

Flash CS4



Chris Grover with
E.A. Vander Veer

POGUE PRESS™
O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

This excerpt is protected by copyright law. It is your responsibility to obtain permissions necessary for any proposed use of this material. Please direct your inquiries to permissions@oreilly.com.

Testing and Debugging

Testing your animation is a lot like filing your income taxes. Both can be tedious, time-consuming, and frustrating—but they’ve got to be done. Even if your animation is short, straightforward, and you’ve whipped out 700 exactly like it over the past 2 years, you still need to test it before you release it into the world. Why? Murphy’s Law: Anything that *can* go wrong *will* go wrong. Choosing a motion tween when you meant to choose a shape tween, adding content to a frame instead of a keyframe, tying actions to the wrong frame or object, or mistyping an ActionScript keyword are just a few of the ways a slip of your fingers can translate into a broken animation. And it’s far better that you find out about these problems *before* your audience sees your handiwork rather than after.

Throughout this book, you’ve seen examples of testing an animation using the Control → Test Movie option (for example, Figure 18-5). This chapter expands on that simple test option, plus it shows you how to test animation playback at a variety of connection speeds. And if you’ve added ActionScript to your animation, this chapter shows you how to unsnarl uncooperative ActionScript code using Flash’s debugging tools.

Testing Strategies

All your audience ever sees is the finished product, not your intentions. So no matter how sophisticated your animation or how cleverly constructed your ActionScript code, if you don’t test your animation and make sure it works the way you want it to, all your hard work will have been in vain.

The following section shows you how to prepare for testing from the very beginning by following good Flash development policies. Also, you find out the differences between testing on the stage and testing in Flash Player, along with tips for testing your animation in a Web browser.

Planning Ahead

The more complex your animation, the more you need a thorough plan for testing it. Few of the guidelines in the next two sections are specific to testing in Flash. Instead, they're tried-and-true suggestions culled from all walks of programming life. Following them pays off in higher-quality animations and reduced time spent chasing bugs.

Ideally, you should begin thinking about testing before you've created a single frame of your animation. Here are some pre-animation strategies that pay off big:

Separate potentially troublesome elements

ActionScript actions are very powerful, but they can also cause a lot of grief. Get into the habit of putting them into a separate layer named "actions", at the top of your list of layers, so that you'll always know where to find it. Putting all your labels into a separate layer (named "layers") and all your sounds into a layer (named "sounds") is a good idea, too.

Reuse as much as possible

Instead of cutting and pasting an image or a series of frames, create a graphic symbol and reuse it. That way, if a bug raises its ugly head, you'll have fewer places to go hunting. You can cut down on bugs by reusing ActionScript code, too. Instead of attaching four similar ActionScript actions to four different frames or buttons, create a single ActionScript method (also called *function*; see page 392) and call it four times.

Be generous with comments

Before you know it, you'll forget which layers contain which images, why you added certain actions to certain objects, or even why you ordered your ActionScript statements the way you did. In addition to adding descriptive comments to all of the actions you create, get in the habit of adding an overall comment to the first frame of your animation. Make it as long as you want and be sure to mention your name, the date, and anything else pertinent you can think of. You create a comment in ActionScript two different ways, as shown below.

```
// This is an example of a single-line ActionScript comment.
```

```
/* This type of ActionScript comment can span more than one line. All you  
have to remember is to begin your multi-line comment with a slash-asterisk  
and end it with an asterisk-slash, as you see here. */
```

Stick with consistent names

Referring to a background image in one animation as “bg,” in another animation as “back_ground,” and in still another as “Background” is just asking for trouble. Even if you don’t have trouble remembering which is which, odds are your office teammates will—and referring to an incorrectly spelled variable in ActionScript causes your animation to misbehave quietly. In other words, type *Backgruond* instead of *Background*, and Flash doesn’t pop up an error message; your animation just looks or acts odd for no apparent reason. Devise a naming convention you’re comfortable with and stick with it. For example, you might decide always to use uppercase, lowercase, or mixed case. You might decide always to spell words out, or always to abbreviate them the same way. The particulars don’t matter as much as your consistency in applying them.

Note: Capitalization counts. Because Flash is case-sensitive, it treats *background*, *Background*, and *BACKGROUND* as three different names.

Techniques for Better Testing

The following strategies are crucial if you’re creating complex animations as part of a development team. But they’re also helpful if it’s just you creating a short, simple animation by your lonesome.

