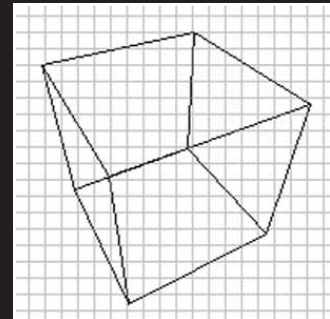


## 3 CUSTOM TOOLS



```
function configureTool()
{
    theTool = fl.tools.activateTool(
    theTool.setToolName("Cube3D");
    theTool.setIcon("Cube3D.png");
    theTool.setMenuString("3D Cube");
    theTool.setToolTip("3D Cube");
    theTool.setOptionsFile("Cube3DOptions.txt");

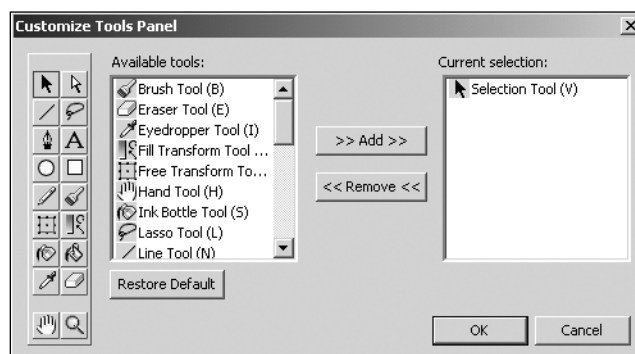
    points = new Array();
    xp = new Array();
    yp = new Array();
}
```



Personally, we find customized tools to be one of the most exciting portions of the JavaScript API. They have a rich architecture that is fun to work with from a programming viewpoint, and the results can be quite useful to both developers and designers. Let's take a look at what they are and how they work. As you go through this chapter, we urge you to be patient. Creating the simplest tool requires a lot of steps, and this is all brand new ground, above and beyond what we covered on custom commands. There are quite a few concepts to study before you can even start coding a tool, and if you jump ahead too far, you'll soon be lost.

In all versions of Flash so far, the toolbar has been sitting, fairly innocuously, over on one side of the screen, happily minding its own business. There have been a few changes over the years, but all in all, it's been pretty much the same old toolbar. You're no doubt pretty familiar with what's on it by now, and up until to now, what you got with Flash was what you were stuck with.

Well, with Flash MX 2004, Macromedia has removed the padlock and let you in to explore and play around. The first thing that might offer a clue to this newfound freedom is in the Edit menu, which has a new item, *Customize Tools Panel*. Go ahead and check it out; you'll get a panel something like the one in Figure 3-1.



**Figure 3-1.** The Customize Tools Panel

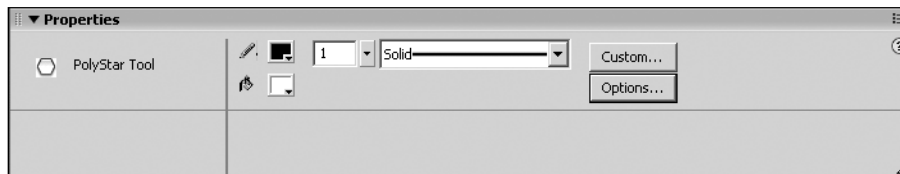
With that, you can add, remove, or rearrange tools on your toolbar to your heart's content, including brand new tools that never existed before—tools that don't exist because *you* haven't written them yet. But you'll remedy that soon. We'll get more into the Customize Tools Panel a little later. That was just a sneak peek to whet your appetite!

Tools are written in JSFL, just as commands are, and can also use the XML to UI dialog boxes. But tools are miles ahead of commands, both in complexity and in power. To get an outside concept of what tools can do, check out the PolyStar tool on the toolbar.

Ah, but first you have to find it. Taking the hint from many other programs with toolbars, Flash now has drop-down tools. If you look at the Rectangle tool, you'll see a small arrow in the lower-right corner. That's a hint that it contains a drop-down toolbar. If you simply click the button for this tool, you'll get the Rectangle tool. But click and hold for a second, and you'll see the PolyStar tool drop down.

Unfortunately, you can't actually add rows or columns to the toolbar, you just have two columns to work with. There are 16 buttons labeled Tools, then 2 labeled View. Below that you have the Colors and Options areas. You can customize any of the buttons in the Tool or View sections. Recognizing that developers will probably be building and sharing many custom tools, Macromedia implemented the drop-down tools in order to let you cram as much as possible in the limited real estate of the toolbar.

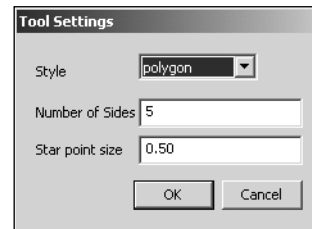
If you choose PolyStar and start drawing on the stage, you'll see a pentagon shape appear where you clicked. It will grow, shrink, and rotate as you move the mouse. When you release the button, that pentagon shape will be permanently drawn onstage in whatever colors are selected. But you might also notice something else if you happen to have the Properties panel open when you select the PolyStar tool. It's subtle, but there is an additional button in there, labeled Options (see Figure 3-2).



**Figure 3-2.** The Properties panel. Note the Options button.

Click that button, and you get a dialog box like the one in Figure 3-3.

You can now choose between a polygon and a star (now you know where they got the name PolyStar!), how many sides, and the star's point size. You might immediately guess that this dialog box was made in XML to UI. Not quite! Although you can of course create additional dialog boxes using XML to UI (see Chapter 5 for more details), you're given one "free" dialog box that gets linked to the Options button in the Properties panel. You just specify the parameters you want, their types, and their values. You still have to write it in an XML file, but it's a bit different to the way you use XML to UI. We'll give you a look at how to write that XML shortly.



**Figure 3-3.** The PolyStar tool settings

Now that you've seen a custom tool in action, you're probably ready to jump in and make one of your own. Don't worry, you'll be doing just that soon enough. But first you need to learn a little about the structure of custom tool files. As we said, they are quite a bit more complex than a command. So this initial examination will be worth it in the long run.

## Event-Based Code

If you look over the command files you created in Chapter 2, you'll see that they're pretty much linear code. You run the file and Flash starts with the first line, executes it, and moves to the next. Of course, when Flash encounters a function, it will jump to the point where the function is defined and run through that code, then back to where it left off.

When it hits the last executable line, that's it, show's over, command is done. If you want it to do more, you have to run it again.

Furthermore, short of an XML to UI dialog box, there is no interaction with the user. If you set some parameters and click OK, the command does its thing and presents you the results. There is no preview of what you're going to get. You simply have to click OK and hope for the best. Imagine trying to make something like a star shape with commands and dialog boxes alone. You'd have to guess at the size, location, number of points, point size, and rotation, then see how it looks. No good? Undo and try some slightly different parameters. Try again. Closer? Try again. Compare that to the ease of using the PolyStar tool.

Don't get us wrong, commands are pretty smart—but soon you'll see just how much more flexible tools are. This is largely due to their event-based structure. So let's discuss that for a moment.

If you've done much with Flash MX ActionScript, you probably have a pretty good idea of what events are and how an event-based program works. In ActionScript you would define various functions as handlers for various events. For example, you would have a function that runs each time the mouse button is clicked. This would be set as the `onMouseDown` handler. Or most commonly, you write an `onEnterFrame` handler function, which responds each time Flash enters a new frame.

Unlike a linear program that starts right in on the first statement and works its way through, a fully event-based program doesn't really do *anything* when you first run it. It just kind of sits there, waiting for some event to occur—a mouse click or movement, a key press, some content finishing loading, or whatever.

And that's basically how tool scripts work. You may define some variables at the beginning of the script, but everything else is all enclosed within functions. When Flash starts up, it executes all the JSFL files in the tool directory, even if they aren't on the toolbar! At this point, a specific function is executed to configure the tool. Then, each tool lies dormant, waiting to be activated. Of course, if a tool isn't on the toolbar, there is no way to activate it, but it sits there waiting anyway.

Once it is activated, it executes another function. And then it waits for something else to happen, such as a mouse action.

The key concept to remember is that tools shouldn't have any code in them that isn't contained in functions. Several of these functions are specifically named, and conform to specific events that occur. When a particular event occurs, Flash will look in your script for a function with the correct name, and run that function. For example, if you define a `mouseDown` function, it will automatically be called whenever the user presses the mouse button.

### Standard Tool Functions

Let's run through the functions that you'll most likely define in your tools. For each one, we'll look at its name, when it executes, and what you should probably do inside of it.

First you have the administrative type functions, which handle setting things up and breaking things down:

- `configureTool`
- `activate`
- `deactivate`
- `notifySettingsChanged`
- `setCursor`

Then the functions for handling keyboard input:

- `keyUp`
- `keyDown`

And finally the mouse-handling functions:

- `mouseUp`
- `mouseDown`
- `mouseMove`
- `mouseDb1C1k`

We'll focus on each of these functions in the following sections.

### **configureTool**

This fires when Flash is starting up. As part of all the actions it takes to set up the authoring environment, it executes each JSFL file it finds in the `tools` folder. This is the reason you don't want any code outside of functions, because that code will be executed as soon as Flash loads, regardless of whether the tool is even used on the toolbar. As it executes each tool file, it also looks for a `configureTool` function and runs it. Here you would put any code to initialize variables, etc.—anything that the tool needs to be set up and ready to run. There are a few statements that *need* to be in here, and these will set the tools icon, name, and so on.

### **activate**

This function runs when you choose the tool on the toolbar. This is where you do any last minute setup for the tool. Where you do the setup—`configureTool` or `activate`—is up to you. You might want to do the bulk of the work in `configureTool` so that the tool is mostly ready to run when you select it. In `activate`, you'll probably minimally grab a reference to the `activeTool` property here, with the line `var curr_toolObj = fl.tools.activeTool`, and then get any properties off the Tool Settings dialog box after clicking the Options button in the Properties panel, if you chose to include one. You'll see more of this later.

### **deactivate**

As you probably guessed, this fires when you deselect the tool, usually by selecting another one. Often, you don't need to do anything here, but if you have any cleanup to do, this would be the place to do it.

### **notifySettingsChanged**

As mentioned earlier, you have the option of having a Tool Settings dialog box. If you created one, and the user uses it to change some setting, this function will fire as soon as that user clicks OK. In this function you need to manually grab these new settings and incorporate them into your code.

### **setCursor**

Two functions have this name. One is `fl.tools.setCursor(cursorNumber)`, which allows you to manually change the appearance of the cursor. *This* `setCursor`, though, is different. It is an event-handling function. From time to time, Flash or the operating system may need to change the cursor to indicate various things, such as changing to an hourglass when the CPU is busy, or changing to standard icons when you move the mouse back over the desktop or other application. When they decide to hand control of the cursor back over to your script, Flash calls the `setCursor` function, if you've included it. Here, you can do whatever you want. Generally speaking this consists of setting the cursor you want your tool to have. Again, more on this later.

### **keyUp**

This is pretty obvious. It fires when a key that has been pressed is released. While tools usually rely on the mouse, it is possible to build in some keyboard interaction.

### **keyDown**

Yes, this fires when any key on the keyboard is pressed down.

### **mouseUp**

Again, pretty self-explanatory. When the mouse button is pressed, and then you release it, this function is run, if you've defined it.

### **mouseDown**

Fires when you press the mouse button. The common way to write these is to have Flash enter temporary drawing mode when the mouse is pressed. A temporary drawing is drawn to a special object called the drawing layer. You can see this in the PolyStar as well as other tools. When you draw with the tool, a kind of faint outline of the shape is visible. Then, when the mouse button is released (`mouseUp`), the actual shape is drawn to the stage, using whatever stroke and fill settings are active.

### **mouseMove**

You guessed it, this fires when you move the mouse. Usually, this is where your temporary drawing will occur. When the mouse is moved, some shape is drawn, larger or smaller, or rotated differently. When it is moved again, the earlier shape is gone and a newer version, based on current mouse position, is redrawn.

### **mouseDblClk**

Another very obvious one, this function fires when the mouse is double-clicked. As you saw in the previous chapter, a double click will actually generate two events: first, a `mouseDown` occurs for the first click, and then if the second click is close enough to the first, a `mouseDblClk` event will fire.

OK, those are all the basic event-handling functions. You could create a pretty basic template for a tool by simply putting all of these functions into a new JSFL file in the following format:

```
function configureTool(){  
}
```

Of course, you don't need to handle *all* of these functions. If you aren't going to be handling keyboard input or double clicks, you can either leave those functions empty, or leave them out of the file altogether.

3

## Setting Up a New Tool

We know, we know, you're probably dying to dive in and start creating a new tool by now. But remember, this is more complicated than just writing a command. You have quite a bit of work to do before you can even have a tool show up in the toolbar.

First of all, if you haven't already done so, take a look in the Tools directory in your Flash configuration directory. As you saw in Chapter 2, this is found within the Configuration directory (C:\Documents and Settings\\Local Settings\Application Data\Macromedia\Flash MX 2004\en\Configuration\) on Windows systems. You'll see a few files there already for the PolyStar tool. There's PolyStar.jsfl, PolyStar.png, and PolyStar.xml.

The JSFL file obviously contains the script itself. However, if you try to open it, you won't see much. It has been encrypted with a special tool available only to Macromedia and select developers. The tools you make will be open for anyone to look at though.

