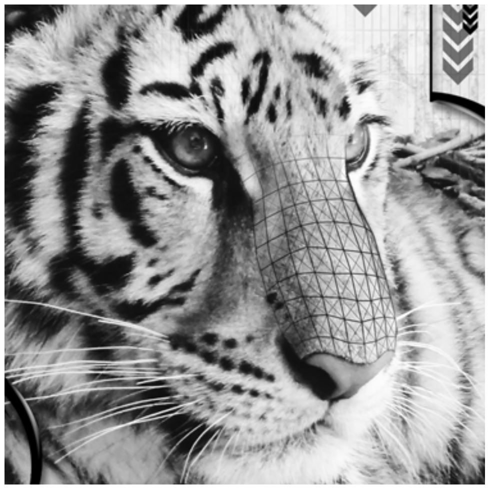


Chapter 13

Dragging and Dropping Objects

To create any kind of game that involves matching objects, such as puzzle or board games, you need to create objects that can be dragged and dropped with the mouse. Lucky for you, any objects that inherit AS3.0's Sprite class, have some built-in properties and methods that are specialized for drag-and-drop interfaces:

- startDrag: Makes an object draggable
- stopDrag: Stops dragging
- dropTarget: Tells you which object is under the mouse when an object is being dragged

Let's first find out how to make a drag and drop interface work, and then see how we can use it to make a simple word matching game.

Basic drag and drop

In the **`DragAndDrop`** project folder in the chapters' source files, you'll find a simple drag and drop example which demonstrates all the basic techniques. Use the mouse to drag the image of the cat onto a matching "cat" target tile. The cat will snap to the target if you let go of the left mouse button, while it's over its matching square, as illustrated in Figure 13-1.

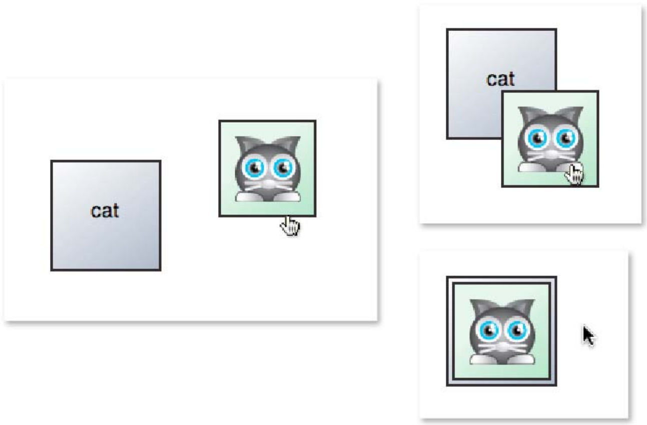


Figure 13-1 Drag the cat to its matching square and it will snap into place.

This project uses three classes to make this work:

A class called `Card` that is the image with the cat on it that you can drag around the stage.

A class called `Match` that is the square with the label "cat".

An application class that creates the card and match objects, and handles all the drag and drop logic.

This example is very different in one very unique way from the other examples we've seen in the book so far. The image of the cat is *not* embedded into the `Card` class. Instead, the application class embeds the image of the cat and passes it to the `Card` class when it creates a card object. Take a look at Figure 13-2, which illustrates how this project is structured. Can you see that the image of the cat is not part of the Card class, and that the Match class doesn't display the word "cat"? We'll see how this works in detail ahead, and how it will become a great advantage when building the matching game.

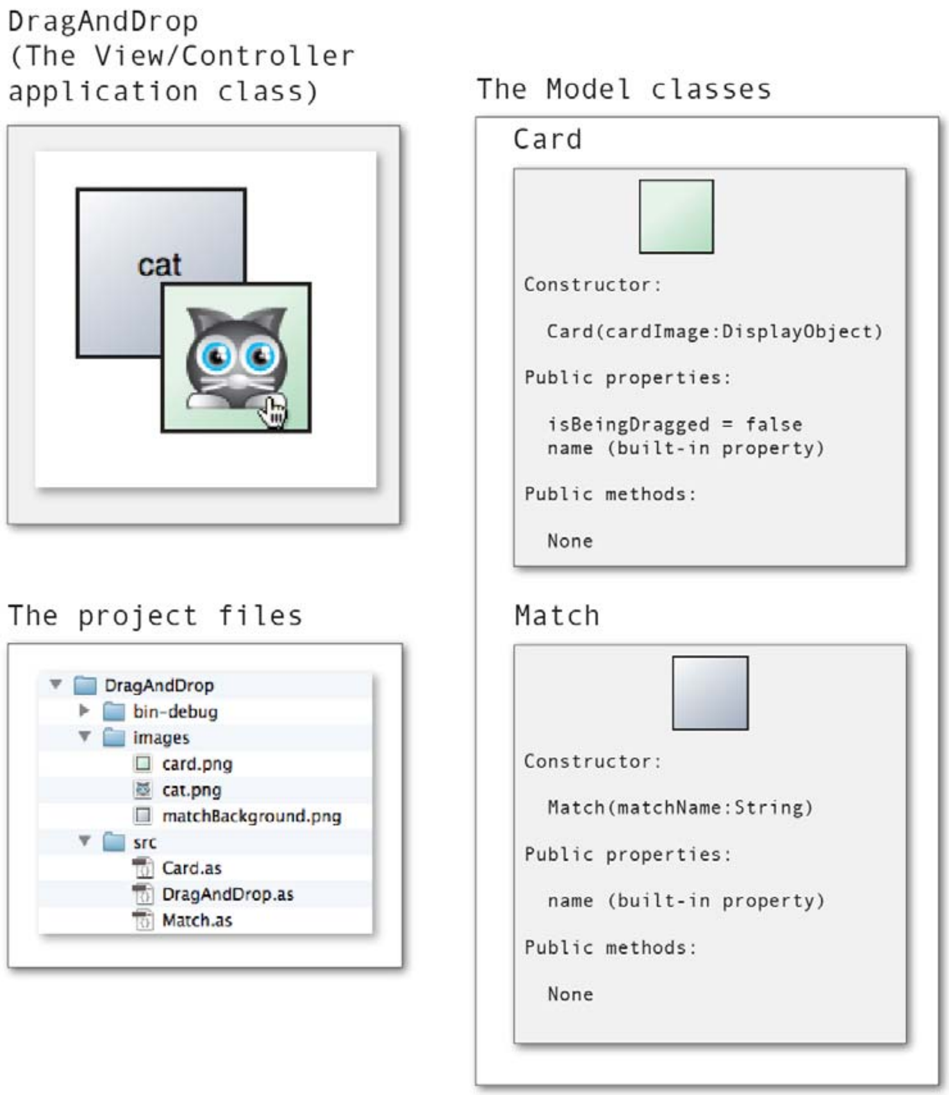


Figure 13-2 The DragAndDrop Project structure.

Here's the entire DragAndDrop application class, and I'll explain how it works ahead.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.MouseEvent;

  [SWF(width="550", height="400",
  backgroundColor="#FFFFFF", frameRate="60")]

  public class DragAndDrop extends Sprite
  {
    //Embed the image for the card foreground
    [Embed(source="../images/cat.png")]
    private var CatImage:Class;

    //Private properties
    private var _catImage:DisplayObject = new CatImage();
    private var _card:Card = new Card(_catImage);
    private var _match:Match = new Match("cat");

    public function DragAndDrop()
    {
      this.addChild(_card);
      _card.x = 350;
      _card.y = 150;
      _card.buttonMode = true;
      _card.useHandCursor = true;

      this.addChild(_match);
      _match.x = 150;
      _match.y = 150;

      //Add event listeners
      _card.addEventListener
        (MouseEvent.MOUSE_DOWN, mouseDownHandler);
      stage.addEventListener
        (Event.ENTER_FRAME, enterFrameHandler);
    }
    public function enterFrameHandler(event:Event):void
    {
      if (_card.hitTestObject(_match))
      {
        if (!_card.isBeingDragged)
        {
          _card.x = _match.x + 5;
          _card.y = _match.y + 5;
             }
      }
    }
    private function mouseDownHandler(event:MouseEvent):void
    {
      var card:Card = event.currentTarget as Card;
      card.startDrag();
      setChildIndex(card, numChildren - 1);
      card.isBeingDragged = true;
      card.addEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
    }
    private function mouseUpHandler(event:MouseEvent):void
    {
      var card:Card = event.currentTarget as Card;
```

```
        card.stopDrag();
        card.isBeingDragged = false;

        //Remove this mouse listener
        card.removeEventListener
            (MouseEvent.MOUSE_UP, mouseUpHandler);
        }
    }
}
```

Composing the Card object

The Card class doesn't embed the image of the cat. Instead, the application class embeds the cat image, and sends it to the Card class when it makes the `_card` object. Here's how it does this:

The application class embeds the image of the cat and creates a `_catImage` DisplayObject using this very familiar code:

```
[Embed(source="../images/cat.png")]
private var CatImage:Class;
private var _catImage:DisplayObject = new CatImage();
```

The application class then sends the `_catImage` to the `Card` class as an argument when it creates the `_card` object:

```
private var _card:Card = new Card(_catImage);
```

The `Card` class receives this as a parameter in its constructor method, and uses `addChild` to display it.

