will stop completely.

Exercise 1: Add Some Gravity

To add gravity to your object, all you need to do is create a new variable, called, surprisingly enough "gravity" and then tell gravity to take control of the object if it is not moving up.

First, you need to add your **gravity** variable in the **onCliptEvent(load)** action. Just add the line in **bold text**:

```
    onClipEvent (load) {
         // variable declarations
         xSpeed = 0;
         ySpeed = 0;
         acceleration = 0.7
         gravity = 0.9;      ←———— Add this
         friction = 0.97;
         speedLimit = 10;
    }
```

Next, you need to add an "Else" statement to the end of your first 4 "If" statements in the **onClipEvent(enterFrame)** action. This tells Flash that if the Up Arrow is not being pressed, it should add our gravity variable to the object's ySpeed. Just add the lines in **bold text**:

```
onClipEvent (enterFrame) {

     //Right Arrow
     if (key.isDown(39) && xSpeed < speedLimit) {
          xSpeed = xSpeed + acceleration;
     }

     //Left Arrow
     if (key.isDown(37) && xSpeed > -speedLimit) {
          xSpeed = xSpeed - acceleration;
     }

     //Down Arrow
     if (key.isDown(40) && ySpeed < speedLimit) {
          ySpeed = ySpeed + acceleration;
     }

     //Up Arrow
     if (key.isDown(38) && ySpeed > -speedLimit) {
          ySpeed = ySpeed - acceleration;
     }
     else{
          ySpeed = ySpeed + gravity;      ←———— Add this
```

```
    }
```

… the rest of the ActionScript code should stay the same.

Try changing the value of the gravity variable and see what kind of effect you get.

You could use this keyboard control to create a game in which a player might have to navigate a helicopter or spaceship over rocky terrain. The game "Lunar Lander" is a good example. Or, you could use it to create a Flash version of the classic game "Joust", where a player controls a giant flying bird. Every time the player presses the Up Arrow, the player's bird flaps its wings and moves up.

I'm sure you can think of many more clever uses!

Advanced Game Design
2) Objects that Move by Themselves

You might not have realized this, but all the techniques that you've been learning over the past few lessons can be applied to *any* objects in your games - not just the player. What this means is that you can create objects that move *by themselves* (without you having to animate them) and that react to the game environment in an unpredictable way. The following exercises will show you how:

Exercise 1: Simple One-Direction Movement

1. Draw a simple circle on the stage and turn it into a Movie Clip Symbol.

2. Open the circle's Movie Clip actions, and add the following ActionScript:

```
onClipEvent(enterFrame){
     this._x = this._x + 1;
}
```

3. Test your movie. What happens? Do you know why it happens?

4. Next, add the **bold text** below:

```
onClipEvent(enterFrame){
     this._x = this._x + 1;
     if (this._x > 550){
          this._x = 0;
     }
}
```

5. Test your movie. What happens when the object reaches the right edge of the screen? Do you know why that happens? You should by now!

What you have just created could be used to create asteroids that fly towards the player or obstacles on a road. If you changed the "_x" to a "_y" you could create raindrops, bombs or trees for a skiing game.

Random Numbers

Very often in games you don't want the player to know where the enemies or obstacles will appear next. You want their movement or appearance to be

random. To create random movement, you first need to create random numbers. Luckily, this is quite easy to do, because Flash provides us with a **Math.random()** action. This is what the random action looks like:

```
    Math.random();
```

What **Math.random()** does is create a random number between 0 and 1. Have a look at the following example:

```
    myRandomNumber = Math.random();
```

In this example, "`myRandomNumber`" would become any number between 0 and 1. It could be "0", it could be "1", but it could also easily be "0.4543245" or "0.76453256" or "0.6356435". We never know - it's random!

