Becoming a Programmer Workshop 15 October 2001

Phillip Kerman
www.phillipkerman.com (supplements: www.phillipkerman.com/fk01/)

Basics

Flash Hierarchy (`01_fish.swf`)
Review of nested clips.

Relative vs. Absolute Addressing (`02_relative_vs_absolute.swf`)
Addressing, targeting, or otherwise referring to clip instances.
this keyword; _parent keyword; _root keyword.

Explicit vs. Dynamic (`03_explicit_dynamic.swf`)
Form: address["string expression"].property
Example: _root["ball_"+n]._x;

Coordinate System (`04_coordinates.swf`)
_root's top left is 0,0
clip's center is 0,0

General Scripting Theory

Writing Instructions
Scripting/programming is nothing more than writing instructions that you want Flash to follow.

Events (`05_events.swf`)
Events trigger your scripts (that is, they determine when the instructions are followed).
Script in keyframe executes when the keyframe is reached.
Script on a button instance executes when the button event occurs (e.g. press).
Form: on (event){ //do this}
Script on a clip instance executes when the clip event occurs (e.g. enterFrame).
Form: onClipEvent(event){//do this}

Clip Events (`06_faces.swf`)
Example of simple clip events.

Demo trick: toggling between Normal and Expert Mode

Syntax

Pseudo Code (`07_pseudo_code.swf`)
See how to refine your own words into ActionScript.

Dot Syntax (`08_dot_syntax.swf`)
General to specific
oregong.portland.weather
clip.clipInClip.ClipInsideThatOne._x

Syntax (continued)

Special Characters (`09_miscellaneous.swf`)

//comment; / start comment; */end comment;*

Expressions and Statements (`10_expressions_statements.swf`)

Statements are like complete sentences; expressions like phrases.
Expressions are evaluated.

Methods (`11_methods.swf`)

Properties are static characteristics; methods are processes.
Hair color is a property; brushing your hair is a method.

Operators (`12_operators.swf`)

Operators "operate" on operands.
Operators can create expressions (that get evaluated) or statements (that cause a change).

Expression Practice (*13_expression_practice.swf*)

*newPrice = price – (price * 0.1);*
shadow._x = box._x + 10;
circle._x = square._x – (square._width/2) + (circle._width/2);
*blue._x = square._x * (blue._x>square._x);*
*score = score * (timesCheated>0);*
*percent = (line._x – bar._x – (bar._width/2)) / bar._width * 100;*
*percent = (box._x – min) / (max – min) * 100;*

Structures

If-statement (`14_if.swf`)

Form: if(condition){//do this}
Form: if(condition){} else {}
Don't forget: == is comparison; = is assignment.

For-loop (`15_for.swf`)

Form: for (init; condition; next){}
Example: for (i=1; i<11; i++){}
Form: while(condition){}
break keyword will jump you out of the current loop.

Fun with Built-in (Simple) Objects

Math Object (`16_objects_math.swf`)

A suite of common math operations that return results.
Form: Math.method();
Form: Math.CONSTANT;
degrees(Math.PI/180) //returns radians*
radians/(Math.PI/180) //returns degrees

Fun with Objects (continued)

String Object (`17_objects_string.swf`)
Interesting operations on strings.
Remember: start counting characters 0,1,2,3...

Variables (`18_variables.swf`)
Variables are a way to store information for later reference.
Variables "live" in one timeline. As such, they are like homemade properties.

Functions (built-in) (`19_functions_builtin.swf`)
By definition, functions return values.

Functions (homemade) (`20_functions_homemade.swf`)
Form (in keyframe): function name() {}
Accepting Parameters: function name(paramName){}
Returning values: function name (){ return "whatever"; }
To make homemade methods just put function inside a clip.

Arrays (21_arrays.swf)
A way to store multiple values in one variable.
Populate: myArray=["index0", "index1", "index3"];
Access: myArray[0] //to return item in first slot.
Change: myArray[0] = "newValue"
Multidimensional Arrays (simply place entire array into an index).
Arrays (plus objects and clips) are a reference data type (compare to copying a shortcuts).
Primitive (or value) data types are more intuitive (compare to copying a file).
Associative Arrays are better named "generic objects" (see Objects Homemade below).