- **Test early, test often.** Don’t wait until you’ve attached actions to 16 different objects to begin testing. Test the first action, and then test each additional action as you go along. This iterative approach helps you identify problems while they’re still small and manageable.
- **Test everything.** Instead of assuming the best-case scenario, see what happens when you access your animation over a slow connection or using an older version of Flash Player. What happens when you type the wrong kind of data into an input text field or click buttons in the wrong order? (Believe this: Your audience will do all of these things, and more.)
- **Test blind.** In other words, let someone who’s unfamiliar with how your animation’s supposed to work test it. In programming circles, this type of testing is known as *usability testing*, and it can flush out problems you never dreamed existed.
- **Test in “real world” mode.** Begin your testing in the Flash authoring environment, as you see on page 596, but don’t end there. Always test your animation in a production environment before you go live. For example, if you’re planning to publish your animation to a Web site, upload your files (including your .swf file and any external files your animation depends on) to a Web server, and then test it there, using a computer running the operating system, connection speed, browser, and Flash Player plug-in version you expect your audience to be running. (Sure, transferring a few files isn’t *supposed* to make a difference—but it often does.) Chapter 19 covers publishing to the Web, as well as other publishing options.

Testing on the Stage vs. Testing in Flash Player

Flash gives you two options for testing your animation: on the stage and in the built-in Flash Player. Testing on the stage is faster, and it's good for checking your work as you go along, but in order to try out your animation exactly as your audience will see it, you have to eventually fire it up in Flash Player. Here's some more advice on when to choose each:

Testing on the stage is the quick and easy option, using the Controller toolbar and the associated menu options (Control → Play, Control → Stop, Control → Rewind, Control → Step Forward One Frame, Control → Step Backward One Frame, and Control → Go to End). Testing on the stage is quicker than testing in Flash Player, because you don't have to wait for Flash to compile (*export*) your Flash document, and then load it into the Player. Instead, when you test on the stage, Flash immediately resets the playhead and moves it along the timeline frame by frame. For simple animations, testing on the stage can be easier as well as quicker than testing in Flash Player because you can position the Controller toolbar on your workspace where it's handy—no need to wait for Flash Player's menu options to appear.

The downside to testing on the stage is that it doesn't always test what you think it's testing. For example, if you test

a frame containing a movie clip instance, you don't see the movie clip playing; you have to switch to symbol editing mode, and then test the symbol there to see the movie clip in action. And if your animation contains a button instance and you forget to turn on the checkbox next to Control → Enable Simple Buttons, the button doesn't work on the stage—even though it may work perfectly well when you test it in Flash Player.

Testing in the built-in Flash Player (Control → Test Movie, and Control → Test Scene) is the more accurate option. When you test an animation by selecting Control → Test Movie or Control → Test Scene, Flash generates a .swf file. For example, if you're testing a Flash document named *myDocument.fla*, Flash generates a file called *myDocument.swf* (or *myDocument_myScene.swf*, if you're testing a scene) and automatically loads that .swf file into Flash Player (test window) that's part and parcel of the Flash development environment. This testing option shows you exactly what your audience will see, not counting computer hardware and connection differences (page 602).

Testing on the Stage

If all you want to do is check out a few simple frames' worth of action, this is the option to use. It's also the best choice if you want to see your motion path or *not* see the layers you've marked as hidden. (For the skinny on hiding and showing layers, check out page 153.)

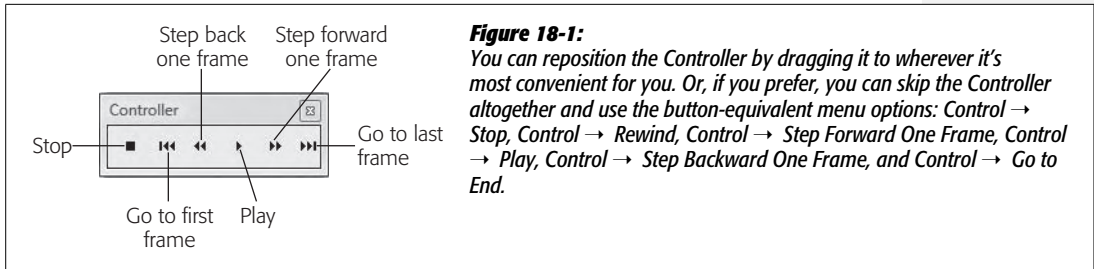
To test your animation on the stage:

1. **Select Window → Toolbars → Controller.**

The Controller toolbar you see in Figure 18-1 appears.

2. **Turn on the checkbox next to one or more of the following options:**

- **Control → Loop Playback.** Tells Flash to loop playback over and over again after you click Play on the Controller. Flash keeps looping your animation until you click Stop. If you don't turn on this option, Flash just plays the animation once.

**Figure 18-1:**

You can reposition the Controller by dragging it to wherever it's most convenient for you. Or, if you prefer, you can skip the Controller altogether and use the button-equivalent menu options: **Control** → **Stop**, **Control** → **Rewind**, **Control** → **Step Forward One Frame**, **Control** → **Play**, **Control** → **Step Backward One Frame**, and **Control** → **Go to End**.

