

Written by Billy D. Spelchan for [www.BlazingGames.com](http://www.BlazingGames.com)

Copyright © 2003-2005 Blazing Games Inc. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the file called fdl.txt

# Chapter 11

## Building Card Classes

### Contents

Both games we are creating in this part share the fact that they use a deck of cards. By creating a general purpose card class, we only have to do the work of creating and managing a deck of cards once. In this chapter, we will be creating a card movie for displaying any of the fifty-two cards that make up a deck of cards. We will also be creating a deck class for managing a deck of cards.

- How to create a Deck of Cards - We look at three ways to create a deck of cards.
- Card Components - Building the card components.
- Creating the Card Movie - Making a deck of cards.
- Testing the Card Movie - Let's make sure the card movie works.
- Flash Classes - An overview of Flash classes.
- Creating the Deck Class - Building the deck class.
- Testing the Deck - Let's make sure the deck class works.

## How to create a Deck of Cards

As you already know, a deck of cards consists of 52 cards (54 if you count the two jokers, but for now we won't worry about those). Obviously this means that we are going to need 52 images. In addition to the images for every card, we are also going to need an image for the back of the cards.

There are two ways we can create images for the cards. One way, if you already have a set of bitmaps for the cards, is to simply import the 52 bitmap images. The advantage of this is speed. The disadvantage is that the resulting images are not vectors and will not scale well. One way to get around the vector problem would be to convert every card into a vector image. This only takes a slight bit more time and you end up with better scalability.

The other way is to build the cards in Flash (or a vector drawing program that lets you export images to flash). This can potentially reduce the overall size of the deck by a fair amount. Having the cards proper flash vectors also enables very good scalability. If you don't already have a set of bitmap cards, then creating the cards in Flash will take just as long as it would to create them in a bitmap based paint program.

## Card Components

For this book, we are going to build all the cards in Flash. One way would be to simply create 53 graphic images. This works, but when you look at a deck of cards you will quickly notice that the cards are only made up of a small subset of images. If we create a series of components for the components that make up a card, we will easily be able to create the entire deck by assembling our components. This helps reduce both the time it takes to build the deck of cards and the amount of memory required for the cards.

When you think about it, a set of cards can be broken into 34 components. Better yet, most of these are small components that only take a few curves to represent. The components used in this are shown in figure 1.



**Figure 1**

Now, the top line consists of “Blank Card”, “Jack”, “Queen”, King. The Blank Card symbol is the background symbol. Notice that we surround the card with a hairline box. The card is 100 by 140 as that is the resolution I had in the bitmap deck that I created for my Java card games. The cards could be any size you desired, and because they are vectors the cards will look good scaled up or down.

The second line Consists of the suits, named “Spade,” “Heart,” “Club,” and “Diamond”. The third and fourth lines consist of the card face value text. There are two versions, one for black cards and one for red cards. The symbols are named, “Blk A,” “Blk 2,” “Blk 3,” “Blk 4,” “Blk 5,” “Blk 6,” “Blk 7,” “Blk 8,” “Blk 9,” “Blk 10,” “Blk J,” “Blk Q,” “Blk K,” “Red A,” “Red 2,” “Red 3,” “Red 4,” “Red 5,” “Red 6,” “Red 7,” “Red 8,” “Red 9,” “Red 10,” “Red J,” “Red Q,”

and “Red K.”

## Creating the Card movie

The first thing we are going to need to create is a card movie. The purpose of this movie is to hold the images for all the cards and go to the appropriate image based on what value is assigned to the card. I have decided to give each card a numerical ID, which is as follows. The ID of 0 represents no visible value (in other words the card back is shown). cards are then given ID in numerical order, with Aces being the first card and kings being the last. We go through the suits in alphabetical order, so we have the Clubs, Diamonds, Hearts, then Spades.

As the ID can be simply calculated based on the value of the card (Aces being valued as 1, Jacks valued as 11, Queens valued as 12 and Kings valued as 13), it makes sense to take advantage of this fact. We do this by mathematically determining the position of the card. How? Every card is a single frame. We have the card back on frame one and start the first card on frame 11. We then simply need to add 10 to the ID to find the correct frame for the card face.

Now, a bit of code is going to be in order. We are going to need to get and set the ID of the card. We also want to be able to show the back or the front of a card. While neither of the games in this part of the book use this feature, it is simple enough to implement and will be of obvious use with many other card games.