Well, you might ask, what good is a number between 0 and 1? Usually, not much. So, you need to multiply **Math.random()** by another number to create a number that will be larger than 1. Have a look at these examples:

- `Math.random()*100;` - creates a number between 0 and 100
- `Math.random()*25;` - creates a number between 0 and 25
- `Math.random()*376;` - creates a number between 0 and 376

All these numbers, however, would produce **decimal values**. That means the numbers that you end up with might look like this:

```
    34.908323
    243.098
    23.88890988
```

Sometimes that's Ok, but sometimes you only want **whole numbers**. To create random whole numbers, you need to put **Math.random()** inside another action called **Math.round**, like this:

```
    Math.round(Math.random()*100);
```

This would create produce a **whole number** between 0 and 100, like this:

```
    84
```

Math.round() rounds off the random number so that there are no decimals.

Ok, now what if you wanted a random number that was within say, 10 and 20? You would do it like this:

```
    Math.round(Math.random()*10)+10;
```

First, this would create a random number between 0 and 10. But then we **add** an extra 10 to it. So the **final number** that we end up with would be between 10 and 20. This might seem a little confusing, but you'll get used to it with practice.
Exercise 2: Creating Random Motion

To create objects that move randomly, all you need to do is assign random numbers to their X and Y positions or their X and Y Speeds. In the circle movie clip that you created earlier, change the object actions so that they look like this:

```
onClipEvent(enterFrame){

     this._x = this._x + Math.random()*10-5;
     this._y = this._y + Math.random()*10-5;

}
```

Test the movie. You've created a completely randomly moving object.

"Math.random()*10 - 5" creates a random number between 5 and -5, and alters the object's _x and _y position accordingly. Now, lets just set an initial **random speed** for the object. Enter the following in the circle's Movie Clip Actions so that it replaces your previous code:

```
onClipEvent(load){

     //Initialize the Speed Variables
     xSpeed = (Math.random()*10-5);
     ySpeed = (Math.random()*10-5);

}

onClipEvent(enterFrame){

     //Move the Object:
     this._x = this._x + xSpeed;
     this._y = this._y + ySpeed;

}
```

Test your movie 3 or 4 times. You should notice that the object moves in a new random direction every time.

Next, **make 20 copies** of the circle on the stage. Do this by copying and pasting the instance of the circle you currently have on the stage, not by dragging a new copy from the library. By copying and pasting the current instance of the circle, all of its ActionScript code will be copied along with it.

Test the movie… what happens? You should notice that each circle has its own behaviour - no two are alike.

Exercise 3: Make Them Bounce

Now, let’s see if we can make our random objects bounce off the edges of the screen.

First, delete all the circles except one.

Add the following **bold text** code inside the circle’s **onClipEvent(enterFrame)** action:

```
onClipEvent(load){

     //Initialize the Speed Variables
     xSpeed = (Math.random()*10-5);
     ySpeed = (Math.random()*10-5);
}

onClipEvent(enterFrame){


     //Check Screen Boundaries
     if(this._x > 550){
          xSpeed = (-xSpeed);
     }
     if(this._x < 0){
          xSpeed = (-xSpeed);
     }
     if(this._y > 400){
          ySpeed = (-ySpeed);
     }
     if(this._y < 0){
          ySpeed = (-ySpeed);
     }

     //Initialize the Speed Variables
     this._x = this._x + xSpeed;
     this._y = this._y + ySpeed;

}
```

Now, make 15 copies of your object (you can do this quickly by holding down ALT and clicking and dragging the original object.)

Test your movie… what happens?

How It Works:

The logic behind this is very simple. When the object reaches the edges of the stage, it is told to **reverse** its x or y speed, such as:

```
      xSpeed = (-xSpeed);
```

This creates the illusion of the object bouncing. The "-" symbol simply sets the speed to the opposite of what it currently is, very easily reversing the object's direction. And, because every object has its own randomly generated speed, every object bounces back in a unique way. Watch the circles for a few minutes: they almost look alive!

