Chapter 7: Classes and Objects

So far we have used a few objects from predefined classes: MovieClips and Arrays. We have seen that we can create multiple instances of objects and that each instance has the same members and methods, but that the data stored in these object, such as _x and _y, can differ. In this chapter we learn how to create our own Classes. First, we learn another predefined class, the Date class, to see examples of its members and methods. We will then model our classes after this class. The date class is also helpful for allowing us to time game phases, for example the amount of time needed for the player to achieve a goal.

7.1: The Date Class

One use of the Date class is to get the current day and time. Lets look at code to print out the current date and time. All that is used in this code is the Date class and a dynamic text box. Note, you could modify this code to present a running time by writing an appropriate onEnterFrame event handler.

www.cs.du.edu/~leut/1671/flashFiles/c7_date1.fla

```
// create a variable to hold Data objects
var theDateObj:Date ;

// create an instance of a date object
theDateObj = new(Date)  ;



// Create an array with the month names in it
var months:Array = new Array(12) ;
months[0] = "Jan" ;
months[1] = "Feb" ;
months[2] = "Mar" ;
months[3] = "Apr" ;
months[4] = "May" ;
months[5] = "Jun" ;
months[6] = "Jul" ;
months[7] = "Aug" ;
months[8] = "Sep" ;
months[9] = "Oct" ;
months[10] = "Nov" ;
months[11] = "Dec" ;


bx_outString = months[ theDateObj.getMonth() - 1] + " " ;
bx_outString += theDateObj.getDate() + " at " ;
```

```
bx_outString += theDateObj.getHours() + ":" ;
var min:Number = theDateObj.getMinutes() ;
if (min < 10)
  bx_outString += "0" + min ;
else
  bx_outString += min ;
```

As you can see in the code we use the following Date class methods: getMonth(), getDate(), getHours(), and getMinutes(). The Date.getMonth( ) command returns a number 1 to 12 which we then use to index the array to convert to a string. The getDate() method returns the date and the getHours() method returns the hour. The getMinutes( ) returns a number 0 to 60, and hence for 0 to 9 needs to be converted to a string and prepended with a "0". If we did not prepend a zero, then for 5 minutes after 1 o'clock the output would be 1:5, which looks wrong, much better to present as 1:05

A Date object contains data members that hold the current date/time values. We access these members by calls to the methods that return the values, such as getMonth() and getMinutes(). We do not access the data members explicitly, in fact, we have no idea how the information is stored within the object, only the methods we can call to retrieve the information.

The Date class provides many methods. To see all of them, open up the Flash Help manual and type in "Date class". Click on the entry in the ActionScript Dictionary section. In addition to methods that return current values of a date object's member, there are also methods to change the object's date. Again, we do not need to know how the data is stored within the object, all we need to know is the method that we use. Consider the following code to change different parts of an date object's date/time:

www.cs.du.edu/~leut/1671/flashFiles/c7_date2.fla

```
var theDateObj:Date ;
theDateObj = new(Date)  ;

bx_outString1 = theDateObj.toString() ;

theDateObj.setHours(3) ;
bx_outString2 = theDateObj.toString() ;

theDateObj.setYear(2009) ;
bx_outString3 = theDateObj.toString() ;
```

The method *toString( )* converts the date information inside the object into a string that we can then print out. The method *setHours( )* changes the hour value of the data object. Likewise, the method *setYear( )* changes the year value of the object.

When we create our own classes, the Date class provides a good example. There are methods to get information about the contents of the object, often call **Getters,** and methods to change object content, often called **Setters.** Methods *getHours( )* and *getMinutes( )* are examples of getters, and *setHours( )* and *setYear( )* are examples of setters.