The PNG is a graphics file—PNG stands for Portable Network Graphics. This is the icon image that shows up in the toolbar. In addition to your JSFL file, you'll need to supply a PNG icon for your tool. When you're just starting out creating a new tool, it's fine to use the PolyStar.png, or any other existing tool icon PNG in the directory. Flash doesn't complain about using the same one more than once. But by the time you wrap up your tool, you'll want to create a nice icon for it. The easiest way to do that is to open up the PolyStar.png in Fireworks, Photoshop, or whatever graphics editor you prefer, change it and save it with a new filename. If you're starting from scratch, just make a 16 × 15 pixel, 24-bit (8 bits per channel) graphic saved as a PNG file.

The XML file is the file for the Options dialog box and is completely optional. Your first few files won't use it, but you'll check into it later on.

You should also see a file called toolConfig.xml and one called toolDefaultConfig.xml. Feel free to look at these and get an idea of how they are made up. The toolConfig.xml file is what Flash uses to load the toolbar up when it starts. This tells it where to place each tool. And toolDefaultConfig.xml is used to reset the tools when toolConfig.xml is missing or you choose to return to the default configuration after you've heavily customized the toolbar. You don't need to edit these files manually. They are created automatically when you use the Customize Tools panel. You use the panel to set up the configuration, and when you click OK, it saves to the XML file automatically.

So now you're almost ready to start making a tool. But first, you're going to make one more command! What? Go backwards? Trust us. This command is a simple workflow enhancement and will take about 30 seconds to make; it'll save you hours in the long run!

Create a JSFL file in your Commands directory called `Reload Tools.jsfl`. It just needs one line of code:

```
f1.reloadTools();
```

Why do you need this? As we covered earlier, when Flash loads up, it goes through the Tools folder and loads the JSFL file for each custom tool it finds there and executes its `configureTool` function. After that, it never touches the JSFL file on the hard disk again.

So, if you're in the process of writing a tool, you save the JSFL file and use the Customize Tools panel to add it to the toolbar. Then you test it out. If you're like us, it doesn't do much of anything on the first try, with the possible exception of generating some interesting error messages. So you go back in and find the bugs and save the file again. But Flash has already loaded the tool and has no idea that you've changed it. It will keep on using the code it loaded into memory.

You have three options:

1. Shut down Flash and restart it.
2. Go into the Customize Tools panel, remove the tool from the menu, close it, open it again, and add it back in. This should load the new code into memory.
3. Run your newly created Reload Tools command. This just tells Flash to redo the process of loading up the toolbar. You'll probably see the toolbar flicker a bit. This is your clue that it has reloaded successfully.

## Creating a Tool

Finally! We've covered all the necessary prerequisites for you to be able to actually start coding a tool. Let's go through it step by step. Your first tool won't do much of anything, but at least you'll be able to see it in the toolbar and get a feel for the process of creating a tool and familiarize yourself with some of the different events and when they fire.

1. First off, create a simple icon of whatever design takes your fancy. Again, this should be a 16-by-15 PNG file. You can skip this step if you want, and just use the `PolyStar.png` file. Save it as `TestTool.png` in your Tools directory.
2. Next, create a new JSFL file in your Tools directory. Name it `TestTool.jsfl`. Using the function names mentioned previously, set up a basic tool file including each function. Inside each one, put a `f1.trace` statement tracing the name of the function. This should look something like the following:

```
function configureTool(){  
    f1.trace("configureTool");  
}
```

```
function activate(){
    fl.trace("activate");
}

function deactivate(){
    fl.trace("deactivate");
}

function keyDown(){
    fl.trace("keyDown");
}

function keyUp(){
    fl.trace("keyUp");
}

function mouseDbClk(){
    fl.trace("mouseDbClk");
}

function mouseDown(){
    fl.trace("mouseDown");
}

function mouseMove(){
    fl.trace("mouseMove");
}

function mouseUp(){
    fl.trace("mouseUp");
}

function notifySettingsChanged(){
    fl.trace("notifySettingsChanged");
}

function setCursor(){
    fl.trace("setCursor");
}
```

3

3. You still need to do a bit more to actually get this tool running, or even be visible on the toolbar, but if you now run your Reload Tools command, you should see a trace of `configureTool`. As you can see, Flash has found the JSFL file, found its `configureTool` function, and executed it. But in order to have the tool accessible, you need to add a few things to this function. In fact, if you look in the Customize Tools panel, you won't even see the tool there available to add. When Flash reloaded the tools, it did not find enough info in the `configureTool` function to be able to add it there.

4. So, add the following lines to the `configureTool` function:

```
function configureTool(){
    fl.trace("configureTool");
    var curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setIcon("TestTool.png");
    curr_toolObj.setMenuString("Test Tool");
    curr_toolObj.setToolName("Test Tool");
    curr_toolObj.setToolTip("Test Tool");
}
```

Let's take a look at this new code. First, you grab a reference to the "active tool." In the `fl` object, there is another object called `tools`. The `tools` object contains an array called `toolObjs`, which contains a reference to each tool. The `tools` object also contains a property called `activeTool`, which contains a reference to whatever tool is active. When a tool is chosen on the toolbar, it is the `activeTool`. Also, while a tool is loaded and configured when Flash is starting up or reloading the tools, it becomes the `activeTool` during that period.

Anyway, you store a reference to the `activeTool` in a variable called `curr_toolObj`. A `toolObj` is an object with several methods and properties. Here you see the important ones for configuring a tool:

- You set the icon. Remember, if you didn't create a custom icon for this tool, use the `PolyStar.png` filename here.
- Then you set the menu string. This is the name you'll see in the drop-down tools list.
- Then you set the tool name, which is what will show up in the Properties panel when the tool is active.
- Finally you set the tool tip, which is the name that pops up when you mouse over a particular tool icon.

All these names are generally set to the same thing, though that isn't required.

5. Save this, and once again open your Customize Tools panel. Now you'll see your Test tool with its icon, ready to add. Click a spot on the toolbar where you want to add the tool. You can stick it in with the Rectangle and PolyStar tools, or tack it onto any other tool. You could even remove an infrequently used tool from its slot and put your custom tool right on the top level if you want. Once you have a spot picked out for it, click its name in the Available Tools list, then click the Add button. Click OK and check out your new, customized toolbar, complete with your new tool!

You should already receive a message for `configureTool` in the Output panel. That's a good sign. Now, select your tool and you should get an activate message. As you move the mouse around, you should get a flurry of `mouseMove` and `setCursor` messages. Try out the mouse up and down actions too. Finally, select another tool and you should see a deactivate message, and that should be the end of it until you select the tool again.

When you're done with this particular learning experience, you'll want to go into the Customize Tools panel and remove it from the toolbar. In addition, you'll probably want to

remove the JSFL file from your Tools directory, lest you get an annoying trace message every time you reload your tools (you'll be doing that quite a lot as you progress through this chapter).

## Grid Tool

Interesting as that was, you'll now create a tool that actually does something! Once again you're going to go with the grid theme, creating a tool that will draw a grid. The first major idea you'll be learning about is the drawing layer. This is a special object in the Flash DOM that allows you to preview what you'll be drawing without actually drawing any content to the document itself.

There's still a lot of material to cover just to make a simple tool like this. In fact, most of the rest of this chapter will be spent just going through the different requirements and options using this same file.

1. First you'll need an icon. You can make your own, naming it `grid.png`, or just download the one that has already been created and is available on this book's website, along with all the other example files.
2. Next, you'll need to start a JSFL file named `grid.jsfl`. To start with, you'll need to include the following event-handler functions:
  - `configureTool`
  - `activate`
  - `mouseDown`
  - `mouseUp`
  - `mouseMove`

You can set these up as empty functions for the moment, and you'll fill them up with code as you go. You can start setting up the `configureTool` function right off the bat. Most of this we've already covered:

```
function configureTool(){
    var curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setIcon("grid.png");
    curr_toolObj.setMenuString("Grid Tool");
    curr_toolObj.setToolName("Grid Tool");
    curr_toolObj.setToolTip("Grid Tool");
}
```

After saving this file, your tool will be set up enough to add to the toolbar. So go ahead and open up the Customize Tools panel and add it wherever you want it. If your Grid tool isn't in the list of available tools, make sure it is saved in the Tools directory, along with the icon file, and check the syntax of the code in the file. If there is any major error that prevents `configureTool` from running, Flash will not be able to get enough information about the tool to list it. Once you successfully add the tool and have it show up in the toolbar, you can move on.

3. Next, you'll set up the `activateTool` function. You really only need one line here to grab a reference to the current tool. This reference will then be available for as long as the tool is active:

```
function activate(){
    curr_toolObj = fl.tools.activeTool;
}
```

Now, in any other function, you can refer to the `activeTool` with `curr_toolObj` without having to set a reference. Remember, you can add other variables or statements here as needed for your particular tool. This is all you need for now though.

4. Next, you'll add some code to `mouseDown` and `mouseUp`. If you think about the other drawing tools in Flash, nothing too much happens until you click the mouse to start drawing. At that point, you enter a kind of interactive drawing mode, which exists until you release the mouse button. At that point, you exit the interactive mode, your drawing is placed on the stage, and Flash waits for you to do something else.

All of the actual drawing will take place in the `mouseMove` function. All you need to do here is enter or exit that drawing mode. You do that with the commands `fl.drawingLayer.beginDraw()` and `fl.drawingLayer.endDraw()`. This tells the drawing layer (the `fl.drawingLayer` object) to get ready to receive some drawing commands. Always remember to call `beginDraw` before doing any drawing, and `endDraw` when you're done.

Also, a drawing tool usually draws something at the physical location where the mouse was first clicked. So it will be good to make note of this point for future reference. Here are the two functions:

```
function mouseDown(){
    startPoint = fl.tools.penDownLoc;
    fl.drawingLayer.beginDraw();
}

function mouseUp(){
    fl.drawingLayer.endDraw();
}
```

Here you store the starting point in the variable `startPoint`. Notice how you get that point. The `tools` object contains a property called `penDownLoc`. This is a point object with `x` and `y` properties. It holds the location of the last place where the mouse was clicked. You don't really need to grab this point right at this time, since it will still be available in `penDownLoc` later, but it makes the subsequent drawing code a little clearer.

5. All right, you're ready to start some interactive drawing! This will all happen in the `mouseMove` function. As mentioned, this temporary drawing occurs in an object called the drawing layer. It has methods drawing lines and other shapes, but remember, the shapes you draw on the drawing layer are temporary. They don't get saved to the drawing. They are just a kind of preview. In effect, you need two whole chunks of code to draw the same shape—one in `mouseMove`, using the drawing layer methods, and another that will occur when the user releases the mouse. This code will use the document drawing methods and draw to the stage. Since the

drawing commands for the two modes aren't exactly the same, there is no way of writing just one function that handles both of them. You have to do the double work. (Actually, there is one shortcut to this process, which you'll learn about when we get into the path object.)

Before you start drawing, you have to issue one more setup command—`fl.drawingLayer.beginFrame()`. This has nothing to do with the frame object or frames in Flash at all. But, since you're doing a sort of animation—drawing a slightly different shape over and over again—it does have to do with frames in a more general sense in any animation. The mouse moves, you begin a frame, you draw something to it, and then you end the frame. Then the mouse moves a little more and you need to update what was drawn, so you begin a new frame, clearing the screen and redrawing a shape.

So, much like `beginDraw` and `endDraw` bracket the entire drawing phase, `beginFrame` and `endFrame` should bracket each individual instance of drawing your shape.

Here's the beginnings of it:

```
function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    fl.drawingLayer.moveTo(startPoint.x, startPoint.y);
    fl.drawingLayer.lineTo(fl.tools.penLoc.x, startPoint.y);
    fl.drawingLayer.lineTo(fl.tools.penLoc.x, fl.tools.penLoc.y);
    fl.drawingLayer.lineTo(startPoint.x, fl.tools.penLoc.y);
    fl.drawingLayer.lineTo(startPoint.x, startPoint.y);
    fl.drawingLayer.endFrame();
  }
}
```

Note that first, you check whether the mouse is down. There's no need to do anything if the mouse isn't pressed. You can find this out by checking the tools property `fl.tools.mouseIsDown`. If it is down, you begin the frame and start drawing.

But hey, wait just a second! Not only are you back to `moveTo-` and `lineTo-` style commands, but you're supplying them with individual `x` and `y` properties rather than a single point object! Just when you started getting used to the JSFL style, `addNewLine(startPoint, endPoint)`, now it looks like you're back in ActionScript. Not very consistent is it? We're not sure exactly why they switched just for this one object. All we can say is be flexible and roll with it, because that's how it is.

Anyway, that's pretty basic drawing. It draws a rectangle from the point you first clicked (`startPoint`), to the current mouse position, which can be found with `fl.tools.penLoc`.

Test this out and you should be able to draw a rectangle on the screen. Don't forget to reload your tools with the Reload Tools command. If done exactly per the code here, it should be a nice, smooth drawing operation. If you're getting a lot of flickering or artifacts, you've probably messed up the placement of your `begin` or `end drawing` or `frame` commands. If you find your tool suddenly disappears off the toolbar, you have some major error in the code that isn't allowing `configureTool` to run. If that's the case, you won't even get an error message, it will just disappear.

Other runtime errors may show up in the Output panel as you run the command, so look for them as you develop and test your own extensions.