```
public function Card(cardImage:DisplayObject)
{
  this.addChild(cardImage);
```

This is great, because it means that you don't have to decide in advance what kind of image the card will display. You can use this same class, unchanged, for hundreds of different cards, all containing different images. This is a powerful programming technique called **composition**. It's the same technique we used to display a custom message at the end of the game in Bug Catcher, except in this example, we're passing the class an object, not text. Figure 13-3 illustrates how this works.

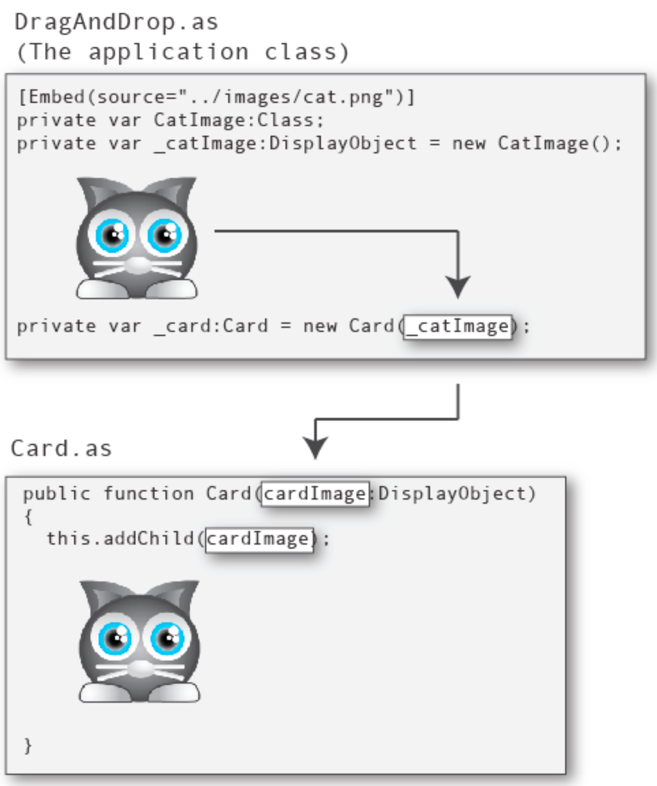

DragAndDrop.as
(The application class)
[Embed(source="../images/cat.png")]
private var CatImage:Class;
private var _catImage:DisplayObject = new CatImage();
private var _card:Card = new Card(_catImage);
Card.as
public function Card(cardImage:DisplayObject)
{
this.addChild(cardImage);
}

Figure 13-3 Use composition to customize the card's image.

The `Card` class embeds the card background image and then centers the cat image over the background. Here's the entire `Card` class that does this.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;

  public class Card extends Sprite
  {
    //Embed the card background image
    [Embed(source="../images/card.png")]
    private var CardBackgroundImage:Class;

    //Private properties
    private var _cardBackgroundImage:DisplayObject
             = new CardBackgroundImage();

    //Public properties
    public var isBeingDragged:Boolean = false;

    public function Card(cardImage:DisplayObject)
    {
      //Add the background image
      this.addChild(_cardBackgroundImage);
      cardImage. x = 12;
      cardImage. y = 12;

      //Add the foreground image to the card
      this.addChild(cardImage);
      cardImage.x
        = (_cardBackgroundImage.width - cardImage.width) / 2;
      cardImage.y
        = (_cardBackgroundImage.height - cardImage.height) / 2;
    }
  }
}
```

The Card class embeds the green square that represents the card's background: `_cardBackgroundImage`. When the class is created, it centers the card image over this background with these two lines of code:

```
cardImage.x
  = (_cardBackgroundImage.width - cardImage.width) / 2;
cardImage.y
  = (_cardBackgroundImage.height - cardImage.height) / 2;
```

This code will center an image of any size over the background as long as the image isn't larger than the background itself.

The `Card` class also contains one public property called `isBeingDragged`.

```
public var isBeingDragged:Boolean = false;
```

This determines whether or not the card is currently being dragged with the mouse, and we'll see how important this will become in programming the drag and drop interface.

Making the Match object

Just like the card, the matching square is also made using composition. The application class sends the `Match` class the name of the thing it needs to match. In this case, it's cat.

```
private var _match:Match = new Match("cat");
```

The `Match` class then accepts this through its constructor parameters and copies it into its `name` property.

```
public function Match(matchName:String)
{
        this.name = matchName;
```

`name` is a built-in property that all Sprite objects have. You can assign any string to a Sprite object's name property.

The `Match` class then displays its name in a text field, like this

```
_output.text = this.name;
```

The `Match` class also embeds a square background image and displays the text on top of it. Here's the entire `Match` class that does all this.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.text.*;

  public class Match extends Sprite
     {
    //Embed the card background image
    [Embed(source="../images/matchBackground.png")]
    private var MatchBackgroundImage:Class;

    //Private properties
    private var _matchBackgroundImage:DisplayObject
      = new MatchBackgroundImage();
    private var _format:TextFormat = new TextFormat();
    private var _output:TextField = new TextField();

    public function Match(matchName:String)
    {
      //Assign the matchName from the application
      //class to this class's name property
      this.name = matchName;

      //Add the background image
      this.addChild(_matchBackgroundImage);

      //Set the text format object
      _format.font = "Helvetica";
      _format.size = 16;
      _format.color = 0x000000;
      _format.align = TextFormatAlign.CENTER;

      //Configure the _output text field
      _output.defaultTextFormat = _format;
      _output.width = _matchBackgroundImage.width;
      _output.border = false;
      _output.wordWrap = true;
      _output.autoSize = TextFieldAutoSize.CENTER;
      _output.text = this.name;

      //Display and position the _output text field
      this.addChild(_output);
      _output.x = 0;
      _output.y = 30;
    }
  }
}
```

Now let's see how the `DragAndDrop` application class makes the card draggable, and how it finds out if the card is over its matching square.

Making the card draggable

The first thing the `DragAndDrop` constructor method does after adding the card to the stage is to make the hand cursor visible when it's over the card

```
_card.buttonMode = true;
_card.useHandCursor = true;
```

This is the same code we used to make the hand cursor appear over the buttons we made in chapter 3. It's purely optional, but a nice effect for drag and drop objects because it cues the user that some kind of mouse interactivity is possible.

The constructor then adds an event listener *directly to the _card object*.

```
_card.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownHandler);
```

This is the same technique we used to attach mouse event listeners to buttons in chapter 3. We haven't seen any code that's done this since that chapter. In all other examples, event listeners were added either to the stage or to "this" class. But it's important to know that you can attach event listeners to *any* object. For drag and drop, this is essential. The code above attaches a `MOUSE_DOWN` listener for when the mouse's left button is pressed down. (To use this mouse event, the class has to import the `flash.events.MouseEvent` class.)

When the mouse button is pressed down over either of these objects, the `mouseDownHandler` is called:

```
private function mouseDownHandler(event:MouseEvent):void
{
        var card:Card = event.currentTarget as Card;
        card.startDrag();
        setChildIndex(card, numChildren - 1);
        card.isBeingDragged = true;
        card.addEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
}
```

Here's what the event handler does:

- It figures out which object has been clicked by accessing the event's `event.currentTarget` property and assigning it to a temporary local variable called `card` (more on the currentTarget property ahead):

```
var card:Card = event.currentTarget as Card;
```

- It then calls the card's `startDrag` method. This starts the object dragging, with the effect you can clearly see on the stage:

```
card.startDrag();
```

All Sprite objects have this built-in `startDrag` method that you can use like this to make them draggable.

- You want the object that you're dragging to be above all the other objects on the stage. This creates the illusion that it's being picked up. To do this, you need to move it to the highest position in the display list. This next line takes care of that quite nicely:

```
setChildIndex(card, numChildren - 1);
```

`setChildIndex` lets you change the position of an object on the stacking order, so that it appears above or below other objects. `numChildren` tells you how many objects you have on the stage. You can make an object on the stage appear above all the others by use `setChildIndex` to a move an object to a position in the stacking order list that's 1 less that the value of `numChildren`. (Refer to the bonus chapter, Case Studies: Maze and Platform Games, in the book's download package for a detailed explanation of how this works).

- The `Card` class that we looked at earlier has a public property called `isBeingDragged`. The next line sets this to true:

```
card.isBeingDragged = true;
```

- It adds a `MOUSE_UP` listener to the card. The listener will call the `mouseUpHandler` when the left mouse button is released over the card.