There are literally thousands of ways that you can use random numbers in your games, and this is just a beginning. Once you start experimenting, you'll be amazed at what you can do.

Adding Gravity

You can easily add gravity by creating a new `gravity` variable and combining it with the calculation that figures out the `ySpeed` of your object. Add the following highlighted lines to your code:

```
onClipEvent (load) {
      //Initialize Variables
      xSpeed = (Math.random() * 10 - 5);
      ySpeed = (Math.random() * 10 - 5);
      gravity = 0.94;
}

onClipEvent (enterFrame) {
      //Check for Screen Boundaries:
      if (this._x > 550) {
            xSpeed = (-xSpeed);
      }
      if (this._x < 0) {
            xSpeed = (-xSpeed);
      }
      if (this._y > 400) {
            ySpeed = (-ySpeed);
      }
      if (this._y < 0) {
            ySpeed = (-ySpeed);
      }

      //Add gravity to the ySpeed:
      ySpeed = ySpeed + gravity;
```

```
        //Move the Object:
        this._x = this._x + xSpeed;
        this._y = this._y + ySpeed;
}
```

You'll notice a small problem when you test the movie, however. The object doesn't come to a complete rest when it hits the bottom of the stage, and continues to be dragged down. We can prevent this from happening by making the following modification to the If Statement that checks whether the object is at the bottom of the screen:

```
        if (this._y  > 400) {
              ySpeed = (-ySpeed);
              this._y = 400;
        }
```

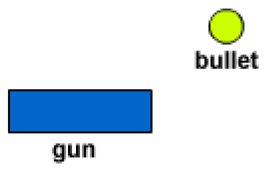
The additional line (`this._y = 400;`) forces the object back onto the stage to Y position number 400. This is the point at which it is no longer being affected by the If Statement. It is therefore no longer "trapped."

Advanced Game Design
3) Firing Bullets

One of the most important things to be able to do in a computer game is to fire bullets. This is quite complicated and there are a lot of new techniques here, so you must follow the instructions VERY CAREFULLY:

A: Create Your Objects

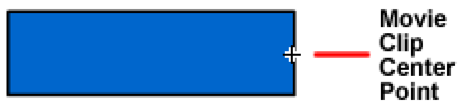
Create 2 objects on the stage: One called "`gun`" and another called "`bullet`".



Don't forget to name these in the properties panel!

You can position the bullet anywhere on the stage for now, but later you will be moving it off-stage.

The center point of the gun object will be where your bullets will be firing from, so make sure that it is at the front of your gun.



You can use the Transform tool to reposition the center point if you need to.

B: Program the Gun

Open the gun's Movie Clip Actions and add the following code:

```
onClipEvent(load){
     shotTimer = 0;
     shotCount = 0;
     shotInterval = 5;
     }

onClipEvent (enterFrame) {

     //Fire the Bullet:
     if ((Key.isDown(Key.SPACE)) && (shotTimer <= 0)) {
          shotCount = shotCount + 1;
          duplicateMovieClip(_root.bullet, "bullet" +
          shotCount, (shotCount % 100) + 1100);
          shotTimer = shotInterval;
     }

     // Decrease the value of shotTimer if the Space Key
     // is not being pressed:
     shotTimer = shotTimer - 1;
}
```

Before our gun will work, we need to program the bullet as well.

How It Works: The Gun

The `onClipEvent(load)` action initializes the variables that we will be using:

```
onClipEvent(load){
     shotTimer = 0;
     shotCount = 0;
     shotInterval = 5;
     }
```

`shotTimer` counts the time that has elapsed between shots. `shotCount` counts the number of bullets being fired. `shotInterval` sets how much time, in frames, should elapse before the gun is permitted to fire again. If you want the gun to fire more slowly, set this to a larger number, like 10.

