exercise 9.1

The two functions in the code define the methods by which you are extending the *MovieClip* class. The custom sub-class *FadeColour* defines the *onPress* and *onRelease* event handlers which when pressed or released set the transparency of the movieclip to *30* and *100* respectively. The *this.* statement means that these functions will be invoked upon whatever *MovieClip* object is attached to the script.

4 Save the document as *FadeColour.as* in the *FadeColour* folder.

Step 2 – Assign the class to a movieclip symbol

1 In Flash, choose *File > New*, then select *Flash Document* from the list of file types, and click *OK*.

2 Using the *Oval* tool, draw a circle on the *Stage*.

3 Select the circle, and choose *Modify > Convert to Symbol*.

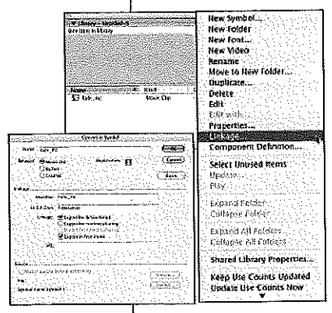
4 In the *Convert to Symbol* dialog box, select *Movie Clip* as the symbol's behaviour, and enter *fade_mc* in the *Name* text box.

5 Select *Advanced* to show the options for *Linkage*, if they aren't already showing.

6 Select the *Export for ActionScript* option, and type *FadeColour* into the *AS 2.0 Class* field. Click *OK*.

7 Save the file as *FadeColour.fla* in the *FadeColour* folder (the same folder that contains the *FadeColour.as* file) and test the movie.

Each time you click the movieclip its transparency should change. When the mouse button is released the transparency should be restored.



Extension

Create a sub-class which defines other properties and methods. For example, you could create a method which moves the movieclip around the stage when the mouse button is clicked. Attach this class to *MovieClip* objects using linkage.

tip **More information on OOP**

Books

Object oriented programming with ActionScript
Branden Hall, Samuel Wan, New Riders Publishing 2003

Flash MX 2004 at your fingertips
Sham Bhangal, Jen deHann, Sybex Inc 2004

Web

www.java.sun.com/docs/books/tutorial/java/concepts/

www.actionscript-toolbox.com

www.actionscript.com

Jigsaw puzzle project

In the following exercises you will create a simple four piece jigsaw puzzle. The aim of this project is to strengthen your understanding of object-oriented and procedural programming with ActionScript. You will construct some external code which defines some methods and extends the built-in *MovieClip* class. You will then attach this code to movieclip instances using the *#include* directive. The completed project can be viewed by opening the supplied file *jigsaw.swf*.

exercise 9.2

Create an external methods script

In this exercise you will create an external ActionScript file which will define some methods. Later in the exercise this code will be attached to elements of the puzzle.

1 Open a new Flash document, save it as *asscript.as* in a new folder called *jigsaw*.

2 In the *Actions* panel, enter the following script.

```
this.onPress = function() {
  if(this.hitTest(_root._xmouse,_root._ymouse)) {
    this.startDrag();
  }
}
```

Continued next page...

exercise 9.2

This first snippet of code defines a method which tests the *x* and *y* coordinates of the mouse when it is clicked and evaluates whether the mouse pointer is positioned over any object on the parent timeline *(_root)* to which this method is attached. If the mouse pointer is over an object which is attached to the script, then that object may be dragged with the mouse while the mouse button is held down.

In the code, *this* identifies an object or movieclip to which the code is attached. Used in conjunction with *#include* a module of code can be attached to any object.

3 Continue to enter the following code in the *Actions* panel below the code you have already entered.

```
this.onRelease = function() {
  stopDrag();
  name = this._name;
  int_name = name.substring(1);
  int_name = parseInt(int_name);
  if(eval(this._droptarget) == _parent["inv_p"+int_name]) {
    _parent.joined_mc["p"+ int_name]._visible = true;
    this._visible = false;
  } else {
      this._x = 100 + Math.random()*180;
      this._y = 200 + Math.random()*200;
  }
}
```

This snippet of code defines a method which, when attached to the puzzle pieces, will allow the selected piece to be dropped when the mouse button is released. When released the puzzle piece will either drop into position on the puzzle board, if correctly targeted, or drop into a random position off the puzzle board if not.