```
card.addEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
```

Now that you can happily drag the object around the stage, what happens when the mouse is released? The `mouseUpHandler` is called:

```
private function mouseUpHandler(event:MouseEvent):void
{
  var card:Card = event.currentTarget as Card;
  card.stopDrag();
  card.isBeingDragged = false;

  //Remove this mouse listener
  card.removeEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
}
```

The first thing this handler does is find out the name of the object that called it. The name of the object is passed to the handler in its event parameter:

```
private function mouseUpHandler(event:MouseEvent):void
{
```

and you can access it like this:

```
event.currentTarget
```

This will tell you the name of the object that called it, but you have to force AS3.0 to interpret it as a `Card` object. You can do this in one of two ways, using a technique we've used many times in the past called **casting**. This is the format we've used in previous examples to cast a number object as a String so that it can be displayed in a text field:

```
String(anyNumber)
```

However, you can use an alternative syntax using the `as` keyword to do the same thing, like this:

```
anyNumber as String
```

This is the casting format I've used to force the `event.currentTarget` object to be interpreted as a card:

```
event.currentTarget as Card
```

It's the equivalent to this bit of code:

```
Card(event.currentTarget)
```

You can use either format you like; they do exactly the same thing so it's just a matter of personal choice. The code uses this casting technique to copy a reference to the card that's being dragged into a temporary local object called `card`:

```
  var card:Card = event.currentTarget as Card;
```

It then calls the card's `stopDrag` method and sets its `isBeingDragged` property to `false`.

```
  card.stopDrag();
  card.isBeingDragged = false;
```

The `stopDrag` method is what stops the mouse from being able to drag the card around the stage. The last thing the `mouseUpHandler` does is, very importantly, remove itself from the card:

```
card.removeEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
```

And this is really all there is to setting up a basic drag and drop interface. The code does one more interesting thing, however: it snaps the card to the matching square. We'll find out how it does this, but first let's find out a little more about what `event.currentTarget` is and how it works.

Using target or currentTarget properties

In chapter 3 when we were programming buttons for our interactive object, we were able to find out what the mouse was clicking on by using the `event.target` object:

```
event.target
```

The `target` object is the object that triggered the event. It's usually the name of the Sprite object that you've attached a listener to.

In this drag-and-drop example, a property called `currentTarget` is used to find which object the mouse is clicking:

```
event.currentTarget
```

What's the difference between `target` and `currentTarget`?

Figure 13-2 illustrates an example. Imagine that you have an object called `square` that contains a sub-object called `triangle`. You registered the event listener with the square object like this:

```
square.addEventListener...
```

`currentTarget` refers to the object that the event is registered with. In this example, it's the `square` object. If you click and drag anywhere on the square, the whole square object moves, including its `triangle` sub-object.

`target` refers to the actual object that the mouse clicked. If you click the `triangle` sub-object, you can drag only the `triangle`; the parent square doesn't move. (If you click an area of the square that doesn't include the triangle, however, the two objects move together).

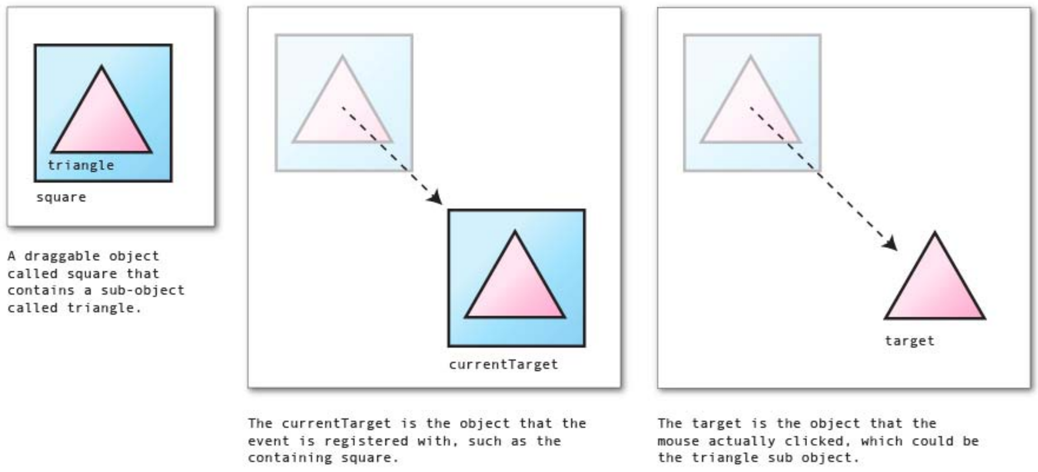

triangle
square
A draggable object called square that contains a sub-object called triangle.
currentTarget
The currentTarget is the object that the event is registered with, such as the containing square.
target
The target is the object that the mouse actually clicked, which could be the triangle sub object.

Figure 13-4. The object you're able to drag depends on whether you referring to the currentTarget or the target.

Draggable Sprite objects also have a property called dropTarget that tells you the object or objects that the mouse was released over. If you add the following code to the mouseUpHandler, it will tell you the names of the objects under the mouse:

if(card.dropTarget != null)

{

trace(card.dropTarget.parent);

trace(card.dropTarget);

}

dropTarget.parent is the main object that the mouse is over.

dropTarget will give you the name of any subobjects that the main object might contain.

Snapping the object to the target

The code that actually snaps the card to its matching square sits in an `ENTER_FRAME` event handler:

```
public function enterFrameHandler(event:Event):void
{
  if (_card.hitTestObject(_match))
  {
    if (!_card.isBeingDragged)
    {
           _card.x = _match.x + 5;
           _card.y = _match.y + 5;
    }
  }
}
```

The code uses `hitTestObject` to check whether the `_card` object is touching the `_match` object. If it is, the code then checks to see whether the card is currently being dragged. `isBeingDragged` will become `false` when the left mouse button is released. You need to check for this so that the card doesn't snap to the matching square before the mouse button is released. If it all seems fine, the card is assigned the exact same x and y position as the matching square, offset by 5 pixels. (The matching square is 10 pixels larger than the card on all sides, so offsetting the card's position by 5 pixels on both axes centers it neatly). The result is that the card appears to snap into place.

Centering the drag object to the mouse

The `startDrag` method has two optional parameters that give you a little more fine control over how your draggable objects behave.

The first parameter is called `lockCenter`. It's a Boolean property that tells AS3.0 whether the object should be centered over the mouse. To use it, just add `true` as an argument in the `startDrag` method, like this:

```
startDrag(true);
```

Try it in the example and you'll notice that the card snaps to the mouse center when you click it.

Better drag and drop

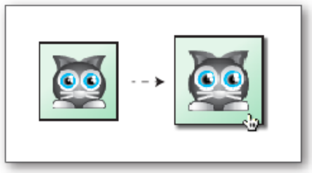
Now that you know how to build a basic drag and drop system, let's refine it a little more by adding some features you'd likely want to use in a game interface:

Make the card bigger and give it a drop shadow when it's clicked. This will make it look like the card is being lifted up slightly over the stage surface.

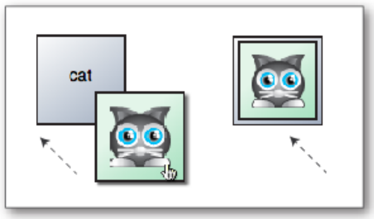
Gradually ease the card into position over the matching square, rather than just snapping it into place.

Confine the area to which an object can be dragged.

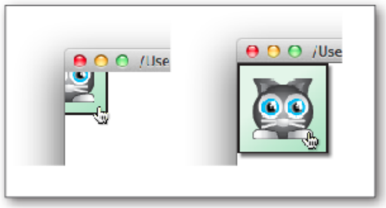
The `BetterDragAndDrop` project in the chapter's source files demonstrates these features, as illustrated in Figure 13-5. It uses exactly the same classes and project structure as the first example, except for some additions to the application class. Here's the entire application class, and I'll explain how it implements all these new features in the pages ahead.



Increase the size and add a drop shadow when the card is being dragged.



Gradually ease the card into position.



Confine the drag area to the stage.

Figure 13-5. Improve the drag and drop interface.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.filters.DropShadowFilter;
  import flash.filters.BitmapFilterQuality;
  import flash.geom.Rectangle;

  [SWF(width="550", height="400",
  backgroundColor="#FFFFFF", frameRate="60")]

  public class BetterDragAndDrop extends Sprite
  {
    //Embed the image for the card foreground
    [Embed(source="../images/cat.png")]
    private var CatImage:Class;

    //Private properties
    private var _catImage:DisplayObject = new CatImage();
    private var _card:Card = new Card(_catImage);
    private var _match:Match = new Match("cat");
    private var _rectangle:Rectangle
      = new Rectangle(0, 0, 465, 315);
    private const EASING:Number = 0.3;

    public function BetterDragAndDrop()
    {
      this.addChild(_card);
      _card.x = 350;
      _card.y = 150;
      _card.buttonMode = true;
      _card.useHandCursor = true;

      this.addChild(_match);
      _match.x = 150;
      _match.y = 150;

      //Add event listeners
      _card.addEventListener
        (MouseEvent.MOUSE_DOWN, mouseDownHandler);
      stage.addEventListener
        (Event.ENTER_FRAME, enterFrameHandler);
    }
    public function enterFrameHandler(event:Event):void
    {
      if (_card.hitTestObject(_match))
      {
        if (!_card.isBeingDragged)
        {
          _card.x += ((_match.x + 5) - _card.x) * EASING;
          _card.y += ((_match.y + 5) - _card.y) * EASING;
        }
      }
    }
    public function addDropShadow(gameObject:Sprite):void
    {
      var shadow:DropShadowFilter = new DropShadowFilter();
      shadow.distance = 4;
      shadow.angle = 45;
```

```
        shadow.strength = 0.6;
        shadow.quality = BitmapFilterQuality.LOW;
        gameObject.filters = [shadow];
      }
      public function changeCardSize(card:Card):void
      {
        if(card.isBeingDragged)
        {
          card.height = 85;
          card.width = 85;
          card.x -= 5;
          card.y -= 5;
        }
        else
        {
          card.height = 75;
          card.width = 75;
          card.x += 5;
          card.y += 5;
        }
      }
      private function mouseDownHandler(event:MouseEvent):void
      {
        var card:Card = event.currentTarget as Card;
        card.startDrag(false, _rectangle);
        setChildIndex(card, numChildren - 1);
        card.isBeingDragged = true;

        //Change the size of the card
        changeCardSize(card);

        //Add a drop shadow filter
        addDropShadow(card);

        card.addEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
      }
      private function mouseUpHandler(event:MouseEvent):void
      {
        var card:Card = event.currentTarget as Card;
        card.stopDrag();
        card.isBeingDragged = false;

        //Change the size of the card
        changeCardSize(card);

        //Remove the drop shadow filter if there is one
        if(card.filters != null)
        {
          card.filters = null;
        }

        //Remove this listener
        card.removeEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
      }
    }
  }
```

Confining the drag area

You might find in some circumstances that you want to confine dragging to a certain area of the stage. The `startDrag` method has a second optional parameter called `bounds` that accepts a Rectangle object as an argument. The Rectangle object defines the area that the object will be confined to.

First, however, you need to actually make a Rectangle object. It's an abstract bit of code that defines an area of a rectangle. Creating one is about as easy or difficult as it is to create a Point object (let's say pretty easy!):

1. Import the Rectangle class with the following `import` statement:

```
import flash.geom.Rectangle;
```

2. Next, declare a Rectangle object in your class. Here's how it's created in the `BetterDragAndDrop` application class:

```
private var _rectangle:Rectangle = new Rectangle(0, 0, 465, 315);
```

The first two arguments are the rectangle's x and y position offset. The third and fourth arguments are its height and width. I came up with those numbers by subtracting the 85 from the width and height of the stage (550 by 400). 85 is the enlarged size of the card when it's being dragged around the stage. The rectangle size above will keep the card confined exactly to the stage area.

3. After the `_rectangle` object is defined, you can use it as the second argument in the `startDrag` method, like this:

```
card.startDrag(false, _rectangle);
```

Because you have to specify the rectangle as the second argument, you also have to supply a first argument, which is where "false" comes from. false refers to the lockCenter parameter that I mentioned in the previous section. Setting it to false tells the program that you don't want the object to snap to the mouse's center point. You can set it to true if you want to; it's entirely up to you.

This code has created an invisible rectangle that confines the card to the area within the stage while it's being dragged.

Making the card bigger when its clicked

To make the card look like it's being lifted slightly off the stage when it's clicked, the code makes it a little bigger and adds a drop shadow effect. A custom method called `changeCardSize` is called both when the mouse is click in the `mouseDownHandler`, and when it's released in the `mouseUpHandler`.

```
changeCardSize(card);
```

The code in this method toggles between two card sizes depending on whether the card is being dragged or not. This lets the same method work for both handlers.

```
public function changeCardSize(card:Card):void
{
  if(card.isBeingDragged)
  {
    card.height = 85;
    card.width = 85;
    card.x -= 5;
    card.y -= 5;
  }
  else
  {
    card.height = 75;
    card.width = 75;
    card.x += 5;
    card.y += 5;
  }
}
```

If the card is being dragged, it makes it larger by 10 pixels. It offsets the card's x and y positions by 5 pixels which makes it look like it's expanding from the center. If the card isn't being drag, the effect is reversed.

You'll notice that the image on the card pixelates slightly when it's enlarged. That's because its pixels are being stretched a bit to compensate for its new size. This is fine for testing in this example, but in a professional project your code should really switch to a higher resolution bitmap image at this point.

Alternatively, use a vector graphic, like a SWF file. Vector graphics can scale to any size without losing resolution.

You can save any of your images as SWF files in Illustrator by using File ~TRA Save for Web & Devices and choosing the SWF format. You can learn how to create SWF images and animations with Flash Professional in chapter in the bonus chapter "Using Flash Professional and Publishing Your Game" in the book's download site.

Adding a drop shadow filter

Filters allow you to apply a special visual effect to Sprite objects. In previous chapters, you learned how to add drop shadow and glow effects to your game graphics in Photoshop and Illustrator. They're great effects of course, but what if you want to add or remove one of them in the middle of a game? AS3.0 allows you to create and apply visual effect filters using code, so you can add them to an object in your game any time you like.

To use a filter, you need to import the filter class for the kind of filter you want to use. Here's how the `DropShadowFilter` class is imported into `BetterDragAndDrop`:

```
import flash.filters.DropShadowFilter;
```

Optionally, if you want to control the quality of the filter, you need to import the `BitmapFilterQuality` class:

```
import flash.filters.BitmapFilterQuality;
```

Being able to control the quality of the filter is very important for games. High-quality filters consume more of the Flash Player's resources to produce, so you generally want the filter's quality setting to be low, especially for any objects that are moving.

Filters are independent objects, just like any other objects you create in AS3.0. To use a filter, you first need to create the filter object, set its properties, and then add it to the object that you want to apply the filter to. Here are the steps to create and apply a drop shadow filter:

4. Create a filter object with the `new` keyword.

```
var shadow:DropShadowFilter = new DropShadowFilter();
```

5. Set any of the filter's properties. Most of the properties for most filters match those that you can set in the Filters pane of the Properties panel. You can set as many or as few of these properties as you need to:

```
shadow.distance = 4;
shadow.angle = 45;
shadow.strength = 0.6;
```

6. Set the optional `quality` property of the filter by applying `LOW`, `MEDIUM`, or `HIGH` from the `BitmapFilterQuality` class:

```
shadow.quality = BitmapFilterQuality.LOW;
```

7. Every Sprite object has a special property called `filters`. Unlike any other property you've looked at before, the `filters` property is actually an array. You apply a filter to an object by adding the filter as an element to the filters array (which is the same way elements were added to an array in Chapter 9). Here's how a filter is added to an object in the BetterDragAndDrop example code:

```
gameObject.filters = [shadow];
```

Adding filters as array elements is very useful because it means that you can apply more than one filter to an object at a time. If you want to also add a bevel or glow filter to the same object, you can create those filter objects and add them to the filters array, like this:

```
gameObject.filters =  [shadow, bevel, glow];
```

To remove all the filters from an object, give the filters array a `null` value, like this:

gameObject.filters = null;

Table 13-1 lists the basic filter classes and their uses.