The `onClipEvent(enterFrame)` action is more complicated.

```
if ((Key.isDown(Key.SPACE)) && (shotTimer <= 0)) {
          shotCount = shotCount + 1;
          duplicateMovieClip(_root.bullet, "bullet" +
          shotCount, (shotCount % 100) + 1100);
          shotTimer = shotInterval;
     }
```

The If Statement checks to see if the Space key is being pressed down and the `shotTimer` variable is less than or equal to zero. If both of these are true, the gun is permitted to fire and the following statements execute:

```
shotCount = shotCount + 1;
```

- Counts the current bullet being fired.

```
duplicateMovieClip(_root.bullet, "bullet" +
     shotCount, (shotCount % 100) + 1100);
```

- Uses the `duplicateMovieClip` action to makes a copy of the `bullet` movie clip on the stage and gives it a unique name, based on the number of shots fired. The first shot will be called "`bullet1`", the second "`bullet2`" and so on. The final section of this line "`(shotCount % 100) + 1100);`" is trick that is used to add the new bullet onto a unique **level**. A **level** in ActionScript is similar to a **layer** in Flash Animation. When you use `duplicateMovieClip` the new object must be on a layer that is not occupied by anything else. To ensure this, we can use the `shotCount` variable and add a number to it, such as 1100, which is sufficiently high that it is unlikely that any other object will occupy the same level.

```
shotTimer = shotInterval;
```

- This line resets the `shotTimer` variable to 5 (or whaterver `shotInterval` was set to) to prevent the player from firing right away.

C: Progam the Bullet

The final step is to program the bullet object. Add the following code to your bullet object:

```
onClipEvent (load) {
     if (_name != "bullet") {
          this._x = _root.gun._x;
          this._y = _root.gun._y;
          xSpeed = 10;
          ySpeed = 0;
     }
}
onClipEvent (enterFrame) {
     if (_name != "bullet") {
          this._x = this._x + xSpeed;
          this._y = this._y + ySpeed;
     }
     //Check for Screen Boundaries
     if ((this._x > 550) || (this._x < 0)  || (this._y <
          0) || (this._y > 400)) {
          removeMovieClip(this);
     }

     //Insert Collision Detection Code Here:
}
```

Test your movie – you should be able to fire the gun. Move the original bullet movie clip off the edge of the stage for the effect to be more realistic.

How It Works: The Bullet

The `onClipEvent(load)` action sets the initial position of each bullet and determines the speed at which it will move:

```
onClipEvent (load) {
     if (_name != "bullet") {
          this._x = _root.gun._x;
          this._y = _root.gun._y;
          xSpeed = 10;
          ySpeed = 0;
     }
}
```

The X and Y position of the bullet are placed at the X and Y center of the gun, which was why it was important to precisely position the gun's center point to the spot at which we want the bullets to emerge.

You may be wondering why the following If Statement was used:

```
if (_name != "bullet")
```

This is essentially saying "*if this object's name is not "bullet" then perform these actions*"

Remember that when we were programming the gun, we used this line

```
duplicateMovieClip(_root.bullet, "bullet" +
     shotCount, (shotCount % 100) + 1100);
```

… to create a duplicate copy of the bullet with a new name, such as "`bullet1`" and "`bullet2`". Our original object, "`bullet`" is itself never being fired from the gun – we're only using it to make copies. That means that we *don't want any of these actions to act on the original bullet, only the copies*.

These actions will only execute if the object's name is not "`bullet`" – our original. They will work only for the copies ("`bullet1`", "`bullet2`" etc.) that we making from it. This works because when we use `duplicateMovieClip`, all of the object's *actions* are duplicated as well as the graphics and animation.

The next block of code moves our duplicated bullets across the stage:

```
onClipEvent (enterFrame) {
     if (_name != "bullet") {
          this._x = this._x + xSpeed;
          this._y = this._y + ySpeed;
     }
```

For now, our `ySpeed` is set to zero, so there won't be any horizontal movement. If you give the `ySpeed` variable a value in the `onClipEvent(load)` section, you should notice the bullets move at an angle. You may not find a use for this right away, but it's important to keep in mind because at some point you will probably want your bullets to move vertically.