6. This is the outline for your grid. Next, you need to fill in the individual grid lines. First you'll need to determine how many lines to draw vertically and horizontally. Later, you can let the user change this with the Options button, but for now you'll just hard code it to five rows and columns. If this were going to stay hard coded, you could do it in the `configureTool` function and be done with it. But since it's going to wind up being dynamic, you'll put it where it's eventually going to go, in the `activateTool` function.

```
function activate(){
    curr_toolObj = fl.tools.activeTool;
    rows = 5;
    cols = 5;
}
```

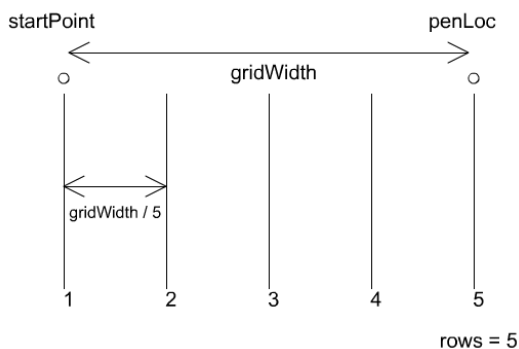
Simple enough!

7. Now you jump back to your `mouseMove` function and change the drawing code. Remember, though, that it all needs to fall between the pair of `beginFrame` and `endFrame` methods:

```
function mouseMove(){
    if(fl.tools.mouseIsDown){
        fl.drawingLayer.beginFrame();
        var gridHeight = fl.tools.penLoc.y - startPoint.y;
        for(var r = 0; r<=rows; r++){
            var offset = gridHeight/rows*r;
            fl.drawingLayer.moveTo(startPoint.x, startPoint.y + offset);
            fl.drawingLayer.lineTo(fl.tools.penLoc.x, startPoint.y + offset);
        }
        var gridWidth = fl.tools.penLoc.x - startPoint.x;
        for(var c = 0; c<=cols; c++){
            offset = gridWidth/cols*c;
            fl.drawingLayer.moveTo(startPoint.x + offset, startPoint.y);
            fl.drawingLayer.lineTo(startPoint.x + offset, fl.tools.penLoc.y);
        }
        fl.drawingLayer.endFrame();
    }
}
```

What you're doing this time is first finding the height of the grid by subtracting the beginning point from the end point. Then you loop through and draw a series of horizontal lines from the start point to the current mouse position. You determine the y position of the lines by dividing the height by the number of rows, then multiplying by the current row being drawn. You then loop through and draw the vertical lines. Figure 3-4 may help to explain this.

Save the file, again as `grid.jsfl`, in the `Tools` directory, reload your tool, and test it out. Try playing around with the settings for rows and columns and make sure everything is working so far.



**Figure 3-4.**  
Drawing the vertical lines  
of the grid

3

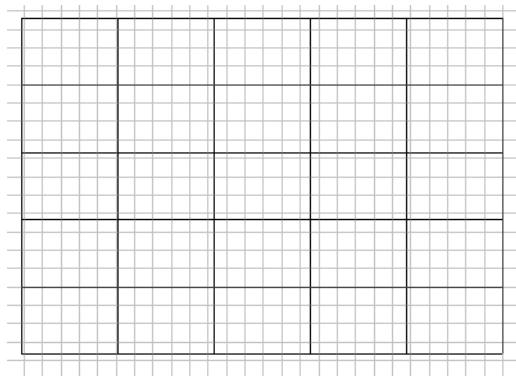
8. Next, you need to write some code to permanently draw the grid into the document. This occurs after users draw their grid to the size and shape they want and then release the mouse button. Thus, this code should go in the `mouseup` function. Essentially this code does exactly the same thing as the drawing layer code, it just uses the document object `addNewLine` method instead. Here is the new `mouseup` function:

```
function mouseUp(){
    fl.drawingLayer.endDraw();
    var endPoint = fl.tools.penLoc;
    var curr_doc = fl.getDocumentDOM();
    var gridHeight = endPoint.y - startPoint.y;
    for(var r = 0; r<=rows; r++){
        var offset = gridHeight/rows*r;
        var lineStart = {x:startPoint.x, y:startPoint.y + offset};
        var lineEnd = {x:endPoint.x, y:startPoint.y + offset};
        curr_doc.addNewLine(lineStart, lineEnd);
    }
    var gridWidth = endPoint.x - startPoint.x;
    for(var c = 0; c<=cols; c++){
        offset = gridWidth/cols*c;
        var lineStart = {x:startPoint.x + offset, y:startPoint.y};
        var lineEnd = {x:startPoint.x + offset, y:endPoint.y};
        curr_doc.addNewLine(lineStart, lineEnd);
    }
}
```

After ending drawing mode, you grab the current mouse position. That way the value you're using doesn't change slightly during the execution of this function, which could throw off what you're drawing. Then you get a reference to the current document. This must be old hat to you by now!

The rest of the function should look pretty similar. Much of it is exactly the same. You get the width and height and offset, and draw the lines. The only difference is that you create `lineStart` and `lineEnd` point objects to store the points in, and then pass the points to the `addNewLine` function. This just results in cleaner code than trying to pass lengthy statements to the function directly.

OK, what you've now got here is a pretty handy tool that will draw any grid you want . . . as long as it's a 5-by-5 grid like the one in Figure 3-5. Now let's dig into that Options button and make it a little more flexible.



**Figure 3-5.**  
A 5-by-5 grid drawn with  
the Grid tool

## Adding Options

You've already had a taste of XML to UI in making commands (Chapter 2), so this next section should come pretty easy to you. In fact, the XML used to create the Tool Settings (Options) panel is even simpler than XML to UI. Let's take a look at the PolyStar.xml file as an example:

```
<properties>
  <property name="Style" variable="style" list="polygon,star"
    ➤defaultValue="0" type="Strings" />
  <property name="Number of Sides" variable="nsides" min="3" max="32"
    ➤defaultValue="5" type="Number" />
  <property name="Star point size" variable="pointParam" min="0"
    ➤max="1" defaultValue=".5" type="Double" />
</properties>
```

The XML's root tag is <properties>, and contains a number of <property> tags. Each <property> tag has several attributes. name is basically the label that will be shown in the Options dialog box, and variable is the actual variable name that will be passed to the tool. You'll see exactly how it is passed in a moment. From there you jump down to type. This is the data type of the variable and can be set to any one of the following:

- Number: Creates a text box accepting integers
- Double: Creates a text box accepting floating-point numbers
- Boolean: Creates a check box for setting any true/false values
- String: Creates a text box that will accept a string
- Strings: Creates a drop-down list box for choosing one of several strings
- Color: Creates a color chip that will pop open a color chooser

Depending on which data type you specify, other attributes can be set. The Number and Double types allow you to set minimum and maximum allowable values with the `min` and `max` attributes. You can also specify a `defaultValue` attribute, which is the value that will initially appear in the dialog box.

The Boolean type also has a `defaultValue` attribute that can be set to `true` or `false`, and the String type has the `defaultValue` attribute as well. The Strings type has an attribute called `list`; this is a comma-separated list of strings. The entire list is enclosed in a single set of quotes like so: `list="dogs,cats,birds"`. Each comma-separated value will be one item in the list box. The `defaultValue` attribute accepts an integer corresponding to which list item you want to show up first in the dialog box. The list is zero indexed, so in the preceding example, `defaultValue="0"` will result in `dogs` being the default string. Finally, the Color type can take a default value.

These variable type definitions are summarized in the following table:

**Table 3-1.** Variable Types for the Tool Settings Panel

Variable Type	Interface Element Created	Attributes
Number	Text field	1. min 2. max 3. defaultValue (integer)
Double	Text field	1. min 2. max 3. defaultValue (floating-point number)
Boolean	Check box	1. defaultValue (true/false)
String	Text field	1. defaultValue (string)
Strings	Drop-down list	1. list (comma separated strings) 2. defaultValue (index number)
Color	Color chooser	1. defaultValue (color value)

If you don't want to hand code all this XML, or can't remember all the options, you can also use the Dialog Designer available on this book's accompanying website ([www.flashextensibility.com](http://www.flashextensibility.com)).

Well, that's all you need to know to get started, so let's go forth and create XML.

1. For your Grid tool, you just need two integer values: one for rows and one for columns. You'll set the minimum at 1 and maximum at 20 for each one.

```
<properties>
  <property name="Rows" variable="rows" min="1" max="20"
  ↪defaultValue="5" type="Number"/>
```

```

    <property name="Columns" variable="cols" min="1" max="20"
    ➔defaultValue="5" type="Number"/>
  </properties>

```

Just save that in your Tools directory as grid.xml.

2. Then you need to tell the script that it should use this XML file as its options file. This goes in the configureTool function:

```

function configureTool(){
    var curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setIcon("grid.png");
    curr_toolObj.setMenuString("Grid Tool");
    curr_toolObj.setToolName("Grid Tool");
    curr_toolObj.setToolTip("Grid Tool");
    curr_toolObj.setOptionsFile("grid.xml");
}

```

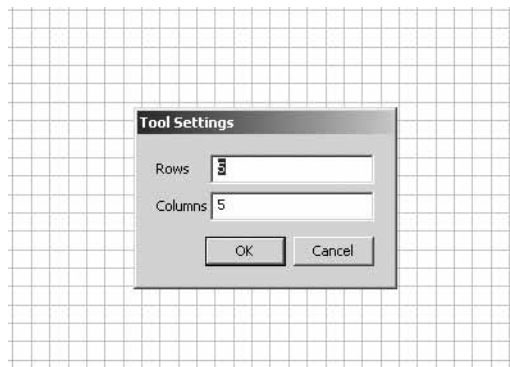
3. Now you need to have your script grab these values. This will be done as soon as the tool is selected, so you'll put it in the activate function. This is why we originally chose to have you hard code the values for rows and columns in this function. Now you can simply change the code in that location. The properties you defined in the XML will become properties of the activeTool. You already got a reference to that object, so you can easily read these two properties and assign them to your variables. Here's the updated activate function:

```

function activate(){
    curr_toolObj = fl.tools.activeTool;
    rows = curr_toolObj.rows;
    cols = curr_toolObj.cols;
}

```

You can save and reload and test this. If all works well, you shouldn't really notice a difference. When the tool is activated, it grabs the default values for rows and columns and uses them. You should notice though that the Options button is now available in the Properties panel. If you click it, you should get a dialog box allowing you to change the values (see Figure 3-6).



**Figure 3-6.**  
Tool Settings dialog box  
for the Grid tool

4. By now you've probably noticed, however, that changing these values has no effect. That's because you only check the values in the activate function. Actually, if you set some values, then change to another tool and change back to the grid, activate will run again, grabbing the new values. But you want to grab those values as soon as they change. As briefly mentioned earlier, when the user clicks OK in the Options dialog box, the notifySettingsChanged function will be called, if it has been defined. So you need to define it!

You've already got a reference to the active tool, so all you need to do here is grab the updated properties off it.

```
function notifySettingsChanged(){
    rows = curr_toolObj.rows;
    cols = curr_toolObj.cols;
}
```

Do the old save, reload, test routine and play around with it. You should be able to change the settings and have them instantly respond.

You've come a long way, and you have a pretty useful tool here. You may think you're just about finished, but if you were to make a tool like this and set it free in the world, you would receive a flood of complaints! Believe it or not, there is still a fair amount of work to do on this thing. So, let's forge on to make a really professional job of it.

## Custom Cursors

If you try using one of the other built-in drawing tools, notice the cursor as you select it and then move around the screen. As soon as you choose the tool and move it onstage, it changes to a crosshair (plus sign). This is a kind of visual cue that you're in drawing mode. Then, if you roll over the toolbar, timeline, or some panel, the cursor changes back to an arrow. Now try the same with the Grid tool. You'll notice that it never changes from the original arrow. It's a minor point, but it goes a long way to making the tool look and feel professional.

Fortunately, all that switching functionality is built into Flash. When Flash decides that the cursor needs to be a particular shape, such as an arrow, it will grab the control and change it. When it deems that it's safe to hand control of the cursor back to your tool, it will call the setCursor event handler, if it is defined. Though you can do anything you want in that function, generally all you want to do is change the cursor to a particular shape, in this case the crosshair. You can do this with the function `fl.tools.setCursor`. This takes one argument, an integer from 0 to 7. Here's what those numbers mean:

- 0: Plus cursor (crosshair)
- 1: Black arrow
- 2: White arrow
- 3: Four-way arrow
- 4: Two-way horizontal arrow
- 5: Two-way vertical arrow
- 6: X cursor
- 7: Hand cursor

So, you just need to make a function like this:

```
function setCursor(){
    fl.tools.setCursor(0);
}
```

It doesn't get much easier than that. Test it out. Isn't that much nicer?

Note that you could also do more advanced cursor handling, using conditionals. Maybe if the cursor is within a certain area you would want to have it look one way, or if it is in another area, have it look another way, as the following pseudo-code shows:

```
if(condition1){
    fl.tools.setCursor(1);
} else if(condition2){
    fl.tools.setCursor(2);
}
```

## Snap to Grid

You may or may not use a grid and the snap-to-grid feature, but many Flash users do (check the menus View ► Grid ► Show Grid and View ► Snapping ► Snap to Grid). These options allow you to set a grid on which objects and drawing tools will jump to the nearest point. It may not be useful for free-form drawing or cartooning, but for creating interface elements or other items that must be a specific size or line up at specific locations, it is quite useful. There's a good chance that someone drawing a grid will want it to snap to a particular size or location, so let's code in that functionality.

The good news is that there is a function in JSFL that allows you to do so rather easily: `fl.tools.snapPoint`. This function takes a point object as an argument and returns a point object. What this does internally is first check if Snap to Grid is set in the authoring environment. If so, it finds the point on the grid that is closest to the point supplied as an argument, and returns this point. If Snap to Grid isn't set, it simply returns the original point. Thus, as a programmer, you never have to worry about whether or not the user wants to snap. You simply convert your points using this function and the user's settings will determine what happens.