Table 13-1. Basic filters classes available

Filter class	What it does
BevelFilter	Creates a shallow 3D raised surface effect.
BlurFilter	Blurs the object slightly, giving the impression that it's out of focus or moving quickly.
DropShadowFilter	Casts a shadow.
GlowFilter	Makes it appear as if a light is being cast from underneath the object.
GradientBevelFilter	An enhanced bevel effect that improves its 3D appearance by allowing you to add a gradient color to the bevel.
GradientGlowFilter	An enhanced glow effect that allows you to add a gradient glow to the edges of an object. This filter optionally requires that you import the BitmapFilterType class so you can specify where on the object to apply the filter.

This has been a very brief introduction to AS3.0 filters, but it's enough to get you started. All the filters have a great number of properties that can be set. You can find them all, including more specific information on these filters, in the chapter "Filtering Display Objects" in Adobe's online document, Programming ActionScript 3. Also, there are some specific issues that you need to be aware of if you want to change an object's filter or make specialized adjustments to it while the SWF is running. The "Potential Issues for Working with Filters" subchapter from the "Filtering Display Objects" chapter outlines some of these problems and how to overcome them.

Other advanced filters that have more specialized uses: the color matrix filter, convolution filter, displacement map filter, and shader filter. I won't be discussing these filters in this book, but you should know that they allow for very fine control over color and alpha effects.

The `BetterDragAndDrop` application class adds a drop shadow to the card when in the `mouseDownHandler` by calling the `addDropShadow` method and passing it the `card` object.

```
addDropShadow(card);
```

Here's the method that adds the shadow.

```
public function addDropShadow(gameObject:Sprite):void
{
        var shadow:DropShadowFilter = new DropShadowFilter();
        shadow.distance = 4;
        shadow.angle = 45;
        shadow.strength = 0.6;
        shadow.quality = BitmapFilterQuality.LOW;
        gameObject.filters = [shadow];
}
```

The shadow is removed when the card is dropped by this bit of code in the `mouseDownHandler:`

```
if(card.filters != null)
{
  card.filters = null;
}
```

Let's now see how the coolest effect of this example was achieved: making the card gradually ease into position over the matching square.

Easing

The code that makes the card ease into position over the matching square looks like this:

```
_card.x += (_match.x + 5 - _card.x) * EASING;
_card.y += (_match.y + 5 - _card.y) * EASING;
```

The code adds the easing formula to the x and y positions of the object to make it move to the new position. The card's final position is also offset by 5 pixels so that it centers neatly over the matching square. Easing is a basic technique in game design to move objects, and you can see many more examples in chapter 10..

We've now got the basics of a great little drag and drop engine that we can use to build a simple game. But before we do, let's take another look at loops and how you can use them to plot objects on a grid on the stage.

Using loops to plot objects on a grid

In chapter 9 you saw how you could use a loop within a loop to check collisions between an array of objects against another array of objects. This is the code we used in Amazing Monster Mayhem to check whether the monsters in the `_monsters` array were colliding with the boxes in the `_boxes` array.

```
for(var i:int = 0; i < _monsters.length; i++)
{
        var monster:Monster = _monsters[i];

        for(var j:int = 0; j < _boxes.length; j++)
        {
                Collision.block(monster, _boxes[j]);
        }
}
```

The code checks the first object in the `_monsters` array against all the objects in the `_boxes` array. It then checks the second object in the `_monsters` array against all the boxes, and so on. This is called a **nested for loop** and is one of the most common programming constructs you'll use.

If you want to plot a grid of objects on the stage, there's another standard construct you can use that also uses a nested for loop. You'll find a basic working example in the **`PlottingGrids`** project folder in the chapter's source files. Run it, and you'll see a grid of 9 squares displayed on the stage, as shown in Figure

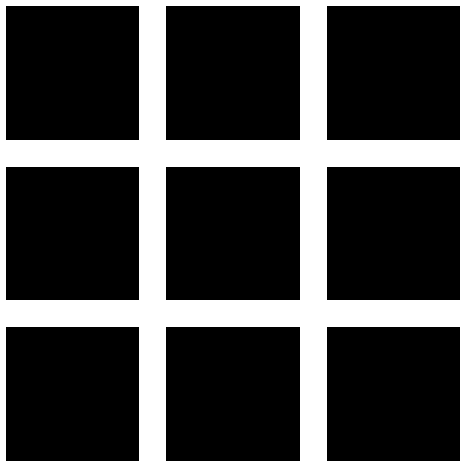


Figure 13-6. Use a nested for loop to plot objects in a grid formation.

Here's the code that does this:

```
package
{
  import flash.display.Sprite;
  import flash.display.Shape;
  public class PlottingGrids extends Sprite
  {
    //The number of columns and rows in the grid
    private const COLUMNS:uint = 3;
    private const ROWS:uint = 3;


    //The space between squares
    private const SPACE:uint = 10;

    public function PlottingGrids()
    {
      for(var columns:int = 0; columns < COLUMNS; columns++)
      {
        for(var rows:int = 0; rows < ROWS; rows++)
        {
          //Make a square
          var square:Shape = new Shape();
          square.graphics.beginFill(0x000);
          square.graphics.drawRect(0, 0, 50, 50);
          square.graphics.endFill();
          addChild(square);

          //Position the square on the grid
          square.x = columns * (square.width + 10);
          square.y = rows * (square.height + 10);
        }
      }
    }
  }
}
```

Two constants are used to represent the numbers of columns and rows we want our grid to have

```
private const COLUMNS:uint = 3;
private const ROWS:uint = 3;
```

An optional constant represents the number pixels of space to leave between squares in the grid.

```
private const SPACE:uint = 10;
```

The code loops through each column, and then loops through reach row of that column

```
for(var columns:int = 0; columns < COLUMNS; columns++)
{
  for(var rows:int = 0; rows < ROWS; rows++)
  {
```

The inner loop creates a black square and adds it to the stage. The x position of the square is found by multiplying its width by the current column number. The y position is found by multiplying its height by the current row number. The optional `SPACE` value of 10 pixels is added to the width and height to leave a bit of padding around the squares.

```
square.x = columns * (square.width + SPACE);
square.y = rows * (square.height + SPACE);
```

This bit of code is a video game staple, and you'll use all the time in games to plot or loop through grids. Don't spend too much time tying your neurons in knots to understand exactly how this works. With time, and the elegant simplicity of how it works will dawn on you eventually. But until then, copy and paste away with glee!

In this example the grid of squares is plotted from the top left corner of the stage. You can start the grid at any position on the stage by using two optional offset variables that tell you where on the stage the grid should appear.

Open the **GridWithOffset** project folder and you'll find another version of this example which starts plotting the rectangles 100 pixels to the left and 100 pixels below the top left corner of the stage, as shown in Figure 13-7.

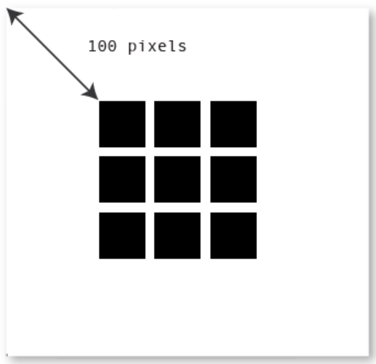


Figure 13-7. Add an optional offset to position the grid anywhere on the stage.

All that's needed to achieve this are two variables that determine the amount of offset:

```
var offset_X:uint = 100;
var offset_Y:uint = 100;

for(var columns:int = 0; columns < COLUMNS; columns++)
{
        for(var rows:int = 0; rows < ROWS; rows++)
        {
                //Make a square...

                //Position the square on the grid
                square.x = offset_X + (columns * (square.width + SPACE));
                square.y = offset_Y + (rows * (square.height + SPACE));
        }
}
```

Once you understand how this is working, there are all sorts of things you can use these nested `for` loops for. One of the most useful is to loop through all the contents of a 2D array. Let's find out how next

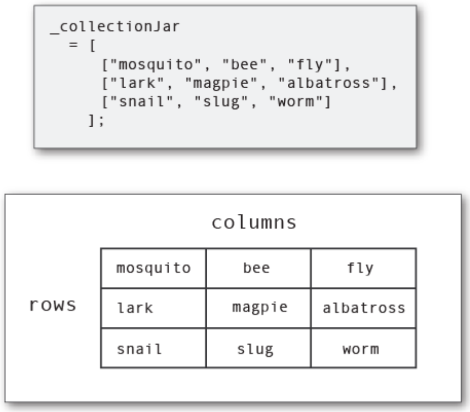
Looping through a 2D array

While we're on the topic, there's only last thing you need know about are nested `for` loops. Because you can use a nested `for` loop to loop through a grid of items, you can also use it to access all the elements in a 2D array. This isn't directly related to what you need to know for the examples for the rest of the chapter, so feel free to jump ahead to the next section if you're tired of all this technical stuff! But you'll certainly need to know how to do this at some point in your game design career, so come back to this section if you need to know how.