- **Control** → **Play All Scenes**. Tells Flash to play all the scenes in your animation, not just the scene currently visible on the stage.
 - **Control** → **Enable Simple Frame Actions**. Tells Flash to play the actions you've added to frames in the timeline. If you don't turn on this option, Flash ignores all frame actions.
 - **Control** → **Enable Simple Buttons**. Tells Flash to make your buttons work on the stage. (If you don't turn on the checkbox next to this option, mousing over a button or clicking it on the stage has no effect.)
 - **Control** → **Enable Live Preview**. Tells Flash to display any components you've added to the stage the way they'll appear in Flash Player. (The components don't work on the stage, but you see how they're supposed to look.) If you have components on the stage and you don't choose this option, only the outlines of your components appear.
 - **Control** → **Mute Sounds**. Tells Flash not to play any of the sound clips you've added to your animation.
3. **Make sure that what you want to test is at least partially visible in the timeline.**
If you want to test a particular scene, for example, click the Edit Scene icon in the Edit bar, and then choose a scene to display the timeline for that scene. If you want to test a movie clip symbol, select Edit Symbols to display the timeline for that movie clip.
 4. **In the Controller toolbar, click Play to begin testing.**

Your other options include:

- **Stop**. Clicking this square icon stops playback.
- **Go to first frame**. Clicking this icon rewinds your animation. That is, it moves the playhead back to Frame 1.
- **Step back one frame**. Clicking this double-left-arrow icon moves the playhead back one frame. If the playhead is already at Frame 1, this button has no effect.

- **Play.** Clicking this right-arrow toggle button alternately runs your animation on the stage, and pauses it. Playback begins at the playhead. In other words, playback begins with the frame you selected in the timeline and runs either until the end of your animation, or until you press the Stop button.
- **Step forward one frame.** Clicking this double-right-arrow icon moves the playhead forward one frame (unless the playhead is already at the last frame, in which case clicking this icon has no effect).
- **Go to last frame.** Clicking this icon fast-forwards your animation to the very end. That is, it sets the playhead to the last frame in your animation.

Note: You can also drag the playhead back and forth along the timeline to test your animation on the stage (a technique called *scrubbing*).

Flash plays your animation on the stage based on the options you chose in step 2.

Testing in Flash Player

Flash's Test mode shows you a closer approximation of how your animation will actually appear to your audience than testing on the stage. When you fire up the Test Movie command, your animation plays in the Flash Player that comes with Flash CS4. Test mode is your best bet if your animation contains movie clips, buttons, scenes, hidden layers, or actions, since it shows you *all* the parts of your animation—not just the parts currently visible on the stage.

Note: Motion paths (the lines) don't appear when you test your animation in Flash Player, for good reason: Flash designed them to be invisible at runtime. If you want to see your motion paths in action, you need to test your animation on the stage.

To test your animation in Flash Player:

1. **Select Control → Test Movie.**

The Exporting Flash Movie dialog box in Figure 18-2 appears, followed by your animation running in Flash Player (test window) similar to the one in Figure 18-3.

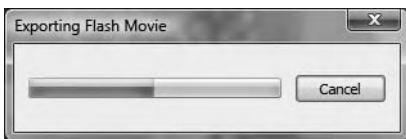


Figure 18-2:

When you see this dialog box, you know Flash is exporting your animation and creating a .swf file. If you've tested this particular animation before, Flash erases the .swf file it previously created and replaces it with the new one. Finishing an export can be fast or slow depending on the size and complexity of your animation and your computer's processing speed and memory.

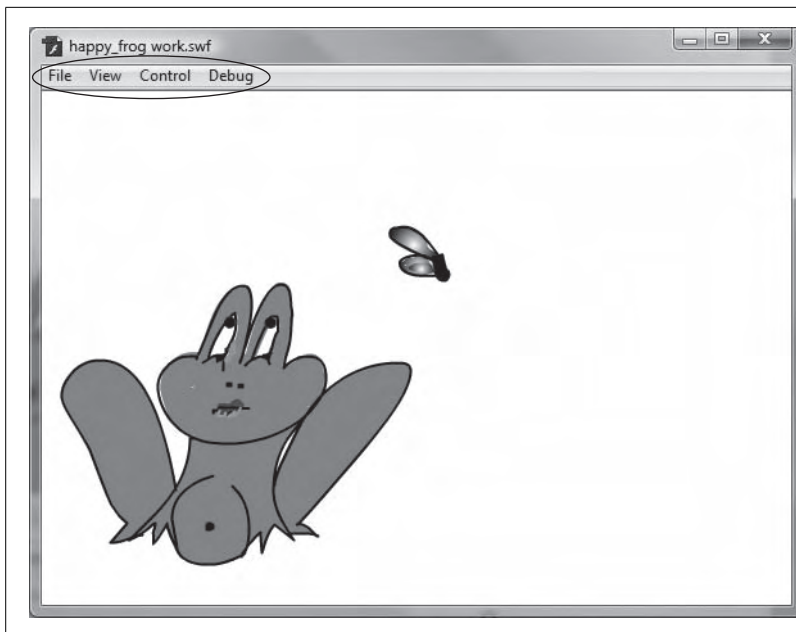


Figure 18-3: Normally, when you select **Control** → **Test Movie** or **Control** → **Test Scene**, Flash opens up Flash Player in its own window. To control playback, you have a couple of choices: You can choose options from the **File**, **View**, **Control**, and **Debug** menus, or you can right-click the window if you're running Windows (Control-click if you're running Mac), and then choose options from the shortcut menus that appear.