The following block removes any bullet that exceeds the stage boundaries:

```
//Check for Screen Boundaries
if ((this._x > 550) || (this._x < 0)  || (this._y <
     0) || (this._y > 400)) {
     removeMovieClip(this);
}
```

The action "`removeMovieClip(this)`" removes the duplicated object from the stage. It is important to remove the bullets when they fly off the edge of the stage because, even if you can't see them, they are still there. As you fire more and more bullets, your game will start to run slower and slower.

More About `removeMovieClip`

You can use **`removeMovieClip`** with any object that you created with **`duplicateMovieClip`**. **`removeMovieClip()`** needs at least one argument: the object that you want to remove.

So, for example, if you wrote:

```
      removeMovieClip(_root.enemy6)
```

…it would remove the "`_root.enemy6`" movie clip.

There is a special argument called "`this`" which is used if you want a movie clip to remove itself:

```
      removeMovieClip(this);
```

This removes the movie clip from the stage.

You can only use **`removeMovieClip()`** if you created the movie clip using **`duplicateMovieClip()`**. It won't work otherwise.

Collision Detection

All your collision detection code should be added right at the end of the bullet's **`onClipEvent(enterFrame)`** action, just before the final brace.

Moving the Gun

To move your gun around the stage, you need to add the keyboard control actions that you were using in the previous lessons. The following code demonstrates how you might do this:

```
onClipEvent (load) {
      acceleration = 0.7;
      friction = 0.97;
      speedLimit = 10;
      shotTimer = 0;
      shotCount = 0;
      shotInterval = 5;
}
onClipEvent (enterFrame) {

      //Keyboard Control:
      //Right Arrow
      if (key.isDown(39) && xSpeed < speedLimit) {
            xSpeed = xSpeed + acceleration;
      }
      //Left Arrow
      if (key.isDown(37) && xSpeed > -speedLimit) {
            xSpeed = xSpeed - acceleration;
      }
      //Down Arrow
      if (key.isDown(40) && ySpeed < speedLimit) {
           ySpeed = ySpeed + acceleration;
      }
      //Up Arrow
      if (key.isDown(38) && ySpeed > -speedLimit) {
           ySpeed = ySpeed - acceleration;
      }

      // Reduce the object's speed if no keys are pressed:
      xSpeed = xSpeed * friction;
      ySpeed = ySpeed * friction;
      //Move the Object:
      this._x = this._x + xSpeed;
      this._y = this._y + ySpeed;

      //Speed Trap
      if (xSpeed < 0.1 && xSpeed > -0.1) {
           xSpeed = 0;
      }
      if (ySpeed < 0.1 && ySpeed > -0.1) {
           ySpeed = 0;
      }

      //Fire the Bullet
      if ((Key.isDown(Key.SPACE)) && (shotTimer <= 0)) {
           shotCount = shotCount + 1;
           duplicateMovieClip(_root.bullet, "bullet" +
           shotCount, (shotCount % 100) + 1100);
           shotTimer = shotInterval;
      }

      // Decrease the value of shotTimer if the Space Key
      // is not being pressed
      shotTimer = shotTimer - 1;
}
```

This should be enough to use as the basis for a space-shooter game.

Exercise 1: Add an Enemy.

1. Create a movie clip called "`enemy`" and place it on the stage.
2. Add some ActionScript code to your bullet movie clip to check to see if it is hitting the enemy.
3. If it is, cause your enemy to explode

You should already know how to do this. All you need to do is add a **hitTest** action inside an If Statement near the end of the bullet's **onClipEvent(enterFrame)** action.

Exercise 2: Firing in Different Directions.