In general, we want to create classes that **encapsulate** data members into a logical whole, providing **getters** and **setters**. Because we hide the actual storage of data from the user of the class, this is also refered to as creating an **abstract data type.** The MovieClip example provides an example of poor data abstraction. The class makes the members directly accessible. For example, we explicitly incremented the _x and _y data members to make our gargoyles move about. A much better object-oriented approach would

Before we move on to creating our own classes, lets first consider one more use of Date objects. Games often require timing of events. We may have an obstacle course and players try to get their best time. One way to get times is by using the ActionScript Date class. Say you want to calculate the time required to complete a task. If you get the times at the beginning and end of the task, you can then just take the difference to get the elapsed time. The Data class method *getTime()* returns the time in milliseconds since midnight January 1, 1970. Using this call twice you can time tasks:

www.cs.du.edu/~leut/1671/flashFiles/c7_date3.fla

```
var theDateObj:Date = new(Date)  ;
var startTime:Number = theDateObj.getTime() ;

var temp:Number ;
// do a bunch of computations to fill up some time
for (var counter:Number = 0 ; counter < 1000000 ; counter++) {
        temp = 2 * 3 * 6 / 8 + 3 ;
}

var dateObj2:Date = new(Date)  ;
var endTime:Number = dateObj2.getTime() ;
var difference:Number = endTime - startTime ;

// convert difference to seconds
difference /= 1000 ;

trace("startTime = " + startTime) ;
trace("endTime = " + endTime) ;
trace("Elapsed time was: " + difference + " seconds") ;
```

7.2: Classes

It is time to start creating our own classes. Being a class-conscious and self-important society, we will of course start by creating classes for people. In Flash/ActionScript we must put class definitions in separate files, and those files must be named with the class name and end in a .as suffix (just like .fla files end with a .fla suffix). For example, if we create a *Person* class, then the code for the class definition must be in a file named *Person.as* , and this file must be in the same folder/directory as the .fla file that uses the class. Furthermore, only one class definition per file is allowed.

And now a somewhat odd restriction: Flash MX 2004 does not let you edit the .as files! (Although the Flash Pro version does). This is not a big deal, it just means you need to use a different editor for the .as files. You can use any editor you want, prefer vi or vim or gvim, but you can use pico, emacs, NotePad (windows) or TextEdit (Mac).

Now that we have those formalities out of the way, lets create a Person class:

www.cs.du.edu/~leut/1671/flashFiles/Person.as

```
class Person  {
        private var name:String ;
        private var age:Number ;

        // constructor,  NOTE, unlike C++/java, only ONE constructor is allowed

        function Person (newName:String, newAge:Number) {
                name = newName ;
                age = newAge ;
        }

        // set methods
        public function setName(n:String) : Void {this.name = n ;}
        public function setAge(age:Number) : Void {this.age = age ;}



        // get methods
        public function getAge() : Number {return age ;}
        public function getName() : String {return name ;}

        // print method: prints out the object contents
        public function print() : Void {
                trace(this.name + " " + this.age) ;
        }

}
```

There are several things to notice about this class definition. First, you have the word *class*, then the class name, and then the definition is all contained within the curly braces. The first part of the definition is the data members. In this case we have two members: *name* and *age*. These look like normal variable definitions except they are preceded by

the word *private*. There are actually two choices: private or public. If you do not specify public or private, the default is public. We explain the difference between these two below. Next comes the *constructor* function. The constructor function is used to construct new objects. This particular constructor specifies that two parameters are expected: a String and a Number. The string is assigned to the member *name* and the number is assigned to the member *age*. Following the constructor function we have the classes methods, which we have divided into set methods (setAge and setName) and get methods (getAge and getName). In addition there is a print method. The class and its methods are used by ActionScript code that creates instances (objects) of the class and uses the methods on those objects:

www.cs.du.edu/~leut/1671/flashFiles/c7_class1.fla

```
var p1:Person ;
p1 = new Person("Mary Jones", 19) ;
trace("p1.name is " + p1.getName() ) ;

trace("p1 contains:") ;
p1.print() ;

p1.setAge(20) ;
trace("p1 contains:") ;
p1.print() ;

var p2:Person ;
p2 = new Person("Davy Jones", 60) ;

trace("p2 contains:") ;
p2.print() ;
```

In this code two objects of type Person are created: p1 and p2. The statement:

```
var p1:Person ;
```

creates a variable that can hold Person objects. The line:

```
p1 = new Person(“Mary Jones”, 19) ;
```

creates and actual object and puts it into the variable p1. The word “new” creates a new object. When the object is created the constructor method is used, hence “Mary Jones” is the string refered to as newName, and 19 is the number referred to as newAge. The method *getName()* is used in the first trace statement. Later on the methods *print()* and *setAge()* are used.