You need to apply this function in three places. The first is when the user first clicks the mouse to begin drawing—in the `mouseDown` function. You already store this point in a variable called `startPoint` with the line

```
startPoint = fl.tools.penDownLoc;
```

You simply change this to the following:

```
startPoint = fl.tools.snapPoint(fl.tools.penDownLoc);
```

Now, the starting point will always snap to the grid (if the user so wishes).

The second place you need to change it is during the drawing process itself—actually in the drawing layer preview code in `mouseMove`. It isn't crucial to snap to the grid at this

point, but you may as well have your preview be an exact copy of the final drawing, so you'll snap to the grid.

Let's take another look at the function as it stands:

```
function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    var gridHeight = fl.tools.penLoc.y - startPoint.y;
    for(var r = 0; r<=rows; r++){
      var offset = gridHeight/rows*r;
      fl.drawingLayer.moveTo(startPoint.x, startPoint.y + offset);
      fl.drawingLayer.lineTo(fl.tools.penLoc.x, startPoint.y + offset);
    }
    var gridWidth = fl.tools.penLoc.x - startPoint.x;
    for(var c = 0; c<=cols; c++){
      offset = gridWidth/cols*c;
      fl.drawingLayer.moveTo(startPoint.x + offset, startPoint.y);
      fl.drawingLayer.lineTo(startPoint.x + offset, fl.tools.penLoc.y);
    }
    fl.drawingLayer.endFrame();
  }
}
```

The parts in bold are where the current mouse location is referred to. These will be the “unsnappped points.” Rather than converting them four times, you can create a new variable called `penPoint` that will be the snapped version of `penLoc`. Then you use `penPoint` to do your drawing layer drawing. Here's the corrected version:

```
function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    var penPoint = fl.tools.snapPoint(fl.tools.penLoc);
    var gridHeight = penPoint.y - startPoint.y;
    for(var r = 0; r<=rows; r++){
      var offset = gridHeight/rows*r;
      fl.drawingLayer.moveTo(startPoint.x, startPoint.y + offset);
      fl.drawingLayer.lineTo(penPoint.x, startPoint.y + offset);
    }
    var gridWidth = penPoint.x - startPoint.x;
    for(var c = 0; c<=cols; c++){
      offset = gridWidth/cols*c;
      fl.drawingLayer.moveTo(startPoint.x + offset, startPoint.y);
      fl.drawingLayer.lineTo(startPoint.x + offset, penPoint.y);
    }
    fl.drawingLayer.endFrame();
  }
}
```

Finally, when the user releases the mouse (`mouseUp`) and you start to draw the final lines in the document, you want to snap there as well. Again, you've already got a variable, `endPoint`. You just need to snap it to the grid. So, in the `mouseUp` function, the line

```
var endPoint = fl.tools.penLoc;
```

becomes

```
var endPoint = fl.tools.snapPoint(fl.tools.penLoc);
```

And that pretty much does it for snapping to the grid. Try that out and see that it does indeed cause items to snap to the grid. You'll need to enable Snap to Grid in the View ► Snapping menu, and it helps if you select Show Grid in the View ► Grid menu. All you have to remember is to pass any points to `fl.tools.snapPoint`, and use the returned point instead. You can even assign a point back to itself such as `myPoint = fl.tools.snapPoint(myPoint)`. Most of the time, though, you'll be snapping points obtained through `fl.tools.penLoc` or `fl.tools.penDownLoc`, and assigning them to brand new variables.

Bit by bit, your tool is becoming very professional. Let's see what else you can do to it.

## Constraining a Shape

Another feature of almost any drawing tool is to be able to hold down the *SHIFT* key and constrain the drawing to a particular shape. For example, if you hold *SHIFT* while drawing a line, you'll get a perfectly horizontal line, vertical line, or 45-degree-angle line. Constraining the Rectangle tool allows you to draw a perfect square, and with the Oval tool you get a perfect circle. Why should you be left out of all this neat constraining action?

Similar to `snapPoint`, `fl.tools` has a method called `constrainPoint`. This takes two point objects as arguments, and returns a single point. The best way to describe it is that it takes the position of the second point and snaps it to the nearest point, which is either vertically or horizontally aligned with the first point, or 45 degrees from it. It's probably best to see it in action, and it can best be seen with the Line tool. If you draw a line with this tool, with the *SHIFT* key held down, you can think of the first point as where you first clicked. The second point is where the mouse currently is located. The return point is where the end of the line falls. Notice that if you get close enough to horizontal or vertical, the line will snap to that. If you're closer to where the end point will be at 45 degrees, it will snap to that.

Also similarly to `snapPoint`, which returns a snapped point only if Snap to Grid is on, `constrainPoint` only returns a constrained point if the *SHIFT* key is held down. Otherwise, it just returns the value of the second point.

So, how do you use it? The only points you need to use `constrainPoint` occur when you're drawing. These will be `mouseMove` and `mouseUp`.

Looking at the first few lines of `mouseMove`, recall that you've just snapped the current pen location and stored it in the variable, `penPoint`. That's the guy you want to constrain. And you want `penPoint` to line up with `startPoint`, so that's the first argument. Just add this line after defining `penPoint`:

```
function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    var penPoint = fl.tools.snapPoint(fl.tools.penLoc);
    penPoint = fl.tools.constrainPoint(startPoint, penPoint);
    //...continued
```

Now, if *SHIFT* is being held down while drawing, `penPoint` will either be lined up with `startPoint` vertically, horizontally, or at 45 degrees.

Then you do the same thing with `endPoint` in `mouseUp`:

```
function mouseUp(){
  fl.drawingLayer.endDraw();
  var endPoint = fl.tools.snapPoint(fl.tools.penLoc);
  endPoint = fl.tools.constrainPoint(startPoint, endPoint);
  //...continued
```

This makes sure the final drawing is the same as the preview drawing. You can go ahead and try that out. Now, you can hold down *SHIFT* and get a square grid if that's what you want. But it's still not quite right. If you get too close to vertical or horizontal, it snaps to that and just draws a line. No need to use a Grid tool to draw a line. Anyway, the Rectangle and Oval tools don't constrain this way. They always form a square or circle. So let's be consistent.

Rather than trying to fix the existing `constrainPoint` function, just go ahead and make your own function called `constrain45`. All it does is compare the two points and adjust the second point accordingly. We won't go into an in-depth explanation of it, but it should be pretty easy to follow if you want to figure it out. Or, you could just plug it into your function and trust us that it works.

```
function constrain45(p1, p2){
  if (fl.tools.shiftIsDown)
  {
    var dx = Math.abs(p2.x - p1.x);
    var dy = Math.abs(p2.y - p1.y);
    var offset = Math.max(dx, dy);
    if(p2.y < p1.y)
    {
      p2.y = p1.y - offset;
    }
    else
    {
      p2.y = p1.y + offset;
    }
    if(p2.x > p1.x)
    {
      p2.x = p1.x + offset;
    }
    else
```

```

    {
      p2.x = p1.x - offset;
    }
  }
}

```

One thing to note about this function is that it directly changes the coordinates of the second point, rather than returning a new point like `constrainPoint`. So when you use it, there's no need to catch the return value.

So, your `mouseUp` and `mouseMove` functions become this:

```

function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    var penPoint = fl.tools.snapPoint(fl.tools.penLoc);
    constrain45(startPoint, penPoint);
    //...continued

function mouseUp(){
  fl.drawingLayer.endDraw();
  var endPoint = fl.tools.snapPoint(fl.tools.penLoc);
  constrain45(startPoint, endPoint);
  //...continued

```

Again, we're just giving the first few lines. Now, your grid is constrained to a square only if *SHIFT* is held down.

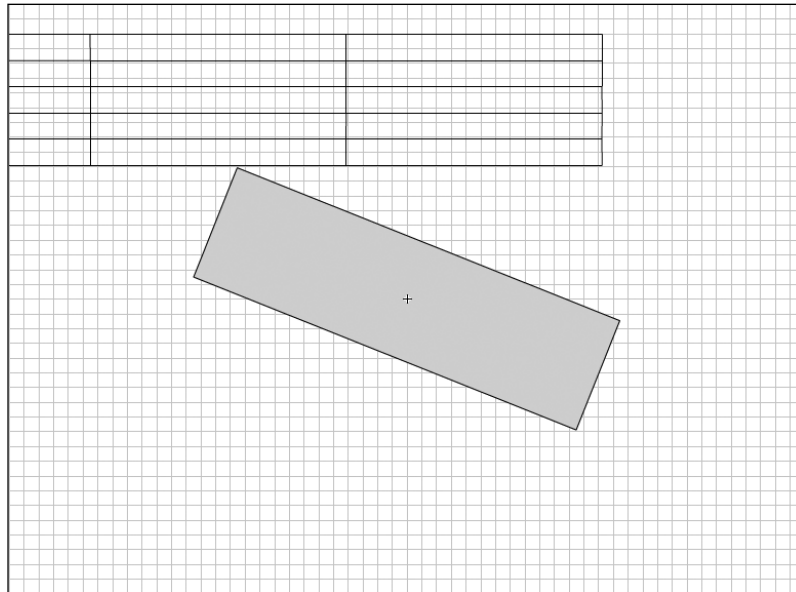
## Enter the Matrix

OK, you're almost finished! There is just one more thing you need to do before you set this tool free. It actually contains a major bug that we haven't addressed so far. You may have even run into this bug if you've played around with this tool enough. If not, follow these steps and see just how bad it is:

1. Create a large shape onstage—a big rectangle will do.
2. Select it and convert it into a movie clip.
3. Now double-click to edit it in place.
4. Inside this shape, choose your Grid tool and draw a grid.

Hey, what's happening here? The grid isn't matching up with the mouse. It gets worse. Go back to the original clip, scale it a bit on different axes, and rotate it some. Now go back inside and try another grid.

You see the bug now (shown in Figure 3-7)? Pretty bad isn't it? Can you even see the grid you're drawing? It's there. You might have to scale down a little or scroll around as it could be offstage. What a mess! How did your grid go so awfully wrong after so much work?



**Figure 3-7.** A grid gone horribly wrong inside a transformed movie clip

Actually, your grid didn't do anything wrong. It's just in kind of a time-space warp. It seems that the pen locations that you get from `fl.tools.penLoc` and `fl.tools.penDownLoc` are always in relation to coordinates of the movie clip you're drawing inside. But the `drawingLayer` drawing functions, as well as the document functions `addNewLine`, and so on, all work with the global, root-level stage coordinates. Unless the clip's registration point happens to line up with 0, 0 on the stage, you're already starting out poorly. Once you rotate or scale it, you're compounding the problem, as the internal coordinates aren't even going to be close to the stage coordinates.

So what do you do to fix it? The answer is in the matrix. No, we're not going all sci-fi on you. The answer is in a document property called the `viewMatrix`. The `viewMatrix` property holds all the information on how the current view has been translated (moved off center), rotated, and scaled.

By itself, the `viewMatrix` does nothing. It simply holds the relevant information. It's up to you to use that to transform `penLoc` or `penDownLoc` into the correct point to draw to.

Simple, right? Well not really, unless you happen to know something about matrix math. There really isn't space in this chapter to get into a long discussion about such a subject, not to mention that if we were going to teach it to you, we'd have to learn it ourselves. Instead, we're just going to hand you a function that will take care of this for you, and show you how to use it. You just have to include this function in any tool file and call it when needed. Here it is:

```
function transformPoint(aPoint, matrix) {
    var x = aPoint.x * matrix.a + aPoint.y * matrix.c + matrix.tx;
    var y = aPoint.x * matrix.b + aPoint.y * matrix.d + matrix.ty;
```

```

    aPoint.x = x;
    aPoint.y = y;
}

```

You need to apply this function to every point you get using `penLoc` or `penDownLoc`. Once again, this means three locations—`mouseDown`, `mouseMove`, and `mouseUp`. You have to do some minor rearranging, but nothing too painful. You'll just grab the point from `penLoc` or `penDownLoc` and store it in a variable. Then you'll transform the variable. This little function will handle all the rotation, scaling, and translation issues. Then you'll continue on as usual by snapping and constraining it. Wow, this little point is getting a real workout!

Here are the revised functions, just showing the first few lines for the longer ones:

```

function mouseDown(){
    startPoint = fl.tools.penDownLoc;
    transformPoint(startPoint, fl.getDocumentDOM().viewMatrix);
    startPoint = fl.tools.snapPoint(startPoint);
    fl.drawingLayer.beginDraw();
}

function mouseUp(){
    fl.drawingLayer.endDraw();
    var endPoint = fl.tools.penLoc;
    transformPoint(endPoint, fl.getDocumentDOM().viewMatrix);
    endPoint = fl.tools.snapPoint(endPoint);
    constrain45(startPoint, endPoint);
    //...continued

function mouseMove(){
    if(fl.tools.mouseIsDown){
        fl.drawingLayer.beginFrame();
        var penPoint = fl.tools.penLoc;
        transformPoint(penPoint, fl.getDocumentDOM().viewMatrix);
        penPoint = fl.tools.snapPoint(penPoint);
        constrain45(startPoint, penPoint);
        //...continued

```

Go ahead and test it out under the conditions described earlier. The grid should stay straight and in place no matter how you've transformed the outer movie clip.