The next thing you need to do is to make your bullet fire in different directions, depending on which arrow keys the player is pressing. If the player presses the up arrow key, the bullet should fire up. If the player presses the left arrow key, it should fire left.

To do this, you first need to remove these lines from the bullet's **onClipEvent(enterFrame)** action:

```
this._x = this._x + xSpeed;
this._y = this._y + ySpeed;
```

In their place, you need to add 4 more If Statements to check to see which arrow keys is being pressed. Then, you need to move the bullet in the correct direction.

The following is an example of the code you will need to move the bullet to the right:

```
if (key.isDown(39)) {
     this._x = this._x + xSpeed;
}
```

This code moves the bullet up:

```
if (key.isDown(38)) {
     this._y = this._y - ySpeed;
}
```

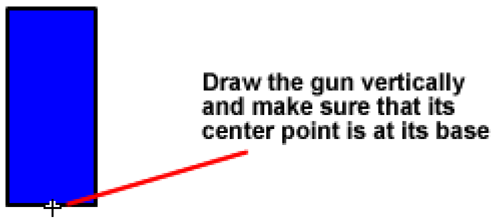
Can you figure out the ActionScript code for left and down???

Acknowledgements: The ActionScript code for the gun and bullet has been adapted with minor modifications from "Flash Games Studio" by Friends of Ed, 2001.

Advanced Game Design
4) Creating a Rotating Gun Turret with Trigonometry

These instructions show you how to create a gun turret that rotates and fires bullets. This exercise uses ***trigonometry*** *to calculate the correct position of the bullet. Unfortunately, using trigonometry is the only way this can be done. Don't let this scare you however: all you need to do is copy the ActionScript code correctly and use it in your game*

Create a Movie Clip Called "`gun`". Draw the gun vertically and make sure that the center point of the gun is positioned at the bottom axis around which you want it to rotate.



This is the opposite of what you did in the previous lesson. In this exercise, the bullets will be emerging from the top end of the gun.

Enter the following Movie Clip Actions. (The new code is highlighted in bold):

```
onClipEvent(load){
   shotTimer = 0;
   shotCount = 0;
   shotInterval = 5;
   }

onClipEvent (enterFrame) {

   //Rotate The Gun
   if (Key.isDown(Key.RIGHT)){
         this._rotation = this._rotation +3;
   }
   if (Key.isDown(Key.LEFT)){
         this._rotation = this._rotation -3;
   }

   //Fire the Bullet
   if ((Key.isDown(Key.SPACE)) && (shotTimer <= 0)) {
         shotCount = shotCount + 1;
```

```
            duplicateMovieClip(_root.bullet, "bullet" +
            shotCount, (shotCount % 100) + 1100);
            shotTimer = shotInterval;
      }
      // Decrease the value of shotTimer if the Space Key is
   not being pressed
      shotTimer = shotTimer - 1;
   }
```

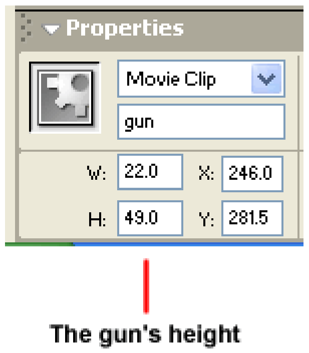
Test the movie now. You will be able to rotate the gun by pressing the right and left arrow keys. The next step is to program the bullet.

Next, create a movie Clip called "`bullet`" and enter the following in its Movie Clip Actions (the new code is highlighted):

```
onClipEvent(load){

     if (_name != "bullet"){

          gunHeight = 49;
          //Find out the height of your gun, in pixels,
          //in the Properties panel and enter it above

          gunPoint = _root.gun._rotation;
          angle = (gunPoint/360) * 2 * Math.PI;
          xComponent = gunHeight * Math.sin(angle);
          yComponent = gunHeight * Math.cos(angle);
          this._x = xComponent + _root.gun._x;
          this._y = -yComponent + _root.gun._y;
          xSpeed = (xComponent/gunHeight) * 10;
          ySpeed = (yComponent/gunHeight) * 10;
     }
}

onClipEvent (enterFrame){
     if (_name != "bullet"){
          this._x = this._x + xSpeed;
          this._y = this._y + -ySpeed;
     }

     //Check for Screen Boundaries
     if ((this._x > 550) || (this._x < 0)  || (this._y <
          0) || (this._y > 400)) {
          removeMovieClip(this);
     }

     //Insert Collision Detection Code Here:
}
```