Earlier in the book, we use a 2D array called `_collectionJar` that looked like this:

```
_collectionJar
  = [
          ["mosquito", "bee", "fly"],
          ["lark", "magpie", "albatross"],
          ["snail", "slug", "worm"]
    ];
```

A 2D array is actually a grid of data, as you can see in Figure 13-8.



```
_collectionJar
  = [
      ["mosquito", "bee", "fly"],
      ["lark", "magpie", "albatross"],
      ["snail", "slug", "worm"]
    ];
```

columns

rows

mosquito	bee	fly
lark	magpie	albatross
snail	slug	worm

Figure 13-8. A 2D array represents a grid of information.

There are three columns, and three rows. You can use a nested `for` loop to loop through this data if you need to use it in a game. You'll find a project folder called **`ArrayLoop2D`** that shows how you can do this. Here's the application class.

```
package
{
  import flash.display.Sprite;

  public class ArrayLoop2D extends Sprite
  {
    //The array
    private var _collectionJar:Array = new Array();

    public function ArrayLoop2D()
    {
      _collectionJar
        = [
            ["mosquito", "bee", "fly"],
            ["lark", "magpie", "albatross"],
            ["snail", "slug", "worm"]
          ];

      //Find the number of rows and columns
      var maxRows:uint = _collectionJar.length;
      var maxColumns:uint = _collectionJar[0].length;

      for(var column:int = 0; column < maxColumns; column++)
      {
        trace("---");

        for(var row:int = 0; row < maxRows; row++)
        {
         //Organize into rows
          trace(_collectionJar[column][row]);

          //Organize into columns
          //trace(_collectionJar[row][column]);
        }
      }
    }
  }
}
```

You'll see that the trace output neatly separates each row with three dashes:

```
---
mosquito
bee
fly
---
lark
magpie
albatross
---
snail
slug
worm
```

However, by making one small change to this code, you can organize the data into columns. Just switch the order of rows and columns in the `trace` statement:

```
trace(_collectionJar[row][column]);
```

Recompile the project, and you'll see that the words between the three dashes tell you the contents of each column.

```
---
mosquito
lark
snail
---
bee
magpie
slug
---
fly
albatross
worm
```

This example uses the array's `length` property to work out the number of rows and columns in the grid.

```
var maxRows = _collectionJar.length;
var maxColumns = _collectionJar[0].length;
```

The `_collectionJar` array has three elements, so that represent the number of rows. The first array that it contains also contains three elements, so that represents the number of columns.

Keep this example in mind if you ever need to read data from a 2D array to assign to any of your game objects. This is how you access it.

Case study: Matching game

Now that you know how to make a drag-and-drop engine, and you know how to use a nested `for` loop to plot objects in a grid, we can fuse these two techniques to build a simple matching game. You'll find it in the **`MatchingGame`** project folder in the chapter's source files. There are nine cards on the stage and nine matching squares. You can drag any of the cards anywhere you like over the stage, but they'll only ever snap to their matching squares. Figure 13-9 shows what the game looks like. This is a simple example, but the techniques used to build it are at the heart of what you need to know to build a grid based logic game, and ultimately even something as complex as a board game like checkers or chess. All you'll need to add is the game logic.

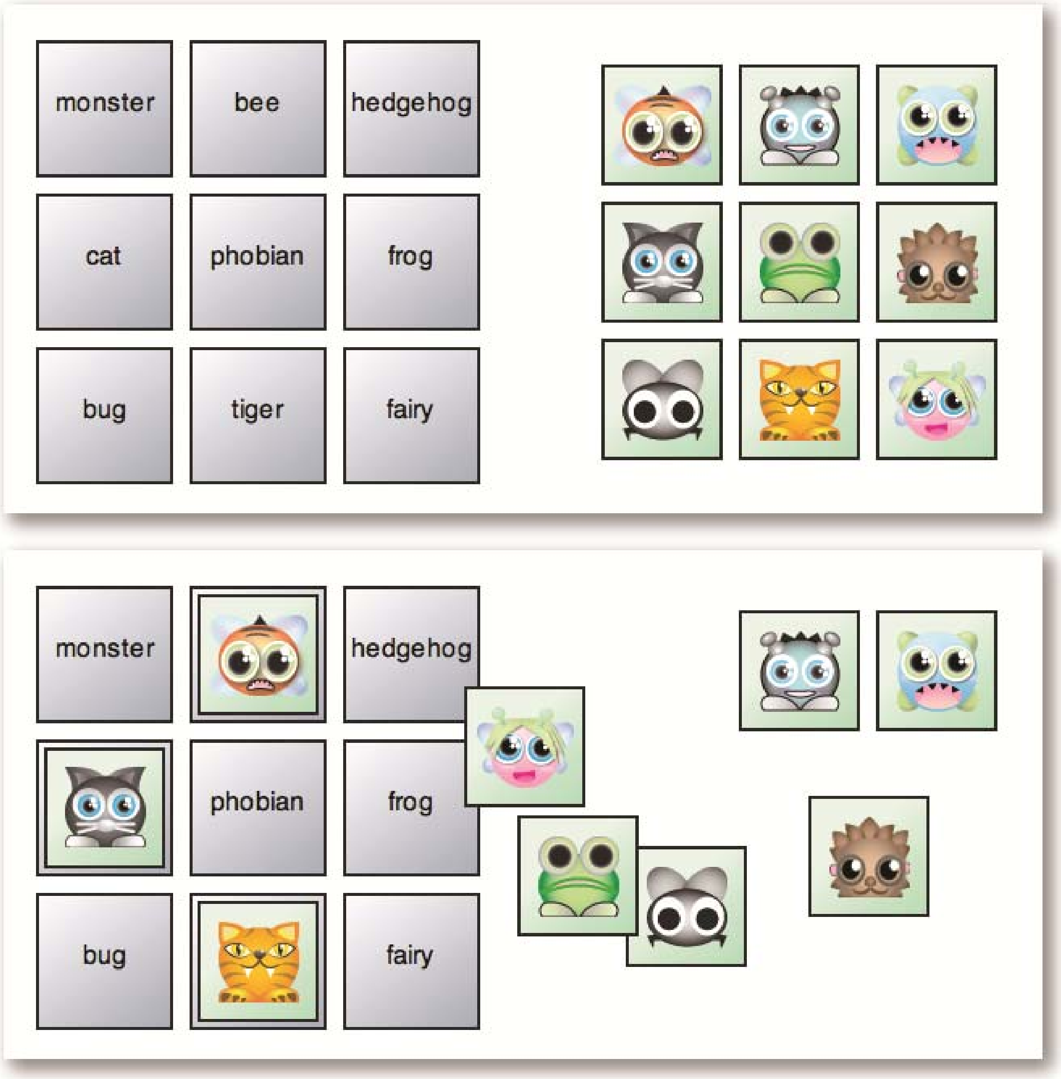


Figure 13-9. Drag the cards into their matching squares.

A unique feature of this game is that the cards are shuffled each time the game is played. Take a closer look at Figure 13-9 and you'll notice that cards are in a different order than their matching square. They'll be completely different again every time the game is run. This was done using very simple code that randomizes the order of the elements in the array that stores the card names, and we'll see how ahead.