Lets go back to the issue of private versus public. If a data member is specified as private is not accessible by any code other than the class definition code, i.e. only the class constructor and methods can access the data. Say I tried to add the following to the above .fla code:

```
p1.name = “Frank Zappa” ;
```

This causes an error because the member *name* is declared to be private. Try it and see what happens. You should get an error message something like:

```
**Error** Scene=Scene 1, layer=Layer 1, frame=1:Line 8: The member is private and cannot be accessed.
    p1.name = "Frank Zappa" ;
```

If a member is declared as private it cannot be accessed by code other than the class method code. If the member declaration is changed to public this access is allowed. Go ahead and change both words *private* to *public* and see what happens when you access p1.name. Now it works just fine. Declaring data members to be private is a way to do *data hiding*. Data hiding allows you to develop classes that the user of the class can only use in the way you intended: i.e. through the methods you make available to them. This approach has been shown to facilitate writing correct code.

Methods can also be either public or private. A private method would be one that is intended to be called only by other methods within the class. Again, if you do not specify public or private the default is public. Consider the following expanded Person class, called Person2():

www.cs.du.edu/~leut/1671/flashFiles/Person2.as

```
class Person2  {
      private var name:String ;
      private var age:Number ;
      private var gender:String ;
      private var ssn:String ;

      // constructor,  NOTE, unlike C++/java, only ONE constructor is allowed

      function Person2 (newName:String, newAge:Number,
                        newGender:String, newSSN:String) {
            setName(newName) ;
            setAge(newAge) ;
            setGender(newGender) ;
            setSSN(newSSN) ;
      }

      // set methods
      private function setName(n:String) : Void {this.name = n ;}
      private function setAge(age:Number) : Void {this.age = age ;}
      private function setGender(gender:String) : Void {this.gender = gender ;}
      private function setSSN(ssn:String) : Void {this.ssn = ssn ;}

      // get methods
      public function getName() : String {return name ;}
      public function getAge() : Number {return age ;}
      public function getGender() : String {return gender ;}
      public function getSSN() : String {return ssn ;}
```

```
// print method: prints out the object contents
public function print() : Void {
  trace(this.name + " " + this.age + " " + this.gender + " " + this.ssn) ;       }}
```

There are a few things to point out here. First, we have added two new members, gender and ssn, and the set and get methods for them. Second, we have changed all the set methods to be private. The data members are also private, thus, there is no way to change a data member once the object is created. So why bother having the set methods at all? The reason they exist is that they are called from within the constructor. Calling class methods from another method is a common practice, and sometimes the called methods are private, meaning the intent is to use these methods only within the class definition. This is usually done because the method is called frequently from other methods and hence it reduces code size, or because the task done in the method is complicated and breaking apart the functionality makes for more understandable code. In this Person2 example, it really does not make sense to have the set methods if they are only private since that code is only used in the constructor and should probably just be put in the constructor. But, if they are intended to be public, then it certainly makes sense to also call the public methods from within the constructor.

To see the effect of private methods, consider the following code:

www.cs.du.edu/~leut/1671/flashFiles/c7_class2.fla

```
var p1:Person2 = new Person2("Mary Jones", 19, "f", "308-55-5175") ;
trace("p1.name is " + p1.getName() ) ;
trace("p1 contains:") ;
p1.print() ;

var p2:Person2 = new Person2("Davy Jones", 60, "m", "303-87-2000") ;
trace("p2 contains:") ;
p2.print() ;

// if you try to do the following it will be an error since setAge() is private
// p2.setAge(15) ;
```

Notice the last line. It is commented out. If you uncomment it and run the code you get an error. That is because the *setAge()* method is private. If you change it to public the code will run just fine.