Move the original bullet Movie Clip off the stage so that it is not visible in your game, and test the movie.

The most important thing about this code is to make sure that you find out the height of the gun object in the Properties panel:



Copy that number into the line that initializes the `gunHeight` variable:

```
gunHeight = 49;
```

This number is used to calculate the exact position and angle of each of the bullets that are being fired, which is what the following lines of code do:

```
     angle = (gunPoint/360) * 2 * Math.PI;
     xComponent = gunHeight * Math.sin(angle);
     yComponent = gunHeight * Math.cos(angle);
     this._x = xComponent + _root.gun._x;
     this._y = -yComponent + _root.gun._y;
     xSpeed = (xComponent/gunHeight) * 10;
     ySpeed = (yComponent/gunHeight) * 10;
```

This code applies basic trigonometry to work out the direction that each bullet should move in, based on the gun's angle of rotation. My advice to you is: close your eyes, copy and paste! If you need to know exactly how it works, consult a math teacher or textbook. The most important thing is not so much that you know how it works, but that you can *use* it to make creative and original games.

Make It Move

If you want to move the gun around the stage, in the same direction in which it is pointing, you need to apply the same trigonometry calculations that you used with the bullet. Add the following highlighted code to your `gun` object:

```
onClipEvent(load){
     shotTimer = 0;
     shotCount = 0;
     shotInterval = 5;
     }

onClipEvent (enterFrame) {

     //Rotate The Gun
     if (Key.isDown(Key.RIGHT)){
          this._rotation = this._rotation +3;
     }
     if (Key.isDown(Key.LEFT)){
          this._rotation = this._rotation -3;
     }
     //Up Arrow
     if (key.isDown(Key.UP)) {
          gunPoint = _root.gun._rotation;
          angle = (gunPoint/360) * 2 * Math.PI;
          xComponent = Math.sin(angle);
          yComponent = Math.cos(angle);
          this._x = xComponent + _root.gun._x;
          this._y = -yComponent + _root.gun._y;
     }

     //Fire the Bullet
     if ((Key.isDown(Key.SPACE)) && (shotTimer <= 0)) {
          shotCount = shotCount + 1;
          duplicateMovieClip(_root.bullet, "bullet" +
          shotCount, (shotCount % 100) + 1100);
          shotTimer = shotInterval;
     }

     // Decrease the value of shotTimer if the Space Key
     // is not being pressed
     shotTimer = shotTimer - 1;
}
```

Can you find a way of fine-tuning your movie so that the gun rotates precisely around its center but the bullets still fire from the correct place? Hint: you'll need to make a small modification to your gun object and one change in the bullet's actions. It's not difficult – try it!

Happy Shooting!!!

Acknowledgements: The ActionScript code for the gun and bullet has been adapted with minor modifications from “Flash Games Studio” by Friends of Ed, 2001.

Action Game Assignment

In the final assignment of the year, you will be given the choice as to whether you want to create a new game, or create a new level, with advanced features, for your previous game. Read the assignment details carefully:

A: New Action Game

Create a game that uses the three new techniques that you have learnt: **random motion**, **advanced keyboard control**, and **bullets**

Your game should include the following:

- A start screen with the **game title** and a button that starts the game.
- Your game screen, which should include the following:
    - A player object which is controlled using the keyboard
    - "Enemies" that the player must avoid or destroy
    - A start position for the player
    - Possibly a goal that the player should reach (this is optional)
    - A background scene where your game takes place
- You need to a **scoring system**, so some other way of finding out whether the player has won or lost.
- You need an **end screen** that tells the player whether they have won or lost, and then give them the option to play again.
- You need to give the player (or the enemy) the ability to **fire bullets**.
- You must include **some form of randomization** in your game. This could be random placement of objects
- Your player object should use some **advanced keyboard control methods**, such as friction, acceleration or gravity.