For completeness, here's the final code for your Grid tool (see also `grid.jsfl`):

```

function configureTool(){
    var curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setIcon("grid.png");
    curr_toolObj.setMenuString("Grid Tool");
    curr_toolObj.setToolName("Grid Tool");
    curr_toolObj.setToolTip("Grid Tool");
    curr_toolObj.setOptionsFile("grid.xml");
}

```

```

function activate(){
    curr_toolObj = fl.tools.activeTool;
    rows = curr_toolObj.rows;
    cols = curr_toolObj.cols;
}
function notifySettingsChanged(){
    rows = curr_toolObj.rows;
    cols = curr_toolObj.cols;
}
function mouseDown(){
    startPoint = fl.tools.penDownLoc;
    transformPoint(startPoint, fl.getDocumentDOM().viewMatrix);
    startPoint = fl.tools.snapPoint(startPoint);
    fl.drawingLayer.beginDraw();
}
function setCursor(){
    fl.tools.setCursor(0);
}
function mouseUp(){
    fl.drawingLayer.endDraw();
    var endPoint = fl.tools.penLoc;
    transformPoint(endPoint, fl.getDocumentDOM().viewMatrix);
    endPoint = fl.tools.snapPoint(endPoint);
    constrain45(startPoint, endPoint);
    var curr_doc = fl.getDocumentDOM();
    var gridHeight = endPoint.y - startPoint.y;
    for(var r = 0; r<=rows; r++){
        var offset = gridHeight/rows*r;
        var lineStart = {x:startPoint.x, y:startPoint.y + offset};
        var lineEnd = {x:endPoint.x, y:startPoint.y + offset};
        curr_doc.addNewLine(lineStart, lineEnd);
    }
    var gridWidth = endPoint.x - startPoint.x;
    for(var c = 0; c<=cols; c++){
        offset = gridWidth/cols*c;
        var lineStart = {x:startPoint.x + offset, y:startPoint.y};
        var lineEnd = {x:startPoint.x + offset, y:endPoint.y};
        curr_doc.addNewLine(lineStart, lineEnd);
    }
}
function mouseMove(){
    if(fl.tools.mouseIsDown){
        fl.drawingLayer.beginFrame();
        var penPoint = fl.tools.penLoc;
        transformPoint(penPoint, fl.getDocumentDOM().viewMatrix);
        penPoint = fl.tools.snapPoint(penPoint);
        constrain45(startPoint, penPoint);
        var gridHeight = penPoint.y - startPoint.y;
        for(var r = 0; r<=rows; r++){

```

```

        var offset = gridHeight/rows*r;
        fl.drawingLayer.moveTo(startPoint.x, startPoint.y + offset);
        fl.drawingLayer.lineTo(penPoint.x, startPoint.y + offset);
    }
    var gridWidth = penPoint.x - startPoint.x;
    for(var c = 0; c<=cols; c++){
        offset = gridWidth/cols*c;
        fl.drawingLayer.moveTo(startPoint.x + offset, startPoint.y);
        fl.drawingLayer.lineTo(startPoint.x + offset, penPoint.y);
    }
    fl.drawingLayer.endFrame();
}
}
function transformPoint(aPoint, matrix){
    var x = aPoint.x * matrix.a + aPoint.y * matrix.c + matrix.tx;
    var y = aPoint.x * matrix.b + aPoint.y * matrix.d + matrix.ty;
    aPoint.x = x;
    aPoint.y = y;
}
function constrain45(p1, p2)
{
    if (fl.tools.shiftIsDown)
    {
        var dx = Math.abs(p2.x - p1.x);
        var dy = Math.abs(p2.y - p1.y);
        var offset = Math.max(dx, dy);
        if(p2.y < p1.y)
        {
            p2.y = p1.y - offset;
        }
        else
        {
            p2.y = p1.y + offset;
        }
        if(p2.x > p1.x)
        {
            p2.x = p1.x + offset;
        }
        else
        {
            p2.x = p1.x - offset;
        }
    }
}
}

```

And, with that, we think we can safely say that you have a professional and robust custom-built Grid tool! We warned you that it was more complex than a command, and hopefully now you see why. With experience, you'll soon become very familiar with the construction process.

That's about all you need to do with that command, but there are more objects and functions available for you to use in making commands. We'll walk you through making a couple more tools that once again will be useful, and we'll also explore some of the other possibilities.

## Arrow Tool

This tool will allow the user to instantly draw an arrow on the stage. You'll allow the user to adjust the length and the width of the arrow shaft via the tool's Options panel.

The Grid tool covered just about all the major bases of the process for creating a tool, and can almost be used as a tool template. In fact, you can start this tool by simply saving the GridTool.jsfl file as Arrow.jsfl. You'll just go through and change what you need to, which is surprisingly little. (Of course, you can also refer to the file arrow.jsfl to see the final version of your tool, which is available to download from [www.flashextensibility.com](http://www.flashextensibility.com).)

First of all, let's hit the configureTool function. You just need to change the names in the various settings from Grid to Arrow:

```
function configureTool(){
    var curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setIcon("arrow.png");
    curr_toolObj.setMenuString("Arrow Tool");
    curr_toolObj.setToolName("Arrow Tool");
    curr_toolObj.setToolTip("Arrow Tool");
    curr_toolObj.setOptionsFile("arrow.xml");
}
```

You'll also need to create another icon file, or download arrow.png.

Next, you'll create the XML file:

```
<properties>
  <property name="Shaft Width (percent):" variable="shaftWidth" min="0"
  ➤max="1" defaultValue=".6" type="Double" />
  <property name="Shaft Length (percent):" variable="shaftLength"
  ➤min="0" max="1" defaultValue=".6" type="Double" />
</properties>
```

As you can see, shaftWidth and shaftLength are floating-point variables that can be from 0 to 1. A shaft length of 0.5 would mean that the shaft extends to half the length of the arrow, and the rest of it is the point. Similarly, a shaft width of 0.5 would produce a shaft that is half as wide as the width of the point.

All you have to do to get these properties is grab them in the activate and notifySettingsChanged functions:

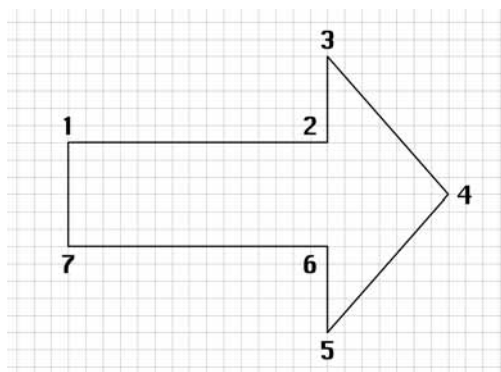
```
function activate(){
    curr_toolObj = fl.tools.activeTool;
```

```

shaftWidth = curr_toolObj.shaftWidth;
shaftLength = curr_toolObj.shaftLength;
}
function notifySettingsChanged(){
shaftWidth = curr_toolObj.shaftWidth;
shaftLength = curr_toolObj.shaftLength;
}

```

The functions `mouseDown`, `setCursor`, `transformPoint`, and `constrain45` don't need to change a bit. All you really need to change at this point is the code that draws the shape itself. If you look at Figure 3-8, you'll see that we've numbered the points 1 to 7 for reference:



**Figure 3-8.** The seven points of an arrow shape

In the `mouseMove` function, you're simply going to move to point 1, then execute a bunch of `lineTo` commands to points 2, 3, 4, 5, 6, 7, and then back to 1:

```

function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    var penPoint = fl.tools.penLoc;
    transformPoint(penPoint, fl.getDocumentDOM().viewMatrix);
    penPoint = fl.tools.snapPoint(penPoint);
    constrain45(startPoint, penPoint);
    var w = penPoint.x - startPoint.x;
    var h = penPoint.y - startPoint.y;
    fl.drawingLayer.moveTo(startPoint.x, startPoint.y - h *
➤shaftWidth);
    fl.drawingLayer.lineTo(startPoint.x + w * shaftLength, startPoint.
➤y - h * shaftWidth);
    fl.drawingLayer.lineTo(startPoint.x + w * shaftLength, startPoint.
➤y - h);
    fl.drawingLayer.lineTo(penPoint.x, startPoint.y);
    fl.drawingLayer.lineTo(startPoint.x + w * shaftLength, penPoint.y);
  }
}

```

```

        fl.drawingLayer.lineTo(startPoint.x + w * shaftLength, startPoint.
    ↪y + h * shaftWidth);
        fl.drawingLayer.lineTo(startPoint.x, startPoint.y + h *
    ↪shaftWidth);
        fl.drawingLayer.lineTo(startPoint.x, startPoint.y - h *
    ↪shaftWidth);
        fl.drawingLayer.endFrame();
    }
}

```

We won't go into detail about how we figured each point. If you're intrigued, you might want to look at each line and graph it out on a piece of paper to double-check.

Finally, you do essentially the same thing in `mouseUp` to add the final lines to the document:

```

function mouseUp(){
    fl.drawingLayer.endDraw();
    var endPoint = fl.tools.penLoc;
    transformPoint(endPoint, fl.getDocumentDOM().viewMatrix);
    endPoint = fl.tools.snapPoint(endPoint);
    constrain45(startPoint, endPoint);
    var curr_doc = fl.getDocumentDOM();
    var w = endPoint.x - startPoint.x;
    var h = endPoint.y - startPoint.y;
    var p1 = {x:startPoint.x, y:startPoint.y - h * shaftWidth};
    var p2 = {x:startPoint.x + w * shaftLength, y:startPoint.y - h *
    ↪shaftWidth};
    var p3 = {x:startPoint.x + w * shaftLength, y:startPoint.y - h};
    var p4 = {x:endPoint.x, y:startPoint.y};
    var p5 = {x:startPoint.x + w * shaftLength, y:endPoint.y};
    var p6 = {x:startPoint.x + w * shaftLength, y:startPoint.y + h *
    ↪shaftWidth};
    var p7 = {x:startPoint.x, y:startPoint.y + h * shaftWidth};
    curr_doc.addNewLine(p1, p2);
    curr_doc.addNewLine(p2, p3);
    curr_doc.addNewLine(p3, p4);
    curr_doc.addNewLine(p4, p5);
    curr_doc.addNewLine(p5, p6);
    curr_doc.addNewLine(p6, p7);
    curr_doc.addNewLine(p7, p1);
}

```

In this case, since you need to use point objects, you go through and create the seven points first, and then just draw the lines between them.

Now, although this code works—it successfully draws an arrow as specified on the screen (you can play with the options for shaft length and width and get an idea how those work)—it isn't pretty. Indeed, you can improve upon it quite a bit. One of the major drawbacks to creating a shape with lines like this is that there is no simple way to fill it.

You've only created the outline of an arrow, and it's up to the user to fill it. For the grid, this isn't bad, as it's just a grid of lines. But you would expect a shape like this to have a fill.

Also, as you get into more and more complicated shapes, it would be ridiculous to have to write all the code to re-create the shape twice—once for the drawing layer and once for the document. For a simple shape like this with just seven lines, it's already getting pretty messy.

## The Path

Here comes a new object to your rescue: path is simply an object that holds a bunch of points and/or curves. For this tool, you'll concentrate on using points. Here are the simple steps to using it:

1. Make a new path with the drawing layer function `newPath`. This returns a reference to the newly created path.
2. Add some points to it with the path object function `addPoint(x, y)`.
3. If you want a closed path, you can add a line from the last point to the first one with the path function `close`.
4. Finally, draw the path to the drawing layer with the drawing layer function `drawPath`. This takes as an argument the name of the path you're drawing.

Implementing a path, your `mouseMove` function thus becomes as follows:

```
function mouseMove(){
  if(fl.tools.mouseIsDown){
    fl.drawingLayer.beginFrame();
    var penPoint = fl.tools.penLoc;
    transformPoint(penPoint, fl.getDocumentDOM().viewMatrix);
    penPoint = fl.tools.snapPoint(penPoint);
    constrain45(startPoint, penPoint);
    var w = penPoint.x - startPoint.x;
    var h = penPoint.y - startPoint.y;
    var points = new Array();
    points[0] = {x:startPoint.x, y:startPoint.y - h * shaftWidth};
    points[1] = {x:startPoint.x + w * shaftLength, y:startPoint.y - h *
    ↪shaftWidth};
    points[2] = {x:startPoint.x + w * shaftLength, y:startPoint.y - h};
    points[3] = {x:penPoint.x, y:startPoint.y};
    points[4] = {x:startPoint.x + w * shaftLength, y:penPoint.y};
    points[5] = {x:startPoint.x + w * shaftLength, y:startPoint.y + h *
    ↪shaftWidth};
    points[6] = {x:startPoint.x, y:startPoint.y + h * shaftWidth};
    arrow_path = fl.drawingLayer.newPath();
    for (var i = 0; i < points.length; i++) {
      arrow_path.addPoint(points[i].x, points[i].y);
    }
    arrow_path.close();
    fl.drawingLayer.drawPath(arrow_path);
  }
}
```

```

        fl.drawingLayer.endFrame();
    }
}

```

You see you first create seven points in an array called `points`. Then you create a path called `arrow_path` and add your seven points to it. Then you close it and draw it—simple enough. But the real advantage to paths is about to be revealed. Notice that when you declare `arrow_path` in the line, like this:

```
arrow_path = fl.drawingLayer.newPath();
```

you don't use the keyword `var`. Doing so would make the variable a local variable, only available from within the `mouseMove` function. When the function is over, it would be destroyed. You want it to stick around, because you have one more job for it.