2. To control playback—to stop the animation, and then rewind it, for example—choose options from the **Control** menu.

In you're running Windows, the **Control** menu appears in Flash Player; on a Mac, you get the **Control** menu in Flash itself. (In Windows you can also see the **Control** menu in Flash itself if you turn on tabbed viewing in Preferences. See the box on page 600 for details.)

3. To close Flash Player, select **File** → **Close** or click the **X** in the upper-right (Windows) or upper-left (Mac) corner of the window.

Note: Testing your animation in Flash Player gives you a great sense of what your audience will see. But factors like connection speed and hardware differences come into play when you actually publish your animation, so you'll want to test your animation in a real-life production setting (using the same kind of computer, same connection speed, and same version browser as you expect your audience to use) before you go live (see page 595).

Testing Inside a Web Page

In addition to letting you test your animation in Flash Player, Flash lets you test your animation embedded in a Web page. This option lets you see how your animation looks in a Web browser based on the animation alignment, scale, and size options Flash lets you set.

Here's how it works. You tell Flash in the Formats tab of the Publish Settings window (Figure 18-4, left) that you want to embed your animation in a Web page. Then, in the HTML tab, you tell Flash how you want your animation to appear in the Web page (Figure 18-4, right). When you choose File → Publish Preview → HTML, Flash constructs an HTML file containing your animation, and then loads it automatically into the Web browser on your computer.

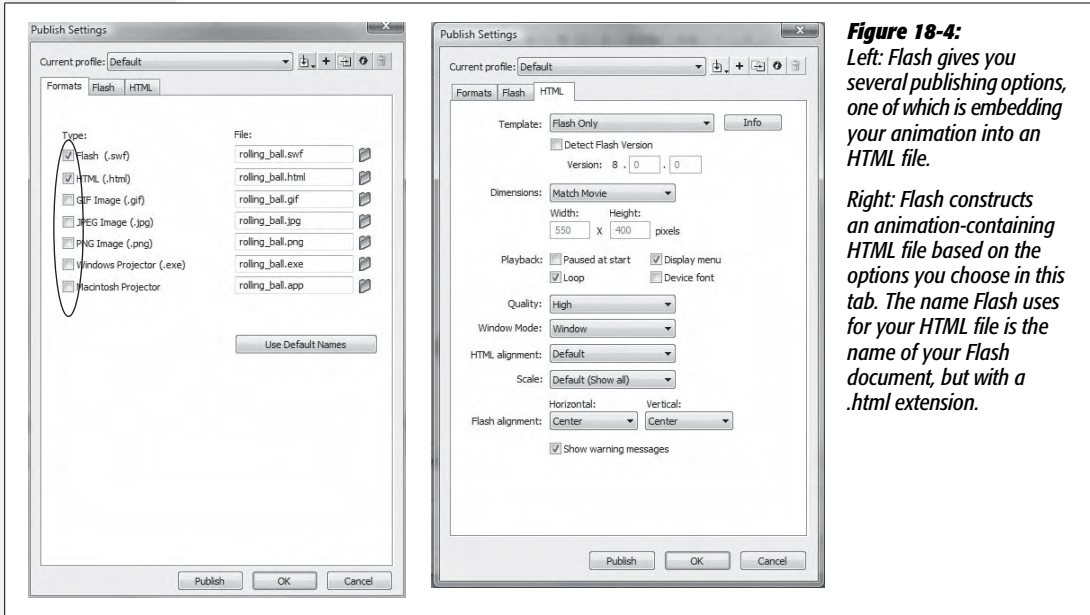


Figure 18-4: Left: Flash gives you several publishing options, one of which is embedding your animation into an HTML file.

Right: Flash constructs an animation-containing HTML file based on the options you choose in this tab. The name Flash uses for your HTML file is the name of your Flash document, but with a .html extension.

Note: Tucking your animation into a Web page is the most popular publishing option, but it's not the only one Flash has. You got acquainted with the other publishing options, including publishing your animation as a QuickTime movie and a standalone Flash projector file, in Chapter 19.

GEM IN THE ROUGH

Testing Multiple Animations

Some folks find a tabbed page—like the tabs in some Web browsers—easier to pop back to, especially if they're trying to test several different animations at once. If you'd rather Flash Player appear in a tabbed page, select Edit → Preferences (Windows) or Flash → Preferences (Mac).

In the Preferences window that appears, click the General category. Turn on the checkbox next to "Open test movie in

tabs." Then choose Control → Test Movie. This time, Flash Player appears as a tab. When you click the tab, the Flash Developer menu options change to the Flash Player options. If you're using Windows, the window with the stage has to be maximized to use tabs.