We will place all the code for the Card movie in frame two of the movie. To start of, we need a bit of initialization code. This code simply sets up the variables that we use in the movie. Notice that the code is designed to run only once. We default to no card ID and the back of the card being shown.

```
initCard();
stop();

function initCard()
{
    if (cardInitialized != undefined)
        return;
    cardInitialized = true;
    cardID = 0;
    faceShowing = false;
}
```

Next, we are going to need some functions for getting and setting the card's value. When the card's value is changed, the card has to go to the proper frame of the movie and stop. However, if the card's face is not showing, we show the back image instead. Getting the card value is simply the matter of returning the cardID variable.

```
function setCard(n)
{
    cardID = n;
    if ((cardID == 0) || (faceShowing == false))
        gotoAndStop(1);
    else
        gotoAndStop(10+cardID);
}

function getCard()
{
    return cardID;
}
```

Finally, we are going to add functions that will let us show and hide the card's face. Showing a card is simply a matter of setting the faceShowing variable to true and going to the appropriate frame. The only complication is the fact that we need to handle a card that has yet been assigned a value.

```
function showFace()
{
    faceShowing = true;
    if (cardID == 0)
        gotoAndStop(1);
    else
        gotoAndStop(10+cardID);
}

function hideFace()
{
    faceShowing = false;
    gotoAndStop(1);
}
```

## Testing the Card Movie

It is always wise to test any code that you create. When you are creating a movie or class that is going to be used by more than one movie, then testing becomes even more important. With that in mind, let us test the Card movie that we just created. The test will simply loop through every card in the deck. While displaying the state of a card it will show both the front and back of the card. This way you can simply watch all the cards reveal themselves.

To start with, we will create a movie with two layers. A copy of the cards folder should be copied to the library of our test movie. Within this folder is the Card movie. We will drag an instance of the Card movie to the middle of the screen. Set the Instance Name of the movie to `card_movie`.

Now in the other layer, which we will call the code layer, enter the following code in the first frame. This line simply holds the current ID for the card we are going to show.

```
curCard = 1;
```

Now, in Frame 5 of the code layer we create a keyframe. We then add the code to set the value of the card and make sure the card's back is showing. We also want this to be the location to be looped to, so we set the frame's label to "showLoop".

```
card_movie.setCard(curCard);  
card_movie.hideFace();
```

We want the back to be shown for about a second so we will leave 10 frames. On frame 15 of the code layer we add another keyframe. This time, we are going to show the face of the card by having the following line of code.

```
card_movie.showFace();
```

Finally, we leave 20 frames so you will be able to see the card being show. Frame 35 we then add the looping code. This code simply adjusts the card ID of the next card to be shown and then loops back to the showLoop. Notice that we need code to make sure the next ID to be used is valid.

```
++curCard;  
if (curCard > 52)  
    curCard = 1;  
gotoAndPlay("showLoop");
```

And now we have a demo of our card class.

## Flash Classes

We now have cards, but we are going to need a way of shuffling and dealing cards. While we could attach such code to the movie (which is essentially a class anyway), this doesn't make too much sense, as the card movie represents a single card, while the deck represents 52 cards. So, we are going to create a separate deck class. This, however, is not a movie clip. Instead we will manually create the class.

Before we can create the deck class, however, we are going to need to know a bit more about how classes work and how to create them. You should already be familiar with the new operator, which is used for creating instances of a class. The format for this is

```
new constructor([param1,...,paramn]);.
```

What is actually happening is the new operator is trying to find a constructor function named whatever name was provided with the number of parameters indicated. Some constructor functions, such as Array, are built right in to Flash. It is, however, possible to create new Constructor functions by simply having a function named whatever it is you want to call the constructor. Convention has it that you start constructor functions with a capital letter. This is a good idea as it distinguishes them from other functions.

Constructors can have no parameters, or as many parameters as you desire. Here is an example of a simple constructor function for creating a point.

```
function Point(x, y)
{
  this.x = x;
  this.y = y;
}
```

It is important that you use the this keyword, as that tells the function that it should be using the variables stored in the instance that is calling it.

Adding functions to a class is a bit tricky to explain. All classes are given a prototype object that holds functions and other variables that are available to any instance of that class. To add a function to a class, you need to add the function to the prototype object. This can be done two ways. First, you can create the function directly in the constructor by assigning the function name to the code for a function. The other way is to assign a function to the prototype, by using `classname.prototype.functionname = function(...) { ... };`

I wrote a demonstration movie that demonstrates this. The first frame of the movie creates two dynamic text objects named “output1\_txt” and “output2\_txt” which displays the result of the test. In the code layer on frame 1 we create the class as follows.

```
function Point(x, y)
{
    this.x = x;
    this.y = y;

    this.getX = function() { return this.x; };
    this.getY = function() { return this.y; };
}

Point.prototype.move = function(x, y)
{
    this.x = x;
    this.y = y;
}
```

On frame 2 of the code layer, we add a break point and have the following code that actually tests the class we created.

```
firstPoint = new Point(10,10);
secondPoint = new Point(10, 10);

output1_txt.text = "Starting points are (" +
    firstPoint.getX() + ", " + firstPoint.getY() + ") and (" +
    secondPoint.getX() + ", " + secondPoint.getY() + ")";
secondPoint.move(21, 42);
output2_txt.text = "Ending points are (" +
    firstPoint.getX() + ", " + firstPoint.getY() + ") and (" +
    secondPoint.getX() + ", " + secondPoint.getY() + ")";
stop();
```

## Creating the Deck class

Now we are ready to create a deck class. While we could create this code right inside the flash program, we are going to create it outside of the Flash editor. If you have Dreamweaver, you can use it to create the file as Dreamweaver understands Action Script. If you don't have Dreamweaver, any text editor will work. To use this file, we then only need to use Flash's `#include` command to include the file. Creating the class this way makes it much more portable and easier to use in other projects.