In summary, when we create a class:

- It must be stored in a separate file whose name is the same as the class name and ends in a ".as" suffix
- Only one class per file
- These external files can be edited with any editor (vi, emacs, notepad, TextEdit)
- A class may have public or private data members. If you make them private, which in general we recommend, then the can only be accessed by member methods, thus the class designer can control how member data is accessed.
- Member methods usually include **set** methods and **get** methods

- You should always create a constructor method

Now that we have the basics down, lets move on to creating a more interesting class.

7:3: The MovingMC class

So far we have used the built-in MovieClip class for our avatars. To make them do what we wanted we had to have external velocities (g1xv and g1yv for example) or arrays of velocities. In addition we had to explicitly move them around. We can improve upon the MovieClip class by creating our own class that adds functionality to it and has a MovieClip object within. Consider the following class definition:

www.cs.du.edu/~leut/1671/flashFiles/MovingMC.as

```
class MovingMC {

        // (1)
         private var x:Number ;
         private var y:Number ;   // current x,y location
         private var xv:Number ;   // x velocity
         private var yv:Number ;   // y velocities
         private var xscale:Number ;
         private var yscale:Number ;
         private var width:Number ;
         private var height:Number ;
         private var internalName:String ;
         private var depth:Number ;
         private var mc:MovieClip ;

        // (2)
        function MovingMC(x:Number, y:Number, xs:Number, ys:Number, xv:Number,
yv:Number, linkName:String) {

                var nextDepth:Number = _root.getNextHighestDepth() ;
                internalName = "mc" + nextDepth ;
                this.mc = _root.attachMovie(linkName,internalName,nextDepth) ;
                setX(x) ;
                setY(y) ;
                setXscale(xs) ;
                setYscale(ys) ;
                setXV(xv) ;
                setYV(yv) ;
                setWidth(mc._width) ;
                setHeight(mc._height) ;

        }


        // (3)
        public function setX(newX:Number) : Void {    this.mc._x = newX ; this.x = newX ;}
        public function setY(y:Number) : Void {this.mc._y = y ; this.y = y ;}
```

```
        public function setYscale(ys:Number) : Void    {this.mc._yscale = ys ; this.yscale = ys ;}
        public function setXscale(xs:Number) : Void    {this.mc._xscale = xs ; this.xscale = xs ;}
        public function setXV(xv:Number) : Void {this.xv = xv ;}
        public function setYV(yv:Number) : Void {this.yv = yv ;}
        public function setWidth(width:Number) : Void {
            this.width = width ;
            this.mc._width = width ;
        }

        public function setHeight(height:Number) : Void {
                    this.height = height ;
                    this.mc._height = height ;
        }


        // (4)
        public function getX() : Number {return x ;}
        public function getY() : Number {return y ;}
        public function getXV() : Number {return xv ;}
        public function getYV() : Number {return yv ;}
        public function getWidth() : Number {return width ;}
        public function getHeight() : Number {return height ;}
        public function getMC() : MovieClip {return mc ;}
        public function getInternalName() : String {return internalName ;}
        public function getInternalDepth() : Number {return depth ;}

        // (5)
        public function updateX() : Void {this.x += this.xv ; this.mc._x = this.x ; }
        public function updateY() : Void {this.y += this.yv ; this.mc._y = this.y ; }
        public function updateXY() : Void {updateX() ; updateY() ;}

        public function destroy() : Void { removeMovieClip(mc) ; }

        // (6)
        public function hit(mc2:MovingMC) : Boolean {
                    if (this.mc.hitTest(mc2))
                       return(true) ;
                     else
                       return(false) ;
        }


        public function print() : Void {

                    trace("Inside MovingMC.print(): " + this.x + " " + this.y + " " +
                    this.xv + " " + this.yv + " " + this.width + " " + this.height) ;
        }
}
```

The code after the (1) declares variables to hold the data member. The code after the (2) is the constructor. The constructor creates a new MovingMC object and has input parameters of the x,y location, the desired x-scale and y-scale, the desired x-velocity and y-veolocy, and the name of the MovieClip exported in the Flash library. The first three lines of this constructor create an instance of a MovieClip object and store it as a member (this.mc) of the MovingMC object. Remember that Flash MovieClip objects need a unique name and a

unique depth level. The built in function, *root.getNextHighestDepth()*, returns the next unused depth within the Flash animation, thus, we can use it to ensure unique depth levels and unique internal names. We accomplish the later by appending the depth to the characters "mc". The rest of the constructor is a sequence of calls to set methods. This set methods are defined after the (3). Note that these set methods set the values of the MovingMC as well as the embedded MovieClip.