To test your animation inside a Web page:

1. **Choose File → Publish Settings.**

The Publish Settings dialog box in Figure 18-4 (left) appears.

2. **Make sure the “HTML (.html)” checkbox is turned on, and then click the HTML tab.**

Flash displays the contents of the HTML tab shown in Figure 18-4 (right).

3. **Click the Template drop-down menu, and then choose “Flash only.” Click OK.**

Flash accepts your changes and closes the Publish Settings dialog box.

Note: For a description of each of the settings on this tab, see page 640.

4. **Choose File → Publish Preview → HTML.**

The Publishing dialog box appears briefly to let you know Flash is creating an HTML file. When the dialog box disappears, Flash loads the completed HTML, including your embedded animation, into the Web browser on your computer (Figure 18-5).

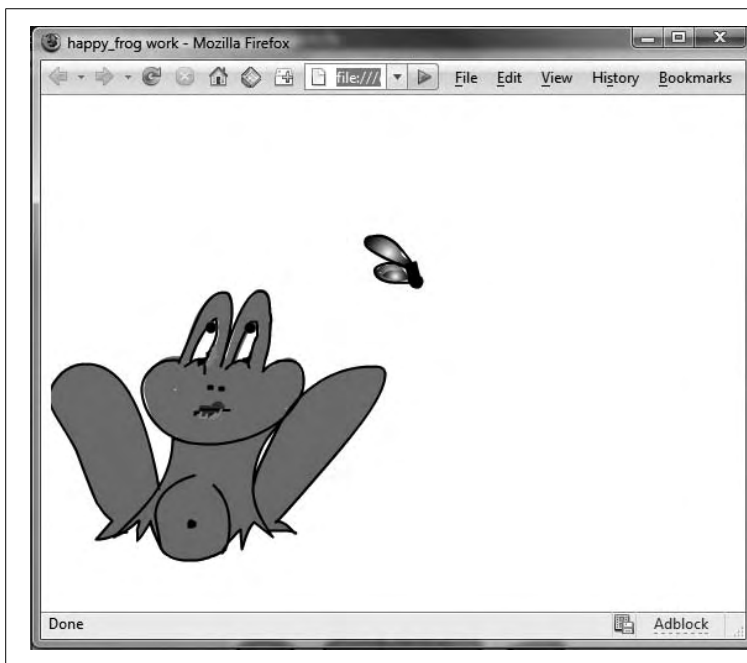


Figure 18-5:
In addition to creating an HTML file, choosing File → Publish Preview → HTML launches your Web browser preloaded with that file.

Right-clicking (Windows) or Control-clicking (Mac) the running animation shows you standard menu options you can use to control playback inside the browser, although how many options you see depends on whether you turned off the checkbox next to “Display menu” in the Publish Settings dialog box.

Testing Download Time

If you're planning to publish your animation on the Web, you need to know how long it takes your animation to download from a Web server to somebody's computer. Chapter 19 gives you several optimization techniques, including tips for pre-loading content and reducing your animation's file size; but before you begin to optimize your animation, you need to know just how bad the situation is and where the bottlenecks are. The following sections show you how.

Simulating Downloads

You *could* set up a bank of test machines, each connected to the Internet at a different transfer speed, to determine the average download time your audience will eventually have to sit through. But Flash gives you an easier option: simulating downloads at a variety of transfer speeds with the click of a button. The simulation takes into consideration any additional, non-Flash media files that you've included in your animation, like sound and video clips.

To simulate different download speeds:

1. Choose **Control** → **Test Movie**.

The Flash Player (test window) appears.

2. Select **View** → **Download Settings** (Figure 18-6), and then, from the submenu, select the connection speed you expect your audience to be running.

Your choices range from 14.4 (1.2 KB/s) to T1 speed (131.2 KB/s). If you need to simulate a faster speed, check out Figure 18-7.