This class needs to hold 52 cards in a deck, deal out those cards, and shuffle those cards. First, we are going to need to create the constructor function for this class. As all the information we need about the deck is already known this can be a simple empty constructor, which will just so happen to create an array for itself to hold the deck and fill that array with in-order values.

```
function Deck()
{
    this.cardArray = new Array(52);
    for (var cntr = 0; cntr < 52; ++cntr)
        this.cardArray[cntr] = cntr + 1;
    this.nextCardIndex = 0;
}
```

Next, we are going to need a way of shuffling the deck. Think about the different ways to shuffle a deck. There are many ways of shuffling, but most of them are inefficient as far as the computer is concerned. What our goal is, simply stated, is to make the cards in the deck appear in a random order. That is why I came up with my swap method of shuffling a deck.

```
Deck.prototype.shuffle = function()
{
    var cntr, temp, rnd;

    for (cntr = 0; cntr < 52; ++cntr)
    {
        rnd = Math.floor(Math.random() * 52);
        temp = this.cardArray[cntr];
        this.cardArray[cntr] = this.cardArray[rnd];
        this.cardArray[rnd] = temp;
    }
    this.nextCardIndex = 0;
}
```

We need to be able to deal cards from the deck. To do this we just take the card pointed at by the `nextCardIndex` variable and then increment `nextCardIndex` so that it points to the next card.

```
Deck.prototype.draw = function()
{
    var rv = this.cardArray[this.nextCardIndex];
    ++this.nextCardIndex;
    return rv;
}
```

And finally, we want to be able to clone a deck. Why? our squares games are going to use this in order to allow a player to replay the same deal. By passing the deck we want to clone as a variable, we need only have to copy the variables stored in the source class.

```
Deck.prototype.cloneDeck = function(source)
{
    for (var cnt = 0; cnt < 52; ++cnt)
    {
        this.cardArray[cnt] = source.cardArray[cnt];
    }
    this.nextCardIndex = source.nextCardIndex;
}
```

There are other things that we could probably add to this class. For the two games we are creating in this book, and for that matter for most other card games, this is enough.

## Testing the Deck

The deck class test is very similar to the card test we created earlier. This class creates two copies of the deck class. It shuffles one deck and then clones the deck with the other class. We then reveal all the cards and end the demo once all 52 cards have been shown.

To start with, we will create a movie with two layers. A copy of the cards folder should be copied to the library of our test movie. Within this folder is the Card movie. We will drag an instance of the Card movie to the middle of the screen. Set the Instance Name of the movie to `card_movie`.

Now in the other layer, which we will call the code layer, enter the following code in the first frame. It simply tells Flash to include an external source file when building the swf file. Notice that there is no semicolon at the end of the line.

```
#include "Deck.as"
```

Next, in the second frame, we create the two copies of the deck class. We are creating two copies so that we can test the `cloneDeck` function. We again use a `curCard` variable, but this time it is simply used to count how many cards we have shown.

```
sourceDeck = new Deck();
sourceDeck.shuffle();
myDeck = new Deck();
myDeck.cloneDeck(sourceDeck);
curCard = 0;
```

Now, in Frame 5 of the code layer we create a keyframe. We then add the code to set the value of the card and make sure the card's back is showing. Notice the value of the card is obtained by calling the `draw` function in the Deck class we are using. We also want this to be the location to be looped to, so we set the frame's label to "showLoop".

```
card_movie.setCard(myDeck.draw());
card_movie.hideFace();
```

We want the back to be shown for about a second so we will leave 10 frames. On frame 15 of the code layer we add another keyframe. This time, we are going to show the face of the card by having the following line of code.

```
card_movie.showFace();
```

Finally, we leave 40 frames so you will be able to see the card being show. We are having a longer delay this time to be sure that it is possible to mark down what cards have been shown. Frame 55 we then add the looping code. This code simply adjusts the card ID of the next card to be shown and then loops back to the showLoop if all the cards have not been shown.

```
++curCard;  
if (curCard > 52)  
    curCard = 1;  
gotoAndPlay("showLoop");
```

Finally, on frame 56 we change the main layer to a “That’s all folks!” message and place a stop() on the code layer. And now we have a demo of our deck class.

To be positive that all the cards are used, you can simply create a simple chart that lists the four suits on the top and the thirteen face values on the side. As the card appears, check it off the chart. As you will see, this demo works fine.