Please feel free to use graphics that you found on the internet or created for another project. In this project, the most important thing is that you use the new techniques properly, not necessarily create the best graphics that you can. Use dingbat and clipart graphics wherever possible to save on time.

Game Ideas:

Here are some classic game ideas that you *could* adapt as the basis for your game:

• Lunar Lander. The player carefully controls a spaceship over treacherous terrain to collect fuel cells, rescue crewmembers, or land in a narrow landing pad.
• Space Invaders: Fight off attacking aliens invading from the skies above.
• Joust: Fly a giant bird between floating platforms and eliminate your enemies.

• Mario Brothers: Run and jump your way around an imaginary world, collecting items and avoiding enemies. This is an *advanced* project: please let me know if you decide to try it, and I will show you how to make your player object "jump."
• Breakout (DX Ball): This is not difficult to make with the skills that you currently have. Let me know if you want to make this game, and I will show you how to make your objects bounce.

By the way, it is *very easy* to create a terrain, map or maze for your player to move through… can you figure out how???

New Action Game Evaluation:

A) Game Play:

- Was the theme and idea of the game clear and well developed? (2 Marks)
- Was the game fun to play (not too easy, not too hard)? (2 Marks)?

B) Technical Features:

- Did you have a proper title screen with a "Start" button and a proper finish screen that gave the player the option of playing again? (2 Marks)
- Did you use the **hitTest** action properly? (2 Marks)
- Did you use advanced keyboard control techniques? (2 Marks)
- Did you use randomization somewhere? (2 Marks)?
- Did you use some sort of scoring system with a Dynamic Text Field? (2 Marks)
- Did you use bullet firing somewhere? (2 Marks)
- Can the player win and lose? (2 Marks)

TOTAL: /18
Project Duration: 8 periods.

B: A Second Level

Create a “second level” for your game that uses ***any two*** of the three new techniques that you have learnt: **random motion**, **advanced keyboard control**, and **bullets.** You don’t need to use all of them. You will, however, need to add sound.

Your new level should include the following:

- Any two of the three new techniques you have learnt: random motion, advanced keyboard control or bullets.
- You must add at least **4 sources of sound**. These could include a musical soundtrack, background sounds or sound effects (such as a laser firing)
- Your second level must be **significantly different** in game-play style from your first level. The objectives and strategies should be different: it should almost feel like a new game - a new challenge.
- Your new level must contribute to the previous level. That means that you probably shouldn’t change the way the player character looks, and your new graphics should be in the same style.
- You must make sure that the new level works properly with the previous level.
- You need to create new background, enemy and obstacle graphics for you new level.

Second Level Evaluation:

A) Game Play:

- Was the theme and idea of the second level a well-developed extension of the first level? (2 Marks)
- Was the second level different enough from the first level to make it a new challenge? (2 Marks)?
- Did you create new background graphics for your second level that enhance the style of the first level? (2 Marks)?
- Did you create new enemy/obstacle graphics for your new level (2 Marks)?
- Was the new level “fun to play” - not too easy, not too difficult? (2 Marks)?

B) Technical Features:

- Did you use at least 4 sources of sound (2 Marks)

- Did you use **any two** of the new techniques: advanced keyboard control, bullet firing or randomization? (4 Marks)
- Did you properly integrate the new level with your scoring system and any other variables you might have used? (2 Marks)

TOTAL: /18
Project Duration: 8 periods.