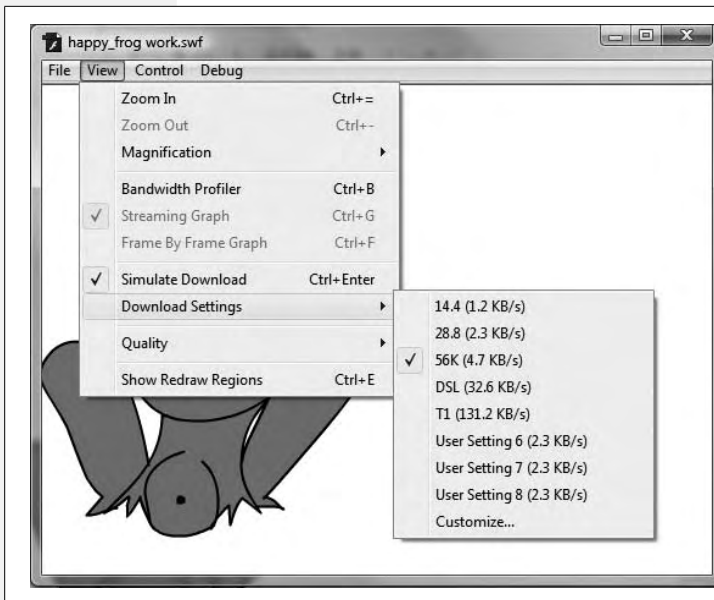
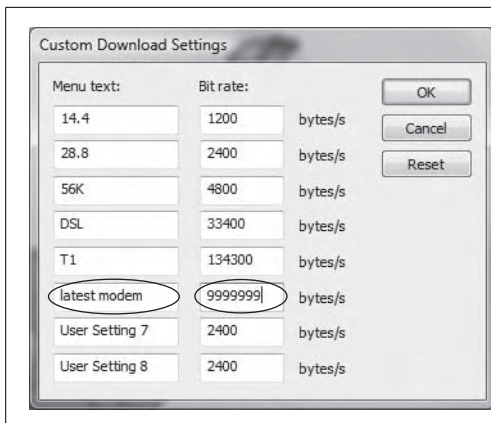


Figure 18-6: If you're used to testing your animation inside the Flash development environment, you'll be shocked when you see how long it takes to download and play your animation over the Web. Flash automatically adjusts for standard line congestion to give you a more realistic picture. So, for example, when you choose the 14.4 kbps setting, Flash actually simulates the transfer at the slightly lower rate of 12.0 kbps.

**Figure 18-7:**

To keep up with the latest advances in transfer technology, you can select a faster transfer rate than any of the options Flash has. To do so, select View → Download Settings → Customize, and then type a label and the new transfer speed you want to test (from 1 byte per second to 10,000,000).

Note: Unless you're planning to let only certain folks to view your animation (for example, students in your company's training classes), you can't possibly know for sure what connection rates your audience will be using. The best approach is to test a likely range. If the animation plays excruciatingly slowly at the lowest connection speed in your test range, consider either optimizing or offering a low-bandwidth version. Chapter 19 (page 624) tells you how.

3. Choose View → Simulate Download.

The test window clears, and Flash plays your animation at the rate it would play it if it had to download your file from a Web server at the connection speed you chose in step 2.

4. Repeat steps 2 and 3 for each connection speed you want to test.

If you're like most folks, you'll find that your animation takes too long to play at one—or even all—of the simulated connection speeds you test. Fortunately, Flash gives you additional tools to help you pinpoint which frames take longest to download (so that you know which frames to optimize). Read on for details.