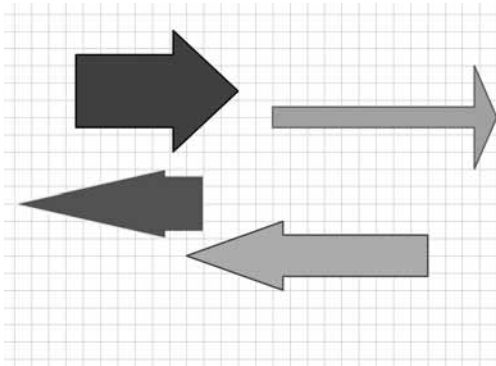
Remember the last version's `mouseUp` function? There was something like 25 lines of code just to draw seven lines on the stage. Well, here's your new version using paths:

```

function mouseUp(){
    fl.drawLayer.endDraw();
    arrow_path.makeShape();
}

```

No more double work! You simply call the `makeShape` method of the path object, and it is drawn to stage (see Figure 3-9).



**Figure 3-9.**  
Some of the arrows you can draw with the Arrow tool

There is a little more you should know about this command. It actually has two arguments, both Boolean. The first will suppress the fill of the shape if `true`, and the second will suppress the stroke. So, if you just wanted an outline arrow, you could say

```
arrow_path.makeShape(true, false);
```

And likewise, if you wanted just an arrow shape with no outline, you could say

```
arrow_path.makeShape(false, true);
```

Of course, setting them both to true would result in nothing being drawn. Also, if you want to draw both the outline and the stroke, just omit both arguments—they will default to false.

Now, this would all be great, except Macromedia decided to throw you a little curveball. Remember when you learned about the transformation matrix stuff? (Come on, it was only a few pages back!) You learned that inside a transformed movie clip, `penLoc` and `penDownLoc` report coordinates local to that clip, and the various drawing tools use global stage coordinates, so you needed to make a conversion. Well, it turns out that path operations work with the local, transformed coordinates. While the preceding code works fine for drawing on the main stage, inside a movie clip, it might go very much awry. Try it. See, back to where you started!

The simple solution would seem to be to just not transform the points `penPoint` and `startPoint`. If you want to try that, go ahead. You'll see that it is a bit better, but still not perfect.

The solution is to transform the original points you're drawing from and to (`startPoint` and `penPoint`), and then set all the in-between points, as you're doing. Then, you need to kind of "untransform" them. In other words, take them from global stage coordinates, and convert them back to local movie clip coordinates.

So we wrote two new functions, `transform` and `invert`, which handle these actions. The `transform` function is a little different from the `transformPoint` function you've been using, which is why we gave it a different name. These functions are pretty short, but would take several pages to explain. Fortunately, you don't need to understand them, just use them! Here they are:

```
function transform( pt, mat )
{
    var x = pt.x*mat.a + pt.y*mat.c;
    var y = pt.x*mat.b + pt.y*mat.d;
    pt.x = x;
    pt.y = y;
}

function invert( pt, mat )
{
    var det = mat.a*mat.d-mat.b*mat.c;
    var x = pt.x * mat.d/det + pt.y * -mat.c/det;
    var y = pt.x * -mat.b/det + pt.y * mat.a/det;
    pt.x = x;
    pt.y = y;
}
```

You'll use the `transform` function in place of `transformPoints` in both `mouseDown` and `mouseMove` to transform `startPoint` and `penPoint`. Then, you'll invert each point before you add it to the path. Here are the two final mouse functions showing the changes:

```

function mouseDown() {
    startPoint = fl.tools.penDownLoc;
    transform(startPoint, fl.getDocumentDOM().viewMatrix);
    startPoint = fl.tools.snapPoint(startPoint);
    fl.drawingLayer.beginDraw();
}
function mouseMove(){
    if(fl.tools.mouseIsDown){
        fl.drawingLayer.beginFrame();
        var penPoint = fl.tools.penLoc;
        transform(penPoint, fl.getDocumentDOM().viewMatrix);
        penPoint = fl.tools.snapPoint(penPoint);
        constrain45(startPoint, penPoint);
        var w = penPoint.x - startPoint.x;
        var h = penPoint.y - startPoint.y;
        var points = new Array();
        points[0] = {x:startPoint.x, y:startPoint.y - h * shaftWidth};
        points[1] = {x:startPoint.x + w * shaftLength, y:startPoint.y - h *
        ↪shaftWidth};
        points[2] = {x:startPoint.x + w * shaftLength, y:startPoint.y - h};
        points[3] = {x:penPoint.x, y:startPoint.y};
        points[4] = {x:startPoint.x + w * shaftLength, y:penPoint.y};
        points[5] = {x:startPoint.x + w * shaftLength, y:startPoint.y + h *
        ↪shaftWidth};
        points[6] = {x:startPoint.x, y:startPoint.y + h * shaftWidth};
        arrow_path = fl.drawingLayer.newPath();
        for (var i = 0; i < points.length; i++) {
            invert(points[i], fl.getDocumentDOM().viewMatrix);
            arrow_path.addPoint(points[i].x, points[i].y);
        }
        arrow_path.close();
        fl.drawingLayer.drawPath(arrow_path);
        fl.drawingLayer.endFrame();
    }
}
}

```

And once again, you're back on the straight and narrow!

## Setting the Properties Panel

In all the tools you've created so far, you've let the Properties panel just kind of do its own thing. Let's learn how to rope it in and control it.

To demonstrate just what you can do with it, you'll create a brand new tool. This will create a *callout*, which is essentially a line with some text attached to it. Callouts are useful to annotate diagrams or pictures with intricate parts with pointers and labels. A line comes off a particular item and connects to a text box off on the side explaining what the item is. In this case, you'll implement a simple click-drag functionality to create the pointer, and then add text after some user input.

## EXTENDING MACROMEDIA FLASH MX 2004

Since you'll be creating some shapes and some text, you'll need access to the properties for both. If you observe the Properties panel as you work in Flash, you'll notice it has three main modes:

1. **Shape** mode: This is what you see when you select any of the drawing tools. You have options for stroke color, size, style, and fill color (see Figure 3-10).

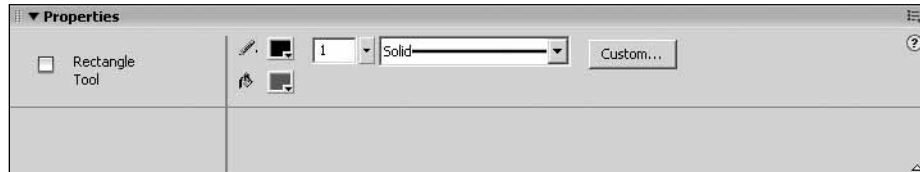


Figure 3-10. Properties panel in shape mode

2. **Text** mode: Whenever you select the text tool or are editing a text box, you see the large array of options for creating or modifying text (see Figure 3-11).

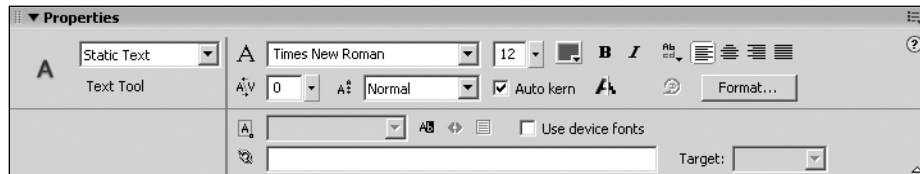


Figure 3-11. Properties panel in text mode

3. **Movie** mode: This is what appears at pretty much any other time. It actually has several different styles depending on what is currently selected—symbol, frame, stage—but basically shows the various options available for that selection (see Figure 3-12).

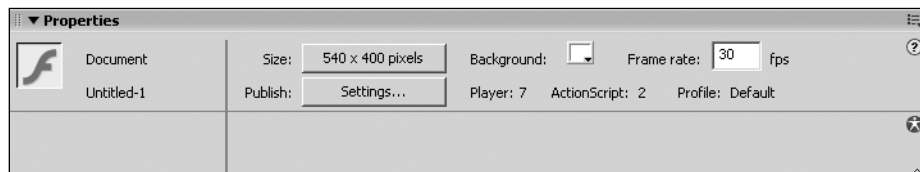


Figure 3-12. Properties panel in movie mode

You can manually set which mode the Properties panel will go into via the `toolObj` function, `setPI` (PI for Property Inspector, as this panel is also known). This function takes a string containing one of the following: `shape`, `text`, or `movie`. The Properties panel will go into that mode and stay there until something else happens to change it, such as selecting an object or another tool. If you don't set the Properties panel, it will default to `shape`.

In the Callout tool, you'll first set the Properties panel to `shape` mode in the `activate` function. This will allow you to draw the line for the callout. Once the line is drawn, you switch it to `text` mode so users can choose their text style.

You'll use a few other advanced features in this file as well, such as a JavaScript alert to notify users they should set their font preferences, and an XML to UI dialog box to get the text to add to the callout.

You'll also make your first use of dynamic functions. Up to now, all of your event-handling functions (`mouseDown`, `mouseUp`, etc.) were permanently assigned. Since you have different modes in this file, you need different behaviors at different times. You can do this by swapping functions in and out for the event handlers.

First let's configure the tool and activate it (don't forget to create an icon too):

```
function configureTool(){
    var curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setIcon("callout.png");
    curr_toolObj.setMenuString("Callout Tool");
    curr_toolObj.setToolName("Callout Tool");
    curr_toolObj.setToolTip("Callout Tool");
}
function activate(){
    curr_toolObj = fl.tools.activeTool;
    curr_toolObj.setPI("shape");
    mouseDown = mouseDownFunc;
    mouseUp = mouseUpFunc;
    mouseMove = mouseMoveFunc;
}
```

The `configureTool` function should be nothing surprising. You're not using an option file here, so that gets left out. Note how you set the Properties panel to shape mode in the second line of the `activate` function, and then set the three mouse event-handler functions to other function names. You'll create those next:

```
function mouseDownFunc(){
    startPoint = fl.tools.penDownLoc;
    transform(startPoint, fl.getDocumentDOM().viewMatrix);
    startPoint = fl.tools.snapPoint(startPoint);
    fl.drawingLayer.beginDraw();
}
```

Again, the `mouseDownFunc` should be nothing new. It's exactly what you used earlier for `mouseDown`. The advantage to assigning it this way is that you can change it later to something else, and then change it back. Note that since you'll be using paths again, you'll use the new `transform` function rather than `transformPoint`.

```
function mouseMoveFunc(){
    if(fl.tools.mouseIsDown){
        fl.drawingLayer.beginFrame();
        penPoint = fl.tools.penLoc;
        transform(penPoint, fl.getDocumentDOM().viewMatrix);
        penPoint = fl.tools.snapPoint(penPoint);
        penPoint = fl.tools.constrainPoint(startPoint, penPoint);
    }
}
```

```

        if(penPoint.x > startPoint.x){
            offset = -20;
        }
        else
        {
            offset = 20;
        }
        points = new Array();
        points[0] = {x:startPoint.x, y:startPoint.y};
        points[1] = {x:penPoint.x + offset, y:penPoint.y};
        points[2] = {x:penPoint.x, y:penPoint.y};
        points[3] = {x:penPoint.x, y:penPoint.y - 20};
        points[4] = {x:penPoint.x, y:penPoint.y + 20};
        call_path = fl.drawingLayer.newPath();
        for(var i=0;i<points.length;i++){
            invert(points[i], fl.getDocumentDOM().viewMatrix);
            call_path.addPoint(points[i].x, points[i].y);
        }
        fl.drawingLayer.drawPath(call_path);
        fl.drawingLayer.endFrame();
    }
}

```

The `mouseMoveFunc` function is also pretty basic. It just creates a few points calculated by the mouse position, inverts them, and adds them to a path. Same thing you did for the arrow, but a different shape.

```

function mouseUpFunc(){
    fl.drawingLayer.endDraw();
    call_path.makeShape();
    curr_toolObj.setPI("text");
    mouseUp = nullFunc;
    mouseMove = nullFunc;
    mouseDown = addTextBox;
    alert("Set font properties and click on the stage.");
}
function nullFunc(){
}

```

The `mouseUpFunc` function contains some important new features. The first couple of lines just end drawing mode and make a shape using the path you just made. Next, you change the Properties panel to text mode. Then you start messing with the mouse event handlers. At this point, you don't want anything to happen on `mouseUp` or `mouseMove`, so you assign those to an empty function called `nullFunc`, which you include here.

Then you set `mouseDown` to a function called `addTextBox`, which you'll see shortly. Finally, you alert users to set the font properties and tell them to click the stage when they're done (see Figure 3-13).