Here's the entire **`MatchingGame`** application class, and I'll walk you through all the details of how the game was put together in the pages that follow.

```
package
{
  import flash.display.DisplayObject;
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.events.MouseEvent;
  import flash.filters.BitmapFilterQuality;
  import flash.filters.DropShadowFilter;
  import flash.geom.Rectangle;

  [SWF(width="700", height="400",
  backgroundColor="#FFFFFF", frameRate="60")]

  public class MatchingGame extends Sprite
  {
    //Embed the images for the card foregrounds
    [Embed(source="../images/cat.png")]
    private var CatImage:Class;
```

```
[Embed(source="../images/frog.png")]
private var FrogImage:Class;
[Embed(source="../images/phobian.png")]
private var PhobianImage:Class;
[Embed(source="../images/bee.png")]
private var BeeImage:Class;
[Embed(source="../images/bug.png")]
private var BugImage:Class;
[Embed(source="../images/monster.png")]
private var MonsterImage:Class;
[Embed(source="../images/hedgehog.png")]
private var HedgehogImage:Class;
[Embed(source="../images/fairy.png")]
private var FairyImage:Class;
[Embed(source="../images/tiger.png")]
private var TigerImage:Class;

//Private properties
private var cat:DisplayObject = new CatImage();
private var frog:DisplayObject = new FrogImage();
private var phobian:DisplayObject = new PhobianImage();
private var bee:DisplayObject = new BeeImage();
private var bug:DisplayObject = new BugImage();
private var monster:DisplayObject = new MonsterImage();
private var hedgehog:DisplayObject = new HedgehogImage();
private var tiger:DisplayObject = new TigerImage();
private var fairy:DisplayObject = new FairyImage();

//Constants to store the columns and rows for the grid layout
private const COLUMNS:uint = 3;
private const ROWS:uint = 3;
private const SPACE:uint = 10;

//An array to store the names of all the cards
private var _cardNames:Array = new Array()

//Arrays to store the matches and cards
private var _matches:Array = new Array();
private var _cards:Array = new Array();

private var _match:Match = new Match("cat");
private var _rectangle:Rectangle
  = new Rectangle(0, 0, 615, 315);
private const EASING:Number = 0.3;

public function MatchingGame()
{
  //Initialize the card names array
  _cardNames =
    [
      "cat",
      "frog",
      "phobian",
           "bee",
      "bug",
      "monster",
      "hedgehog",
      "tiger",
      "fairy"
     ];


  //MAKE THE MATCH OBJECTS
```

```
//Shuffle the cards
_cardNames.sort(shuffleArray);

//A temporary variable to store the
//current match object
var match:Match;

for(var i:int = 0; i < _cardNames.length; i++)
{
  match = new Match(_cardNames[i]);
  _matches.push(match);
}

//A variable to track the current index number
//of the match object being accessed in the loop
var matchCounter:uint = 0;

//The x and y start positions of
//the match grid on the stage
var matchGridStart_X:uint = 50;
var matchGridStart_Y:uint = 50;

for(var columns:int = 0; columns < COLUMNS; columns++)
{
  for(var rows:int = 0; rows < ROWS; rows++)
  {
    //A temporary object to reference the
    //current match object in the loop
    match = _matches[matchCounter]
    addChild(match);

    //Position the match object in a square
    //on the grid
    match.x
      = matchGridStart_X
      + (columns * (match.width + SPACE));

    match.y
      = matchGridStart_Y
      + (rows * (match.height + SPACE));

    matchCounter++;
  }
}

//MAKE THE CARD OBJECTS

//Shuffle the cards again
_cardNames.sort(shuffleArray);

//A temporary variable to store the
//current card object
var card:Card;
for(var j:int = 0; j < _cardNames.length; j++)
{
  card = new Card(this[_cardNames[j]]);
  card.addEventListener
    (MouseEvent.MOUSE_DOWN, mouseDownHandler);
  card.name = _cardNames[j];
  _cards.push(card);
}

//A variable to track the current index number
//of the card object being accessed in the loop
```

```
  var cardCounter:uint = 0;

  //The x and y start positions of
  //the card grid on the stage
  var cardGridStart_X:uint = 400;
  var cardGridStart_Y:uint = 65;

  for(columns = 0; columns < COLUMNS; columns++)
  {
    for(rows = 0; rows < ROWS; rows++)
    {
      //A temporary object to reference the
      //current card object in the loop
      card = _cards[cardCounter];
      addChild(card);

      //Position the match object in a square
      //on the grid
      card.x
        = cardGridStart_X
        + (columns * (card.width + SPACE));

      card.y
        = cardGridStart_Y
        + (rows * (card.height + SPACE));

      cardCounter++;
    }
  }

  stage.addEventListener(Event.ENTER_FRAME, enterFrameHandler);
}
public function enterFrameHandler(event:Event):void
{
  for(var i:int = 0; i < _cards.length; i++)
  {
    var card:Card = _cards[i];

    for(var j:int = 0; j < _matches.length; j++)
         {
      var match:Match = _matches[j];

      if (card.hitTestObject(match)
      && card.name == match.name)
      {
        if (!card.isBeingDragged)
        {
          card.x += (match.x + 5 - card.x) * EASING;
          card.y += (match.y + 5 - card.y) * EASING;
        }
      }
    }
  }
}
private function shuffleArray
  (valueOne:String, valueTwo:String):int
{
  if (Math.random() < 0.5)
  {
    return -1;
  }
  else
  {
    return 1;
```

```
      }
    }
    public function addDropShadow(gameObject:Sprite):void
    {
      var shadow:DropShadowFilter = new DropShadowFilter();
      shadow.distance = 4;
      shadow.angle = 45;
      shadow.strength = 0.6;
      shadow.quality = BitmapFilterQuality.LOW;
      gameObject.filters = [shadow];
    }
    public function changeCardSize(card:Card):void
    {
      if(card.isBeingDragged)
      {
        card.height = 85;
        card.width = 85;
        card.x -= 5;
        card.y -= 5;
      }
      else
      {
        card.height = 75;
        card.width = 75;
        card.x += 5;
        card.y += 5;
      }
    }
    private function mouseDownHandler(event:MouseEvent):void
    {
      var card:Card = event.currentTarget as Card;
      card.startDrag(false, _rectangle);
      setChildIndex(card, numChildren - 1);
      card.isBeingDragged = true;
      //Change the size of the card
      changeCardSize(card);

      //Add a drop shadow filter
      addDropShadow(card);

      card.addEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
    }
    private function mouseUpHandler(event:MouseEvent):void
    {
      var card:Card = event.currentTarget as Card;
      card.stopDrag();
      card.isBeingDragged = false;

      //Change the size of the card
      changeCardSize(card);

      //Remove the drop shadow filter if there is one
      if(card.filters != null)
      {
        card.filters = null;
      }
      //Remove this listener
      card.removeEventListener(MouseEvent.MOUSE_UP, mouseUpHandler);
    }
  }
}
```

A quick glance at this code will tell you that the core of it is identical to the previous drag and drop examples we looked at. The only difference is the way in which we're creating and adding objects to the stage. Let's find out how this is done.

Making the cards and their matching squares

As you can see from Figure 13-9, there are nine customized cards on the stage. There's no way you'll want to go to all the trouble of making a unique class for each card. Fortunately, because we're using composition to customize a single `Card` class, we don't have to. We can just embed nine images into the game and then use them to make nine instances of the card class, each with a custom image.

Take a careful look at the instance names of the image objects, which I've highlighted below:

```
private var cat:DisplayObject = new CatImage();
private var frog:DisplayObject = new FrogImage();
private var phobian:DisplayObject = new PhobianImage();
private var bee:DisplayObject = new BeeImage();
private var bug:DisplayObject = new BugImage();
private var monster:DisplayObject = new MonsterImage();
private var hedgehog:DisplayObject = new HedgehogImage();
private var tiger:DisplayObject = new TigerImage();
private var fairy:DisplayObject = new FairyImage();
```

Now take a look at the names in the `_cardNames` array:

```
_cardNames =
  [
    "cat",
    "frog",
    "phobian",
    "bee",
    "bug",
    "monster",
    "hedgehog",
    "tiger",
    "fairy"
  ];
```

They're *exactly* the same, and this is vital for this system to work. This array is used to make both the cards and the matching squares. The names in the array have to be the same as the image object names so that the correct image can be loaded into each card. The matching squares also use this array to display the name of the image they're matched with.

Here's the code that makes the cards:

```
var card:Card;
for(var j:int = 0; j < _cardNames.length; j++)
{
        card = new Card(this[_cardNames[j]]);
        card.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownHandler);
        card.name = _cardNames[j];
        _cards.push(card);
}
```

A loop repeats for as many names as there are in the `_cardNames` array. It creates a new card with this line of code. Take a close look at the argument used to make the card:

```
card = new Card(this[_cardNames[j]]);
```

Remember that Card objects are created by sending the `Card` class an image object. However, the words in the **_cardNames** array are Strings, not objects. You can force AS3.0 to interpret a string as an object by surrounding it by square brackets and preceding it with the keyword `this`:

```
this["anyStringThatRepresentsAnObject"]
```

AS3.0 will then know that you want to the string to represent an object.