CODERS' CLINIC

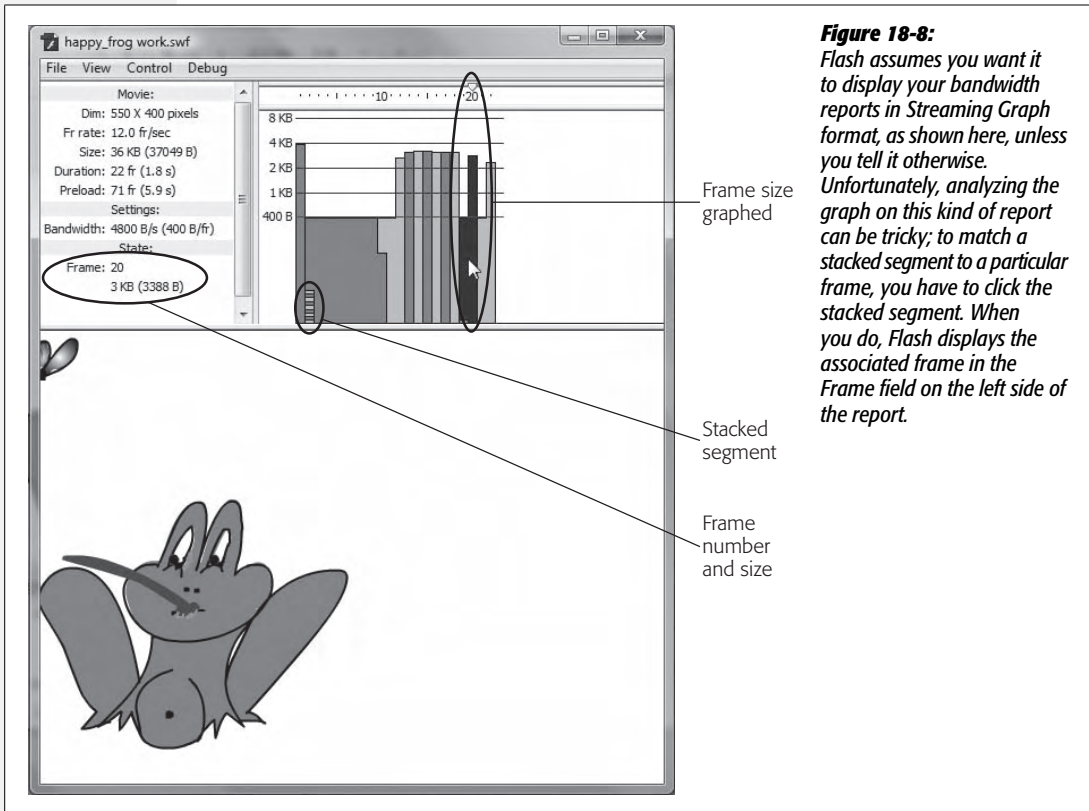
Size Reports

Flash has a second statistical report called a *size report*. To create a size report, choose File → Publish Settings → Flash, and then turn the "Generate size report" checkbox. Make sure you can see the Output window (Window → Output). Then, when you choose File → Publish, Flash displays the size report in the Output window. It also automatically generates a text file named *yourFlashFile Report.txt* that you can pull into a text editor or word processor.

This report provides detailed information about the elements that add to the size of the published .swf file. It thoroughly breaks down the details frame by frame, giving a running total of the file size in bytes. A summary lists scenes, symbols, and fonts and shows how the shapes, text, and ActionScript code adds to the heft of your published file. If you're suffering from file bloat, generate a report to see which images, symbols, sounds, or other elements are causing the problem.

Pinpointing bottlenecks with a bandwidth profiler report

Simulating downloads at different connection speeds gives you a general, overall feel for whether or not you'll need to optimize your animation or give your audience a low-bandwidth alternative (or both). But to get more precise information, like which frames represent the greatest bottlenecks, you need to run a *bandwidth profiler report* (Figure 18-8).



The report gives you information you can use to figure out which frames of your animation are hogging all the bandwidth. There are a timeline and a playhead at the top of the report. As your animation plays, the playhead moves along the timeline to help you see at a glance which frames are causing Flash to display those tall bandwidth-hogging frame bars. *Preload*, the most useful number, tells you how long your audience will have to sit and wait before your animation begins playing. Additional download details in the bandwidth profiler report include:

- **Dimensions(Dim).** The width and height of the stage in pixels (page 41).
- **Frame rate(Fr rate).** The frame rate you set for this animation (page 475).
- **Size.** The size of the .swf file Flash created when you exported (began testing) the movie.

- **Duration.** The number of frames in this animation, followed by the number of seconds the frames take to play based on the frame rate you set.
- **Preload.** The total number of seconds it takes Flash to begin playing the animation at the bandwidth setting you chose (see page 632).
- **Bandwidth.** The connection simulation speed you chose by selecting View → Download Settings.
- **Frame.** The frame Flash is currently loading.

To generate a bandwidth profiler report:

1. **Choose Control → Test Movie.**

The Flash Player (test window) appears containing your running animation.

2. **In the test window, select View → Bandwidth Profiler.**

In the top half of the window, Flash displays a report similar to the one in Figure 18-8.

3. **Select View → Frame By Frame Graph.**

The graph Flash displays when you choose the Frame By Frame option makes detecting rogue frames much easier than if you stick with Flash's suggested View → Streaming Graph option shown in Figure 18-8. Figure 18-9 has an example of a Frame By Frame graph.