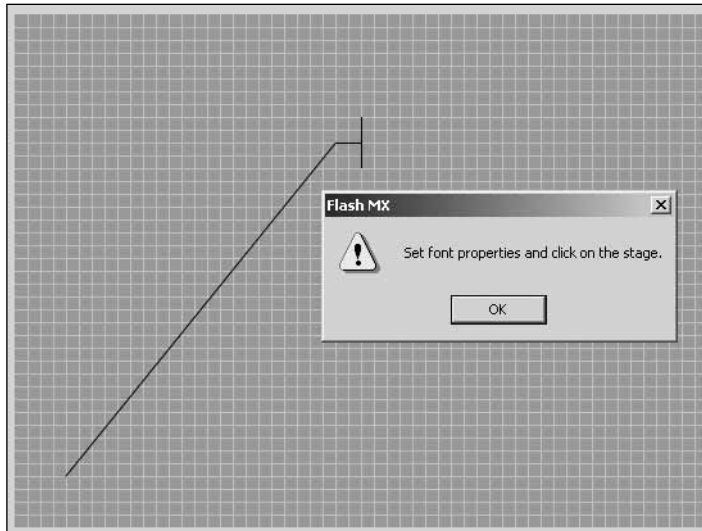


Figure 3-13. The alert box informing users what to do next

When users clear the alert, change the font settings, and then click the stage, this will activate your addTextBox function, which is right here:

```
function addTextBox(){
    curr_doc = fl.getDocumentDOM();
    testResult = curr_doc.xmlPanel(fl.configURI+"/Tools/callout.xml");
    if(testResult.dismiss == "accept"){
        curr_lib = curr_doc.library;
        var textName = "Callout Text";
        var tryCount = 0;
        while(curr_lib.itemExists(textName)){
            tryCount++;
            textName = "Callout Text" + tryCount;
        }
        curr_lib.addNewItem("movie clip", textName);
        curr_lib.editItem(textName);
        curr_doc.addNewText({top:0, left:0, right:10, bottom:10});
        var selArray = new Array();
        selArray[0] =
        ↪curr_doc.getTimeline().layers[0].frames[0].elements[0];
        curr_doc.selection = selArray;
        curr_text = curr_doc.selection[0];
        curr_text.autoExpand = true;
        curr_text.setTextString(testResult.text);
        curr_doc.exitEditMode();
        curr_lib.addItemToDocument({x:0, y:0}, textName);
        curr_doc.moveSelectionBy(penPoint);
        if(penPoint.x > startPoint.x){
```

```

        curr_doc.moveSelectionBy({x:curr_doc.selection[0].width/2+10,
➤y:0});
    }
    else {
        curr_doc.moveSelectionBy({x:-curr_doc.selection[0].width/2-10,
➤y:0});
    }
    curr_doc.selectNone();
}
else {
    alert("Callout Text cancelled.");
}
}
}

```

This is a biggie, so we'll describe it a little at a time. First you call an XML to UI dialog box and make sure the user clicked OK.

```

curr_doc = fl.getDocumentDOM();
testResult = curr_doc.xmlPanel(fl.configURI+"/Tools/callOut.xml");
if(testResult.dismiss == "accept"){

```

And here is the XML for the dialog box shown in Figure 3-14 (see callout.xml):

```

<dialog buttons="accept, cancel" title="Callout Tool Options">
  <hbox>
    <label value="Callout Text"/>
    <textbox id="text"/>
  </hbox>
</dialog>

```

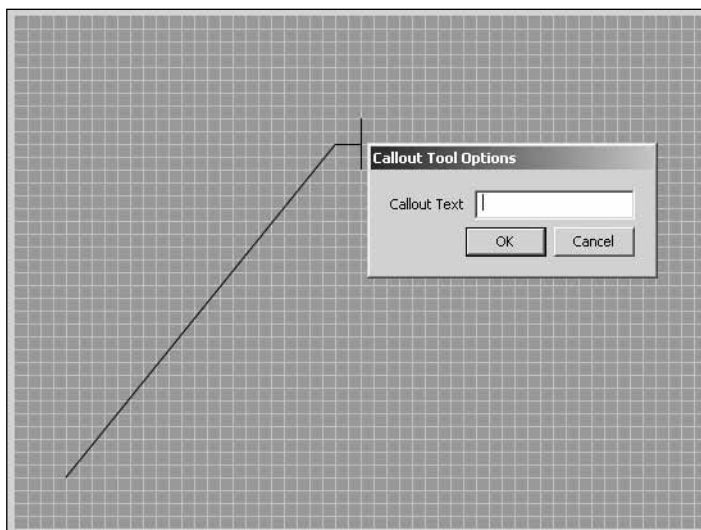


Figure 3-14. The XML to UI dialog box requesting the callout text

So the XML file just defines one property called text. Easy enough. Next, if the user did click OK, you create a new movie clip in the Library, finding a unique name first, and then start editing that movie clip.

```
curr_lib = curr_doc.library;
var textName = "Callout Text";
var tryCount = 0;
while(curr_lib.itemExists(textName)){
    tryCount++;
    textName = "Callout Text"; + tryCount;
}
curr_lib.addItem("movie clip", textName);
curr_lib.editItem(textName);
```

Once you're inside this new clip and editing it, you add a new text box and put it into the selection array.

```
curr_doc.addNewText({top:0, left:0, right:10, bottom:10});
var selArray = new Array();
selArray[0] =
    ↪curr_doc.getTimeline().layers[0].frames[0].elements[0];
curr_doc.selection = selArray;
curr_text = curr_doc.selection[0];
```

The size of the text box doesn't really matter, since you'll let it expand to allow for whatever text is put in it. Also, note that because this is a brand new movie clip, you know there is no other content. Thus you can get a reference to the text box by selecting the current timeline, layer zero, frame zero, element zero. You now have a variable, `curr_text`, that refers to this text box. You change some settings, add some text (the text returned from the XML to UI dialog box), exit edit mode, and add the item to the stage at 0, 0.

```
curr_text.autoExpand = true;
curr_text.setTextString(testResult.text);
curr_doc.exitEditMode();
curr_lib.addItemToDocument({x:0, y:0}, textName);
```

At this point, the newly added movie clip will be the only thing selected, and it will be at 0, 0. You need to move it to the end of the callout line. The object `penPoint` refers to the end of the line you just drew. You can move the movie clip to that point using `curr_doc.moveSelectionBy(penPoint)`, as follows:

```
curr_doc.moveSelectionBy(penPoint);
```

Then, depending if the callout line is going left or right, you slide it over half its width one way or another (plus another 10 pixels as an offset). You then use `selectNone()` for a truly professional finish.

```
if(penPoint.x > startPoint.x){
    curr_doc.moveSelectionBy({x:curr_doc.selection[0].width/2+10,
    ↪y:0});
}
else {
```

```

        curr_doc.moveSelectionBy({x:-curr_doc.selection[0].width/2-10,
    ➔y:0});
    }
    curr_doc.selectNone();
}

```

Finally, you wrap up by alerting users that the action was cancelled if they chose Cancel in the dialog box.

```

    else {
        alert("Callout Text cancelled.");
    }
}

```

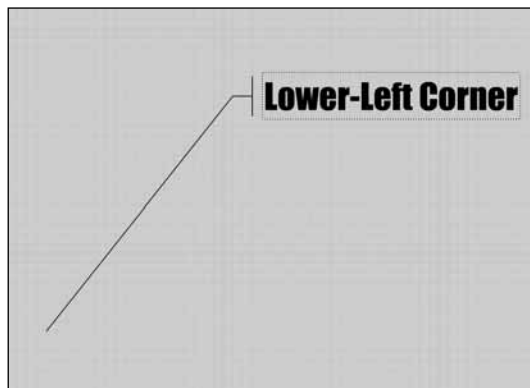
The rest of the functions that follow are nothing new at all, but we'll include them here for completeness:

```

function setCursor(){
    fl.tools.setCursor(0);
}
function transformPoint(aPoint, matrix){
    var x = aPoint.x * matrix.a + aPoint.y * matrix.c + matrix.tx;
    var y = aPoint.x * matrix.b + aPoint.y * matrix.d + matrix.ty;
    aPoint.x = x;
    aPoint.y = y;
}
function constrain45(p1, p2)
{
    if (fl.tools.shiftIsDown)
    {
        var dx = Math.abs(p2.x - p1.x);
        var dy = Math.abs(p2.y - p1.y);
        var offset = Math.max(dx, dy);
        if(p2.y < p1.y)
        {
            p2.y = p1.y - offset;
        }
        else
        {
            p2.y = p1.y + offset;
        }
        if(p2.x > p1.x)
        {
            p2.x = p1.x + offset;
        }
        else
        {
            p2.x = p1.x - offset;
        }
    }
}
}

```

Test your new Callout tool to see it in action (see Figure 3-15).



**Figure 3-15.**  
The completed callout

3

We've covered most of what you need to know to build tools. Let's finish off with one final impressive example for inspiration.

## 3D Cube Tool

We don't really have the space to get into a big explanation of all the math involved in this next example tool, but we wanted to include it here as a brief look at just how neat these tools can be. This one creates a cube that you can rotate around in 3D space. Although the 3D perspective math is a bit complex, there are no real JSFL concepts here that we haven't thoroughly covered. We'll give you the functions one at a time and give a brief explanation of what is happening in each one. You can also refer to `cube3d.jsfl` to play around with the completed code.

```
function configureTool(){
    theTool = fl.tools.activeTool;
    theTool.setToolName("Cube3D");
    theTool.setIcon("Cube3D.png");
    theTool.setMenuString("Cube 3D Tool");
    theTool.setToolTip("Cube 3D Tool");
    theTool.setOptionsFile( "Cube3D.xml" );

    points = new Array();
    xp = new Array();
    yp = new Array();
    zp = new Array();
    xp = [-1, 1, 1, -1, -1, 1, 1, -1];
    yp = [-1, -1, 1, 1, -1, -1, 1, 1];
    zp = [-1, -1, -1, -1, 1, 1, 1, 1];
}
```

In `configureTool` you do your normal setup actions, and then create a few arrays. The `points` array holds the 3D points for each corner of the cube. `xp`, `yp`, and `zp` hold the relative positions of each point on each axis. The `Cube3D.xml` file is shown here:

```
<properties>
  <property name="X Size" variable="xSize" min="5" max="300"
  ➤defaultValue="100" type="Number" />
  <property name="Y Size" variable="ySize" min="5" max="300"
  ➤defaultValue="100" type="Number" />
  <property name="Z Size" variable="zSize" min="5" max="300"
  ➤defaultValue="100" type="Number" />
  <property name="Perspective" variable="pers" min="100" max="500"
  ➤defaultValue="200" type="Number" />
</properties>
```

Next:

```
function setPoints(){
  for(var i=0;i<8;i++){
    points[i] = new Object;
    points[i].x = xp[i]*xSize/2;
    points[i].y = yp[i]*ySize/2;
    points[i].z = zp[i]*zSize/2;
  }
}
```

The `setPoints` function just takes the relative coordinates in `xp`, `yp`, and `zp`, and multiplying them times the sizes retrieved from the Options dialog box, creates a 3D point for each corner, storing it in the `points` array.

```
function activate(){
  curr_toolObj = fl.tools.activeTool;
  xSize = curr_toolObj.xSize;
  ySize = curr_toolObj.ySize;
  zSize = curr_toolObj.zSize;
  pers = curr_toolObj.pers;
  if(xSize>pers){
    xSize = curr_toolObj.xSize=pers;
  }
  if(ySize>pers){
    ySize = curr_toolObj.ySize=pers;
  }
  if(zSize>pers){
    zSize = curr_toolObj.zSize=pers;
  }
  setPoints();
}
function notifySettingsChanged(){
  curr_toolObj = fl.tools.activeTool;
  xSize = curr_toolObj.xSize;
```

```

ySize = curr_toolObj.ySize;
zSize = curr_toolObj.zSize;
pers = curr_toolObj.pers;
if(xSize>pers){
  xSize = curr_toolObj.xSize=pers;
}
if(ySize>pers){
  ySize = curr_toolObj.ySize=pers;
}
if(zSize>pers){
  zSize = curr_toolObj.zSize=pers;
}
}

```

The activate and notifySettingsChanged functions are the same. In fact, if you wanted to you could create a function called getSettings and dynamically assign this to both of these handlers. All this does is grab the properties out of the Options dialog box and adjust some of them if necessary.

```

function mouseDown(){
  fl.drawingLayer.beginDraw();
}
function setCursor(){
  fl.tools.setCursor(0);
}

```

Nothing very complicated here. In this file, snapping and constraining doesn't really make sense, so you don't need to use them. Since you didn't do anything to it, you can just grab fl.tools.penDownLoc later when needed.

```

function mouseMove(){
  if(fl.tools.mouseIsDown){
    // get the view matrix
    var viewMat = fl.getDocumentDOM().viewMatrix;
    // get the distance moved on x and y
    var dx = fl.tools.penLoc.x - fl.tools.penDownLoc.x;
    var dy = fl.tools.penLoc.y - fl.tools.penDownLoc.y;
    // using these distances as angles to rotate the cube,
    // get the sine and cosine of the angles
    var cosx = Math.cos(dy*.01);
    var sinx = Math.sin(dy*.01);
    var cosy = Math.cos(dx*.01);
    var siny = Math.sin(dx*.01);
    // loop through each point
    for(var i = 0; i<8; i++){
      // some fancy math to rotate the points in 3D
      var x1 = points[i].x*cosy - points[i].z * siny;
      var z1 = points[i].z*cosy + points[i].x * siny;
      var y1 = points[i].y*cosx - z1*sinx;
      var z2 = z1*cosx + points[i].y * sinx;
    }
  }
}

```

```

        // apply perspective
        var scale = pers/(pers+z2);
        points[i].xp = x1 * scale + fl.tools.penDownLoc.x;
        points[i].yp = y1 * scale + fl.tools.penDownLoc.y;
        // apply the transformation matrix
        transformPoint(points[i], viewMat);
    }
    // draw all the lines that make the cube
    fl.drawingLayer.beginFrame();
    fl.drawingLayer.moveTo(points[0].xp, points[0].yp);
    fl.drawingLayer.lineTo(points[1].xp, points[1].yp);
    fl.drawingLayer.lineTo(points[2].xp, points[2].yp);
    fl.drawingLayer.lineTo(points[3].xp, points[3].yp);
    fl.drawingLayer.lineTo(points[0].xp, points[0].yp);
    fl.drawingLayer.lineTo(points[4].xp, points[4].yp);
    fl.drawingLayer.lineTo(points[5].xp, points[5].yp);
    fl.drawingLayer.lineTo(points[6].xp, points[6].yp);
    fl.drawingLayer.lineTo(points[7].xp, points[7].yp);
    fl.drawingLayer.lineTo(points[4].xp, points[4].yp);
    fl.drawingLayer.moveTo(points[1].xp, points[1].yp);
    fl.drawingLayer.lineTo(points[5].xp, points[5].yp);
    fl.drawingLayer.moveTo(points[2].xp, points[2].yp);
    fl.drawingLayer.lineTo(points[6].xp, points[6].yp);
    fl.drawingLayer.moveTo(points[3].xp, points[3].yp);
    fl.drawingLayer.lineTo(points[7].xp, points[7].yp);
    fl.drawingLayer.endFrame();
}
}

```

Wow! Don't get scared. First you grab a reference to the `viewMatrix`, since you'll be using it a bunch of times. Then you figure out how much the mouse has moved by subtracting `penDownLoc` from `penLoc`. You use that in a bunch of fancy trigonometry to rotate the 3D points in the `points` array in relation to how much the mouse has moved.