In this example, the first time the loop runs it will select the first element in the `_cardNames` array and use it to make a card object, like this:

```
card = new Card("cat");
```

But of course, you can't send the Card class a String, it has to be an object. So you can force the string to become an object like this:

```
this["cat"]
```

This is what you end up with:

```
cat
```

What is the `cat` object? It's the image of the cat that we embedded back in the class definition:

```
private var cat:DisplayObject = new CatImage();
```

And that's exactly what you need to make a card object. So the final code that AS3.0 sees after all of this looks like this:

```
card = new Card(cat);
```

It sends the `cat` image to the `Card` class, and we end up with a card object that displays the image.

If you ever need to, you can use this same syntax to access an object's properties. Here's how you could access the cat object's x and y properties by accessing the string "cat" in the _cardNames array:

this[_cardNames[j]].x

this[_cardNames[j]].y

On the first loop, this would be interpreted as:

cat.x

cat.y

The next thing the loop does is add the `MOUSE_DOWN` listener to the card, which is identical to the listener we added in the previous example.

```
card.addEventListener(MouseEvent.MOUSE_DOWN, mouseDownHandler);
```

The code then assigns the current name from the `_cardNames` array to the card's `name` property.

```
card.name = _cardNames[j];
```

The card is going to share exactly the same name as its matching square, and we'll use this feature in the game to find out whether a card is on the correct square.

Finally, the loop pushes the newly minted card into the `_cards` array so that we can access it later.

```
_cards.push(card);
```

Repeat these steps nine times, and you end up with nine customized cards, each displaying one of the embedded images, and each with a unique `name` property that matches its image.

The matching squares are made in the same way, using exactly the same `_cardNames` array.

```
var match:Match;
for(var i:int = 0; i < _cardNames.length; i++)
{
        match = new Match(_cardNames[i]);
        _matches.push(match);
}
```

Remember that to make a Match object, you have to send the class a string that represents the name of the object you want it to match. Here's the code that does this:

```
match = new Match(_cardNames[i]);
```

The elements in the `_cardNames` array are already strings, so you can use them as-is to send to the Match class. The new `match` object is then pushed in to the `_matches` array for later use.

```
_matches.push(match);
```

This is a wonderfully efficient system because we only need one array to create all nine cards and all nine matching squares. We could make hundreds of cards like this if we wanted to, and our code wouldn't become any more complex or more difficult to manage. That's the beauty of using composition to make game objects.

Displaying the cards and matching squares

The code that displays the cards and squares follows the same format for plotting objects on a grid that we looked at in previous examples. However, there's one important difference: the loop needs to know which card in the **_cards** array it should display. To do this, it uses a special loop counter variable to count the number of loops, and then uses that number to access the next card in the **_cards** array. The loop counter is initialized to zero before the loop starts:

```
var cardCounter:uint = 0;
```

`cardCounter` is then used to access the next card in the `_cards` array so that it can be displayed and positioned on the stage.

```
card = _cards[cardCounter];
addChild(card);
```

`cardCounter` is then updated by one when the new card has been added and displayed.

```
cardCounter++;
```

Here's all the code that does this, and I've highlighted the `cardCounter` variable so that you can follow how and where it's used to access the next correct card object.

```
//A variable to track the current index number
//of the card object being accessed in the loop
var cardCounter:uint = 0;

//The x and y start positions of
//the card grid on the stage
var cardGridStart_X:uint = 400;
var cardGridStart_Y:uint = 65;

for(columns = 0; columns < COLUMNS; columns++)
{
  for(rows = 0; rows < ROWS; rows++)
  {
    card = _cards[cardCounter];
    addChild(card);
    card.x  = cardGridStart_X + (columns * (card.width + SPACE));
         card.y = cardGridStart_Y + (rows * (card.height + SPACE));
    cardCounter++;
  }
}
```

The code that displays the matching squares follows exactly this same format.

Shuffling the cards

To make sure that the cards are in a different order than the matching squares, the `_cardNames` array is shuffled before each of these sets of objects are made. Here's the line of code that shuffles the array:

```
_cardNames.sort(shuffleArray);
```

It uses a method called `sort`, which is a built-in method that array objects can use. It sends the `sort` method one argument, which is the return value of the custom `shuffleArray` method. Here's the `shuffleArray` method that you'll find in the `MatchingGame` application class:

```
private function shuffleArray
  (valueOne:String, valueTwo:String):int
{
        if (Math.random() < 0.5)
        {
                return -1;
        }
        else
        {
                return 1;
        }
}
```

This is a useful and well worn trick and I'll briefly describe how it works.

Array objects have a method called `sort` that sorts the elements into alphabetical order. You can alphabetize the `_cardNames` array like this:

```
_cardNames.sort();
```

This will sort the names into this order:

```
bee,bug,cat,fairy,frog,hedgehog,monster,phobian,tiger
```

You can include arguments in the sort method that will sort the array in other ways. The argument `Array.DESCENDING` will sort the names in reverse alphabetical order.

```
_cardNames.sort(Array.DESCENDING);
```

This produces the following result:

```
tiger,phobian,monster,hedgehog,frog,fairy,cat,bug,bee
```

(You can sort arrays of numbers in exactly the same way if you need to.)

The `sort` method also lets you submit a `method` as an argument. In this case, we're submitting the `shuffleArray` method as the argument:

```
_cardNames.sort(shuffleArray);
```

Yes, the argument contains a *method*, not a variable or object. This is something you'll probably never see again in your programming career, or ever need to do yourself, so look on in wonder, and be amazed!

The method that you submit to `sort` has to have two parameters that match the type of thing in the array. (They're strings in the case):

```
private function shuffleArray(valueOne:String, valueTwo:String):int
```

The method also has to return the values 1 or -1. That happens to be exactly what the shuffleArray method returns:

```
if (Math.random() < 0.5)
{
        return -1;
}
else
{
        return 1;

        }
```

It randomly returns 1 or -1 50% of the time. AS3.0's sort method uses these numbers them to rank the order of objects in the array. Because the ranking is random, the array's order becomes random as well.

You'll never, ever need to create a custom sort method for any other reason except for randomizing an array like this, so copy this method into your `GameUtilities` class and use it whenever you need to shuffle an array's contents.

Figuring out if the card is on the correct matching square

To find out if the card is on the right square, the `enterFrameHandler` loops though all the cards in the `_cards` array and checks them for collisions against all the objects in the `_matches` array.

```
public function enterFrameHandler(event:Event):void
{
  for(var i:int = 0; i < _cards.length; i++)
  {
    var card:Card = _cards[i];

    for(var j:int = 0; j < _matches.length; j++)
    {
      var match:Match = _matches[j];

      if (card.hitTestObject(match)
      && card.name == match.name)
      {
        if (!card.isBeingDragged)
        {
          card.x += (match.x + 5 - card.x) * EASING;
          card.y += (match.y + 5 - card.y) * EASING;
        }
      }
    }
  }
}
```

If the objects are touching and they have the same name, then you know the card is on the correct square.

```
if (card.hitTestObject(match)
&& card.name == match.name)
{...
```

Remember that each card and matching square are assigned exactly the same names from the `_cardNames` array when they are created. This means that each card will always have one match.

More uses for composition

You now know how to make a drag and drop game interface, plot objects on a grid, and read the contents of a 2D array. And very importantly, you've learnt how to use composition to make lots and lots of objects from only one class. With a little bit of thought, you should realize how much less code you'll need to write for your games if you use composition wisely.

In previous chapters in this book, each unique game object had its own class. We had to write a class for each object, which was quite a bit of work. But if you plan carefully, you might find that many or most of your objects could share the same class. You could create one class called `GameObject` that contains all the important game object properties, like `vx`, `vy`, and `timesHit`, that most of your game objects share. You could then use composition to create all your objects from this one class. Just send the class a unique image and give it a unique name when you create it, just like we created the cards in the matching game. You could store all the information about these objects in arrays, and loop through the arrays to make your custom game objects from this one class. Do you see the potential here for making your game really quickly and efficiently with just a little bit of code? You'll see exactly how to do this in the examples ahead.

Composition and inheritance are the two complementary cornerstones of Object Oriented Programming. Inheritance is when you make a class from another class using the "extends" keyword, like this:

*public class Card **extends** Sprite*

{…

The lets the new class inherit all the properties and methods from the class it's extending. In this case, the Card class will inherit all of the Sprite classes properties and methods, such as x, y, visible, and addChild.

Composition is when you create an object by sending a class a custom argument. The class uses this argument to make a customized object, like this:

private var _match:Match = new Match("cat");

There are advantages and disadvantages to both inheritance and composition, and with practice and experimentation in your own games you'll discover for yourself when it's best to use one system over the other.