- `name = this._name;` this line of code declares a variable and assigns it the name of the object to which it is attached.
- `int_name = name.substring(1);` declares another variable and assigns it the second character of the value stored in the variable *name*. In the jigsaw puzzle each puzzle piece is given an instance name *p1, p2, p3* or *p4*, so the second character will be either *1, 2, 3* or *4*, it will be stored as a string data type.
- `int_name = parseInt(int_name);` converts the string values *1, 2, 3* or *4* into integer numbers.
- `if(eval(this._droptarget) == _parent["inv_p"+int_name])` evaluates or compares the properties of the object being dragged with the properties of the target object *(inv_p)*.
- `_parent.joined_mc["p"+ int_name]._visible = true;`
  `this._visible = false;`
  If the puzzle piece is dropped over its target, the dropped piece is made invisible while the piece in the finished puzzle is made visible.
- `["p"+ int_name]` refers to an array storing the four puzzle piece movieclips *p1* to *p4*.
- `this._x = 100 + Math.random()*180;`
  `this._y = 200 + Math.random()*200;`
  This snippet of code in the *else* statement is expressed if the puzzle piece is not dropped in the right place on the puzzle board. In this case the puzzle piece is placed at random screen coordinates.

Continued next page...

tip **Character indexing**

The first character in a string is indexed as character 0, the second as character 1 etc.

tip **= and ==**

Watch you don't confuse these two operators.

= is an assignment operator used to assign values to variables.

== is a comparison operator used to evaluate whether the values stored in two variables are equal.

tip **Substring method**

This can also be written as in this example,
`name.substring(2,4);`
If the variable *name* had the value *"Natcoll"* then the above statement would return *"tco"*. It takes all the characters from index 2 through to index 4.

tip **Math.random()**

This built-in method returns a random decimal number between 0 and 1. In the exercise, when the randomly chosen number for x is multiplied by 180 and then 100 is added, the x coordinate will be a value between 100 and 280. Similarly, the y coordinate will have a value between 200 and 400.

exercise 9.2

4 Enter the following code in line 1 of the *Actions* panel (all subsequent code will be moved down one line). Also add another closing curly brace at the very end.

```
onClipEvent(load) {
```

This event handler triggers the actions defined in this script as soon as the object to which this script is attached is loaded into the timeline.

5 Save this file as *asscript.as* in the *jigsaw* folder and close the file.

In the next exercise you will attach the functions you have written here to the jigsaw puzzle pieces. Once functions are attached to objects they are called methods.

exercise 9.3

Assemble the jigsaw

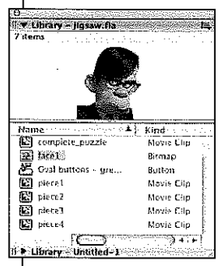
1 Open the file *jigsaw.fla*. In the *Library* you will find all of the elements required to complete this project.

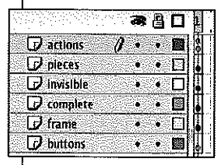
2 In this file you will find one layer named *buttons*. Create four more layers and name them *actions, pieces, invisible* and *complete*.

3 On the *complete* layer, place an instance of the movieclip *completed_puzzle* on to the *Stage*. In the *Properties* panel name this instance *joined_mc*.

4 On the *pieces* layer, place instances of the movieclip symbols *piece 1, piece 2, piece 3* and *piece 4* on to the stage. Give each instance the corresponding instance name *p1, p2, p3* or *p4*.