Built-in Objects (that require instantiation)
Instantiating a movie clip is easy—just drag it from the library. But "soft" objects (like Sound, Color, and Date) require that you create instances by stuffing them into a variable—mySound=new Sound(). Then you can set properties and apply methods (on the mySound variable) like any other object (identically to clip instances).

Sound Object (`22_objects_sound.swf`)
Gives you a way to control sounds with scripting.
Form: mySound= new Sound(); mySound.attachSound("indentifier"); mySound.start();

Applied Exercise (`22_sound_applied_workshop_5.fla`)
See how easy it is to add a fading sound that matches an alpha change.

Color Object (`23_objects_color_date.swf`)
Control a clip's color effect with scripting.
Form: myColor = new Color ("clip") //notice "clip" is in quotes.
myColor.setRGB(0xff0000); myColor.setRGB(r<<16 | g<8 | b); //replace r,g,b with 0-255

Built-in Objects (continued)

Date Object (`23_objects_color_date.swf`)

Do fun stuff with dates.
Form: now = new Date() //or: indyDay = new Date(1776, 6, 4);
now.getDay();// of week starting 0=sun
now.getMonth(); //starting 0=jan
now.getDate(); //of month starting
Use getTime() to compare two dates.

Homemade Objects (`24_objects_homemade.swf`)

In the most basic sense, an object is just a way to store complex structured data.
Form: myObj = new Object(); myObj.prop="val"; // trace(myObj.prop) returns "val"

Constructor function to serve as a template:

```
   function makeBike(){
      this.wheelcount=2;
   }
   roadBike=new makeBike();  bmx=new makeBike();  //now you have two instances.
```

Constructor can also accept (and use) parameters):

```
   function makeBike(tireSize, frameColor){
      this.tire=tireSize;
      this.color=frameColor;
      this.wheelCount=2;
  } // instantiate with: bmx=new makeBike(18, "silver")
```

To create a method, first make the function:

```
         function rePaint(newColor){
            this.color=newColor;
            this.layers++;
         }
```

...then, extend the constructor's prototype property (this property contains all methods):
makeBike.prototype.paint=rePaint;   //now you can do bmx.paint("red");

Creating a constant is similar to making a method. Just give the prototype a property—but point to a value (not a function):
makeBike.prototype.FUEL="leg power";

To inherit all methods and properties from a parent constructor use:
child.prototype = new parent(); //where child and parent both have constructors.
Compare to child.prototype.method = function. This just extends child by adding a new method... the first case replaces all methods of child with those of parent.

--Object example 1

```
function transportation(){ this.position = 0; }
function advance(){ this.position += this.speed; }
//make move() method for transportation object:
transportation.prototype.move = advance;

function makeBike (frameSize){
   this.size = frameSize
   this.speed = 20;      }

function makeCar (model){
   this.model = model
   this.speed = 100;      }

//make makeCar and makeBike objects inherit everything from transportation objects:
makeBike.prototype = new transportation();
makeCar.prototype = new transportation();

//then...
mySUV = new makeCar ("jeep");
myConvertible = new makeCar ("corvette");

trace("before: " + mySUV.position); //"before: 0"
mySUV.move();
trace("after: " + mySUV.position); //"after:100"
```

--Object example 2

Alternatively, you can make a child inherit just the methods of a parent:

```
child.prototype.__proto__=parent.prototype;

function transportation(){ //na}
function advance(){ this.position += this.speed; }
transportation.prototype.move = advance;
function speedUp(){ this.speed += (this.speed/10);}
transportation.prototype.speedUp = speedUp;

function bike (frameSize){
   this.size = frameSize
   this.speed = 20;      }
function car (model){
   this.model = model
    this.speed = 100;      }
function getBikeData(){
   return this.size + " inch bike going " + this.speed;}
bike.prototype.report = getBikeData;
bike.prototype.__proto__ = transportation.prototype;
...then
racer = new bike (27); trace(bmx.report());   // "18 inch bike going 20"
bmx = new bike (18);  trace(racer.report()); //"27 inch bike going 20"
bmx.speedUp();  trace(bmx.report());          //18 inch bike going 22"
```