Then you do some 3D perspective to locate the 2D screen positions of the 3D points and apply the transform matrix to those points. For a full discussion of all the trigonometry and perspective code used here, look into *Macromedia Flash MX Studio*, also available from friends of ED.

Finally, you draw a bunch of lines to make the cube.

```

function mouseUp(){
    fl.drawLayer.endDraw();
    fl.getDocumentDOM().addNewLine({x:points[0].xp, y:points[0].yp},
    ↪{x:points[1].xp, y:points[1].yp});
    fl.getDocumentDOM().addNewLine({x:points[1].xp, y:points[1].yp},
    ↪{x:points[2].xp, y:points[2].yp});
    fl.getDocumentDOM().addNewLine({x:points[2].xp, y:points[2].yp},
    ↪{x:points[3].xp, y:points[3].yp});
}

```

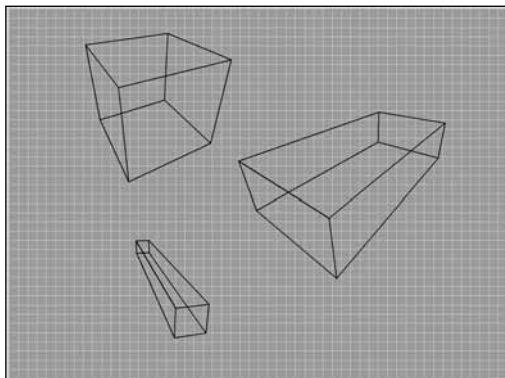
```

    fl.getDocumentDOM().addNewLine({x:points[3].xp, y:points[3].yp},
    ↪{x:points[0].xp, y:points[0].yp});
    fl.getDocumentDOM().addNewLine({x:points[4].xp, y:points[4].yp},
    ↪{x:points[5].xp, y:points[5].yp});
    fl.getDocumentDOM().addNewLine({x:points[5].xp, y:points[5].yp},
    ↪{x:points[6].xp, y:points[6].yp});
    fl.getDocumentDOM().addNewLine({x:points[6].xp, y:points[6].yp},
    ↪{x:points[7].xp, y:points[7].yp});
    fl.getDocumentDOM().addNewLine({x:points[7].xp, y:points[7].yp},
    ↪{x:points[4].xp, y:points[4].yp});
    fl.getDocumentDOM().addNewLine({x:points[0].xp, y:points[0].yp},
    ↪{x:points[4].xp, y:points[4].yp});
    fl.getDocumentDOM().addNewLine({x:points[1].xp, y:points[1].yp},
    ↪{x:points[5].xp, y:points[5].yp});
    fl.getDocumentDOM().addNewLine({x:points[2].xp, y:points[2].yp},
    ↪{x:points[6].xp, y:points[6].yp});
    fl.getDocumentDOM().addNewLine({x:points[3].xp, y:points[3].yp},
    ↪{x:points[7].xp, y:points[7].yp});
  }
  function transformPoint(pt, mat){
    var x = pt.xp*mat.a + pt.yp * mat.c + mat.tx;
    var y = pt.xp*mat.b + pt.yp * mat.d + mat.ty;
    pt.xp = x;
    pt.yp = y;
    return;
  }
}

```

In the `mouseUp` function, you essentially duplicate the `mouseMove` drawing code, using the document drawing methods instead. If you want to try to convert this into a path, go right ahead. The problem is that it isn't a simple path that goes from one point, through some others, and ends somewhere. You'd either have to make several paths to form one cube, or do some backtracking. That may be more efficient, but this way seemed a bit more obvious to us, even if more wordy.

And in Figure 3-16, you can see some of the 3D cubes you can draw using this new tool.



**Figure 3-16.**  
Some cubes made with the  
3D Cube tool

## Packaging Extensions

At this point, it would be a good idea to look into how you can package your extensions (commands, tools, effects, behaviors) for distribution. Whether you plan to sell them, give them away, or just keep them around for your own use, packaging them correctly goes a long way toward having them be truly professional. You could simply zip up the JSFL, PNG, XML, and any other files in an archive and let others figure out where to install it. For a tool, this would include having users figure out how to get it onto the toolbar. You might end up doing a lengthy instruction and installation manual and still have users get it wrong.

Fortunately, Macromedia has given you a way to package up extensions for distribution and installation. All end users have to do is double-click a file and everything gets installed in the right place. It's required that users have the Macromedia Extension Manager installed on their system. But even that is a breeze to download and install. In fact, this is the same tool that you'll use to package your extensions, so why don't you go ahead and download and install the latest version right now, if you don't already have it. You can find it at [www.macromedia.com/exchange/em\\_download/](http://www.macromedia.com/exchange/em_download/).

The latest version as of this writing is version 1.6, and you'll need at least that version to be able to package Flash MX 2004 extensions. Downloading and installing the Extension Manager couldn't be easier. Just download and run the file, and it will run a standard installation program. Now you're ready to package your extensions.

Flash MX users who became knowledgeable with creating and distributing components will already know that a packaged extension is known as a Macromedia Extension Package (MXP) file, and that the Extension Manager is used to create MXPs. First, though, you have to describe to the Extension Manager exactly what files will be packaged in the MXP, where they will go, what they are for, and some general information about what the extension does and who made it. This is all done in XML, and the resulting XML file is called a Macromedia Extension Installation (MXI) file.

Following is a sample MXI file for the 3D Cube tool that you've just seen:

```
<macromedia-extension name="3D Cube Tool" version="1.0.0" type="flash
➤tool" requires-restart="true">
  <description><![CDATA[A drawing tool for your tool bar. Creates a 3D
➤cube that you can rotate in real time. User adjustable size and
➤perspective.]]></description>
  <ui-access><![CDATA[You will find this tool on your tool bar]]>
➤</ui-access>
  <products>
    <product name="Flash" version="7" primary="true" />
  </products>
  <author name="Keith Peters" />
  <files>
    <file source="cube3d.jsfl" destination="$Flash\Tools" />
    <file source="cube3d.xml" destination="$Flash\Tools" />
    <file source="cube3d.png" destination="$Flash\Tools" />
  </files>
```

```

<configuration-changes>
  <toolbar-changes>
    <toolbar-item-insert name="Cube3D" depth="0" position="7" />
  </toolbar-changes>
</configuration-changes>
</macromedia-extension>

```

This is pretty much all you'll need for any extension. In fact, extensions other than tools will be even shorter. Let's walk through each tag:

```

<macromedia-extension name="3D Cube Tool" version="1.0.0" type="flash
  ➤tool" requires-restart="true">

```

This is the root-level tag. It contains, as attributes, the name of the extension, the version, what type of extension it is (tool, command, behavior, effect), and whether or not Flash should be restarted after installation. You should set this to true, as the Extension Manager installs extensions into a special folder called First Run. When Flash starts up, it copies anything in the First Run folder to the appropriate configuration directory. Until this happens, the extension will not be in the right place and will not be available.

```

  <description><![CDATA[A drawing tool for your tool bar. Creates a 3D
  ➤cube that you can rotate in real time. User adjustable size and
  ➤perspective.]]></description>

```

The description tag is just that, a description of what the extension is. This is enclosed in a CDATA tag exactly as shown. Other than that it is pretty free form. This data will be displayed to users by the Extension Manager when your extension is being installed.

```

  <ui-access><![CDATA[You will find this tool on your tool bar]]>
  ➤</ui-access>

```

The <ui-access> tag is pretty similar, and just tells users how to access your tool. You can tell them where it is on the toolbar, or which menu, or how to locate Timeline Effects or behaviors. Again, this is shown by the Extension Manager during the installation.

```

  <products>
    <product name="Flash" version="7" primary="true" />
  </products>

```

Next is a products section, which will contain all the products that the extension is valid for. For the most part, this will probably just contain one <product> tag giving the name of the product, Flash, and its version (MX 2004 is version 7). The primary attribute is for extensions that could be used on more than one product to say which is the primary product. Since Flash is the only product here, it is of course the primary one!

```

  <author name="Keith Peters" />

```

The <author> tag is self-explanatory. Take credit for all your hard work!

```

  <files>
    <file source="cube3d.jsfl" destination="$Flash\Tools" />

```

```

    <file source="cube3d.xml" destination="$Flash\Tools" />
    <file source="cube3d.png" destination="$Flash\Tools" />
</files>

```

Then you have the listing of files, in the <files> tag. Each file gets its own <file> tag. List the source file that you want packaged, and the destination of where you want it to go. For the source, the path can be relative to where the MXI file will be. The preceding syntax assumes that the three files are in the same directory as the MXI file. The destination uses the term \$Flash to refer to the configuration directory where the files will be installed. To put them in a subdirectory, such as Tools, simply tack that on.

```

<configuration-changes>
  <toolbar-changes>
    <toolbar-item-insert name="Cube3D" depth="0" position="7" />
  </toolbar-changes>
</configuration-changes>

```

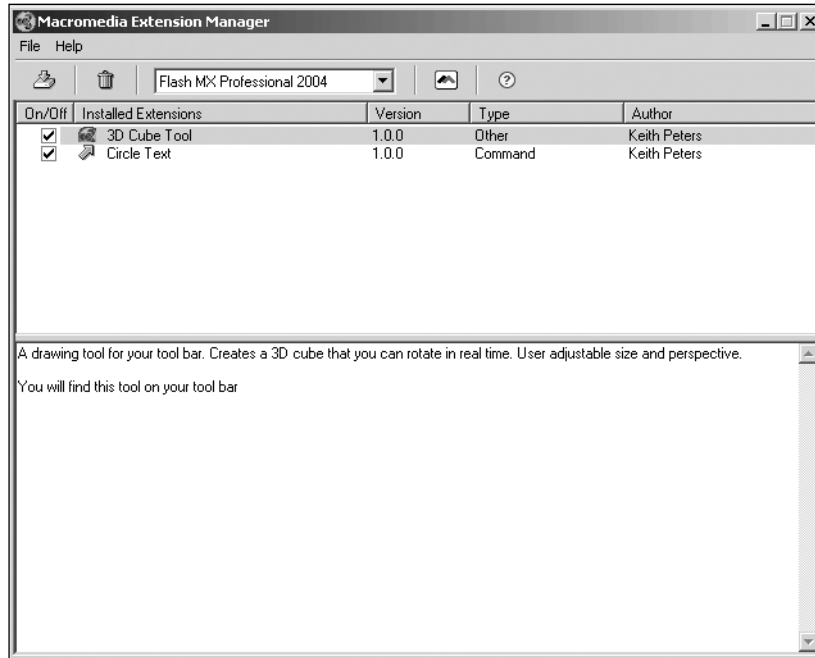
The <configuration-changes> tag only applies to tools. Commands, effects, and behaviors all automatically insert themselves in the proper menus as soon as they are added and reloaded. Tools either have to be manually placed on the toolbar, or can be automatically installed with the preceding XML. Inside <configuration-changes>, you have a <toolbar-changes> tag. Inside that you have a <toolbar-item-insert> tag. You give it the name of the tool, which is the name as specified in the configureTool function under setToolName. Depth refers to the level of the drop-down menu where the tool icon will appear. The position is which slot on the toolbar you want the tool to be added to. You count them starting from zero and going across and then down. So here, position seven is under the rectangle tool.

That does it for your MXI file. Save it with a file extension .mxi, and you're ready to create an MXP file.

Start up the Extension Manager. It will show you a list of installed extensions, if you have any installed of course—if this is your first time with the Extension Manager, then it's likely to be empty. Now go to the File menu and choose Package Extension. A file dialog box will open asking you to choose which extension to package. Navigate to wherever your MXI file is located, and you'll see the available MXI files to package.

Choose your MXI file and click OK. You'll then be asked where and under what name to save the final MXP file. If all goes well, you'll be told that the extension has been successfully created. If you get any errors, it usually means an error in your XML, or a file that is referenced couldn't be located.

The final result is an MXP file that, when double-clicked, will install the extension to the correct location and tell the user to restart Flash, if it is already running. Remember that MXP files aren't self-executable programs; you'll have to inform your customers or colleagues that they will need to install the Extension Manager, shown in Figure 3-17, in order to install your extension.



**Figure 3-17.** The Extension Manager

There you have it, a nice packaged extension. Be sure to see the additional documentation at Macromedia's website for additional features such as adding documentation, but the preceding instructions are enough for you to create a simple, workable packaged extension.

## Summary

Well, you've made some huge leaps forward in this chapter; over your study of how you create custom-built tools, you've actually learned the bulk of the basic JavaScript API. And to cap it all, you've seen how to package your tools into neat little files that enable easy distribution. By now, we're sure you've started experimenting with your own custom tools.

In the next chapter, we'll push it up another notch or two and show you the specialized techniques of creating Timeline Effects.