5 On the *invisible* layer, again place instances of the movieclip symbols *piece 1, piece 2, piece 3* and *piece 4*. Arrange these instances into their correct positions on top of the instance *joined_mc* and give them instance names *inv_p1, inv_p2, inv_p3* and *inv_p4*. For each of these instances select *alpha* from the *Color* option in the *Properties* panel and set the alpha value to *0%*. This will make these instances invisible. The function of these invisible pieces is to act as the targets on to which other objects can be dropped.





exercise 9.4

Attach external ActionScript to objects

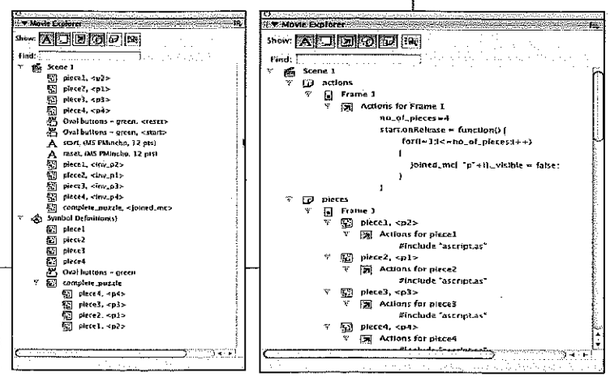
You have already written the ActionScript to control the movement and placement of the puzzle pieces *p1, p2, p3* and *p4*. This was the file saved as *asscript.as*.

1 Open the file *jigsaw.fla* (if it is not already open).

2 Select each instance of the puzzle pieces *p1, p2, p3* and *p4* in turn and in the *Actions* panel attach the following code:

```
#include "asscript.as"
```

#include is a compiler directive which is being used here to attach the module of code saved in the file *asscript.as* to each piece.



The Movie Explorer presents a hierarchial representation of all objects in the movie. Right shows the objects and their hierarchy in the Jigsaw puzzle itself. Far right shows the ActionScript attached to objects and frames.

exercise 9.5

Attach ActionScript for the start button to a frame

This snippet of code is attached to a frame and hides the puzzle pieces on the board when pressed. It creates a function which defines the method (behaviour) of the start button when it is released — the pieces making up the movieclip of the completed puzzle will become invisible.

1 Open the file *jigsaw.fla* (if it is not already open).

2 In frame 1 of the *actions* layer enter the following in the *Actions* panel:

```
var no_of_pieces:Number = 4;
start.onRelease = function() {
    for(i = 1; i <= no_of_pieces; i++) {
      joined_mc["p"+i]._visible = false;
    }
}
```

The *for* loop will complete four iterations, one for each of the puzzle pieces *p1, p2, p3* and *p4*. Each time through it will make the current puzzle piece invisible.

tip Looping backwards

To count backwards from ten to one using a loop you could use something like this:

```
for(i = 10; i > 0; i--)
```

3 Save and test your movie. It should now be fully functional.

project 3

Extension activities

1 Use ActionScript to add the following functionality to the jigsaw puzzle:
- Use the start button to show as well as hide the completed puzzle.
- Make the reset button functional.
- Have puzzle pieces return to their original position when not correctly placed.
- Change the image on the puzzle.

2 Use the Flash online *Help* menu to locate all built-in classes and their properties, methods and event handlers.

3 Extend the *MovieClip* class to create a library of custom visual effects (sub-classes of the *MovieClip* class).

Other examples

Look at the jigsaw puzzle sample provided by Macromedia. The file can be found inside the Flash 2004 folder on your hard drive, it is called *mypuzzle.fla*. What built-in classes have been used in the production of this application? Have any custom classes been created for use by this application?

OOP modular nature

The modular nature of OOP enables you to quickly and easily mix, match and extend classes, objects, properties and/or methods. Once a script is written it can be attached to any suitable object and act to define/redefine its properties and/or methods, extending the class to which the object belongs.