The code after the (4) defines a bunch of get methods. The code after the (5) is pretty slick. Method updateX() increases the x-coordinate of the MovingMC (and hence also the embedded MovieClip) by using the xv data member. You do NOT need to keep track of separate velocities, they are built right into the object. Likewise, methods updateY() and updateXY() update the y-coordinate or both coordinates respectively. The code after the (6) provides a hit test for this MovingMC and a second MovieClip passed in as a parameter. Finally, the last code prints out the contents of a MovingMC object, and is helpful for debugging. Now lets consider some code that uses this class:

www.cs.du.edu/~leut/1671/flashFiles/c7_class3a.fla

```
var mc1:MovingMC ;
var mc2:MovingMC ;
var mc3:MovingMC ;

mc1 = new MovingMC(50,50,25,25,3,2,"penguin") ;
mc2 = new MovingMC(75,78,25,25,3,2,"penguin") ;
mc3 = new MovingMC(100,100,25,25,3,2,"penguin") ;

onEnterFrame = function () {
        mc1.updateXY() ;
        mc2.updateXY() ;
        mc3.updateXY() ;
}
```

This code creates three MovingMC objects. Notice the arguments to the constructors all differ in the x,y location. Then, the onEnterFrame function calls updateXY() on each of these three objects.

Notice how much easier it is to create and move objects using this approach. In general classes allow for easier to understand code because a lot of the complexity is suffered only once, in creating the classes, and then we can just use the much simpler code.

What if we want to create lots-o-penguins? Again we turn to for loops and arrays:

www.cs.du.edu/~leut/1671/flashFiles/c7_class3b.fla

```
var tempMMC:MovingMC ;          // temporary holder of a MovingMC object
var numPenguins = 60 ;          // number penguins we want to create
var penguinArray:Array = new Array(numPenguins) ;          // array to hold MovingMC objects

var randX:Number ;  // holds a randomly generated X-coordinate
var randY:Number ;  // holds random Y-coord
```

```
var randXV:Number ;  // holds a randomly generated x-velocity
var randYV:Number ;  // holds random y-velocity


// create numPenguins penguing MovingMC objects randomly located with random velocities
for (var i:Number = 0 ; i < numPenguins ; i++) {
        randX = Math.random() * Stage.width ;
        randY = Math.random() * Stage.height ;
        randXV = Math.random() * 3 - 1.5 ;    // random between -1.5 and 1.5
        randYV = Math.random() * 3 - 1.5 ;

        // create a new penguion MovingMC object with random x,y,xv,yv as above, scale 25
        tempMMC = new MovingMC(randX,randY,25,25,randXV,randYV,"penguin") ;

        // store this MovingMC in the array
        penguinArray[i] = tempMMC ;
}

onEnterFrame = function () {
        // loop through the array and update the X,Y locations of each MovingMC object
        for (var i:Number = 0 ; i < numPenguins ; i++)
                penguinArray[i].updateXY() ;
}
```

Here we have a for-loop that creates a MovingMC object (with random initial x,y coordinates and random x,y-velocities) and then assigns these objects to an array. Then, inside the onEnterFrame function, we loop through the array calling *updateXY()* on each object in the array.

Note when you run this, the penguins eventually move off the screen never to return. This is because we have no boundary checking conditions and velocity-negation. We need to add this to our code. We could do this using MovingMC class' getX(), getY(), setX(), and setY(), but, it probably makes more sense to build this behavior right into the class!

NOTE: MORE TO BE ADDED SOON