Custom classes can also be created to define the properties and objects to suit any specific application. In the jigsaw puzzle project you attached a module of ActionScript *(asscript.as)* to the puzzle pieces (instances of a movieclip object). The script was attached using the *#include* command and acts to determine the methods (drag, drop etc) of each puzzle piece. The properties of the puzzle pieces can be edited independently of their methods. Try it for yourself.

In summary:

Class = MovieClip
Properties = all the properties of the class
Methods = all the methods of the class

Class = MovieClip
Object = complete_puzzle
Properties = all properties of complete_puzzle
Methods = no methods called

Objects = piece 1, piece 2, piece 3, piece 4
Instances = p1, p2, p3, p4
Properties = all properties of complete_puzzle, all properties of the instances
Methods = all of the methods of the class, no methods defined on instances

+

External script
Properties = none
Methods = all methods defined in this file

Objects = piece 1, piece 2, piece 3, piece 4
Instances = p1, p2, p3, p4
Properties = all properties of complete_puzzle, all properties of the instance
Methods = all methods defined in the external AS file

For any of the puzzle pieces enter *Symbol edit mode* and use the drawing tools to change their appearance. The properties of this instance of the movieclip will be changed but its methods will not.

This project also illustrates the event driven nature of ActionScript and the non-linear nature of OOP. Puzzle pieces can be selected in any order the user determines. It is an event (the pressing of the mouse button) that determines when the code is actioned. Another event (if the puzzle piece has found its target or not) determines how the movieclip object will respond once dragged and dropped onto the puzzle board.

In the second exercise you extended the methods of the *MovieClip* class by defining a new method in an external script and attached it to a *MovieClip* object using linkage. The external script created is available to any *MovieClip* object.

Applications of classes and sub-classes

Using these techniques you could create classes which define a host of different objects and their respective specific properties and methods. For example, visual effects could be designed and implemented as a class, printing properties could be defined in a class which is expressed when a user prints from a *.swf* file.

Classes can be used to represent a collection of similar objects. For example, you could create a class of *Animals* which contains all of the properties and methods common to all animals. You could then create sub-classes of the animal group which are representative of specific types of animals. For example, *Land dwelling animals* and *Ocean dwelling animals.* All the properties and methods common to all animals are defined in the *Animal* class. Properties and methods specific to the type of animal are defined in the sub-classes.

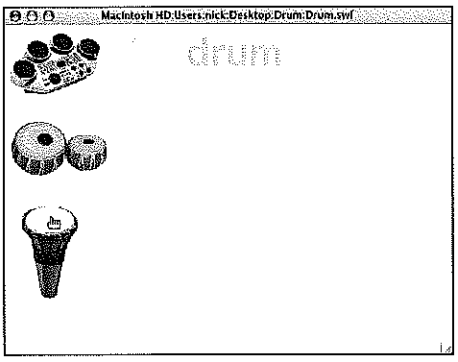
Object-oriented design (OOD)

The concepts of classes, objects, methods and properties are the tools of OOP. The challenge is designing what you want to build with these tools. You have the hammer, nails and wood but the blueprint has to be drawn before you can begin to build. Object-oriented design is the 'draw a blueprint' phase of OOP.

Drum project

In the following exercises you will create an interactive web page using OOP. The page will display three sets of drums, when you click on each set a different message will be displayed. The completed project can be viewed by opening the supplied file *Drum.swf.*

The three drum images on the left hand side of the page are movieclips which operate as buttons. When they are clicked content appears on the page. The content linked to each of the images is also contained within individual movieclips. The title on the page, *drum,* is another movieclip.



All elements on this page have been loaded into, and are displayed via, an empty *container* movieclip. All of the clips used are linked to and controlled by external scripts.

If you open the file *Drum.fla* you will notice that the entire movie contains just one empty frame. The properties and methods of all of the movieclips are defined in external ActionScript files. The script linked to the empty container movieclip is responsible for the creation, initialising and layout of all other elements of the movie.