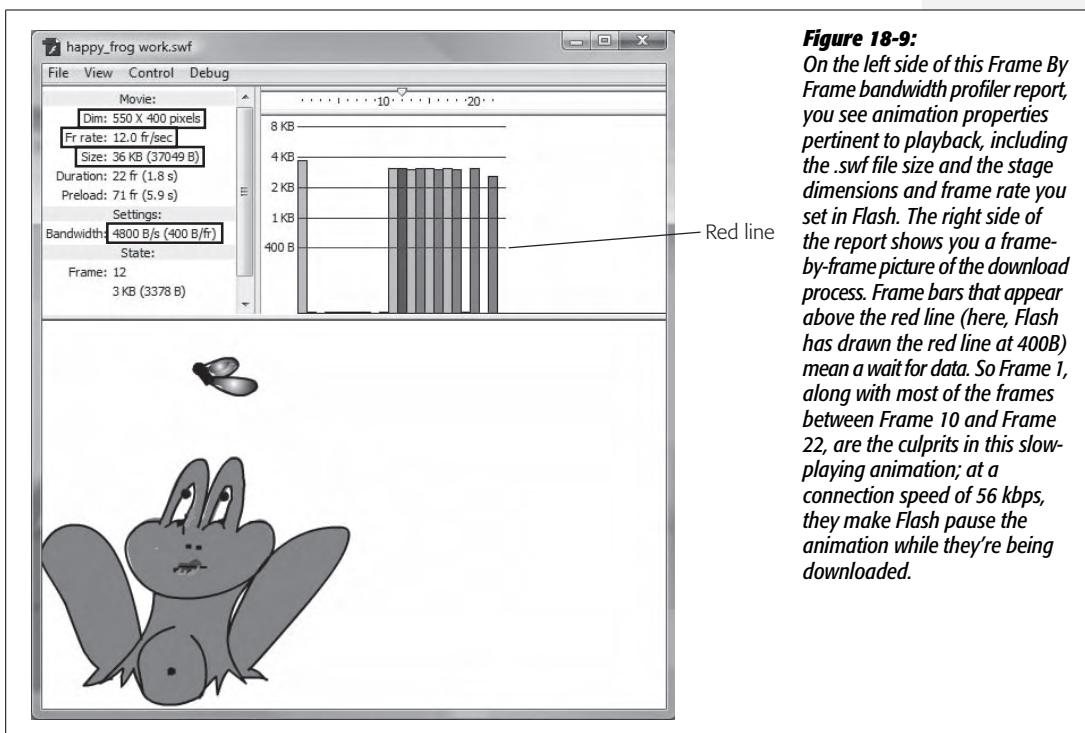


Figure 18-9: On the left side of this Frame By Frame bandwidth profiler report, you see animation properties pertinent to playback, including the .swf file size and the stage dimensions and frame rate you set in Flash. The right side of the report shows you a frame-by-frame picture of the download process. Frame bars that appear above the red line (here, Flash has drawn the red line at 400B) mean a wait for data. So Frame 1, along with most of the frames between Frame 10 and Frame 22, are the culprits in this slow-playing animation; at a connection speed of 56 kbps, they make Flash pause the animation while they're being downloaded.

4. Select View → Simulate Download.

The progress bar at the top of the bandwidth profiler report moves as Flash simulates a download.

If your animation played just fine, try testing it using a slower simulated connection. (Your goal is to make sure as much of your potential audience can enjoy your animation as possible—even folks running over slow connections and congested networks.) To do this test, redisplay the bandwidth profiler report, this time using a different connection simulation speed:

1. Choose View → Download Settings.

A submenu menu appears, showing a list of possible connection simulation speeds, like 28.8, 56K, and T1.

2. Choose the new simulation speed you want to test. Then choose View → Simulate Download again.

A new bandwidth profiler report appears, based on the new connection speed (Figure 18-10).

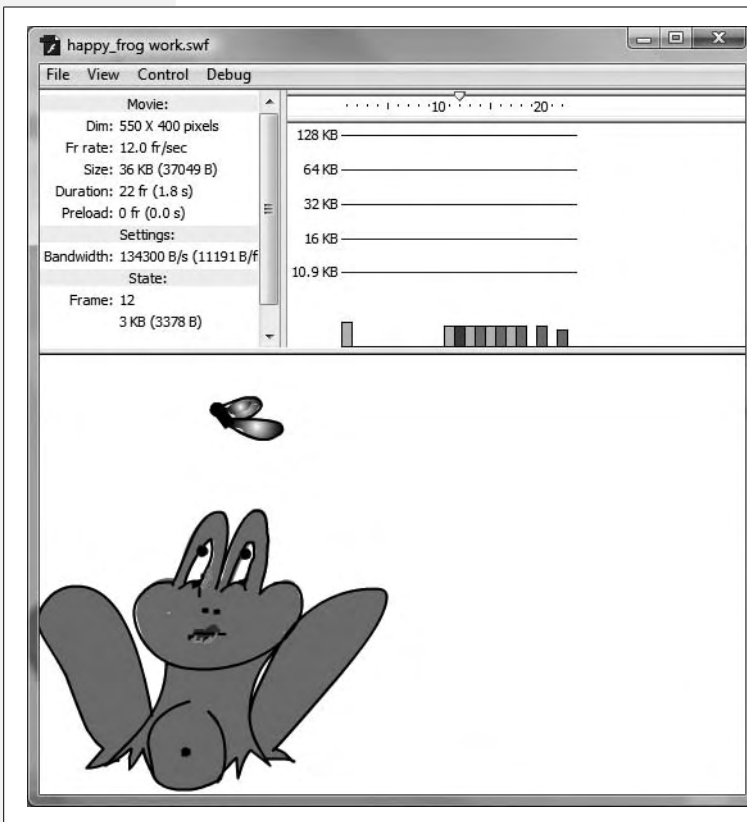


Figure 18-10:

Oh, what a difference a faster connection speed makes! Here, every last one of the frames in the animation appears below the red line that Flash has drawn at 10.9 KB, meaning that audiences running T1 connections don't have to wait one split second for the animation to download and begin playing.

UP TO SPEED

The Flash Player View Menu Options

Flash Player has several menu options that you can use to change the way your animation appears as it's playing. If you turn on the checkbox next to *Display menu* in the Publish Settings → HTML dialog box (coming up in Figure 18-1), your audience can see some of these same options, by right-clicking (Windows) or Control-clicking (Mac).

Note: If you're running a Mac, the following menu options don't appear directly in Flash Player; instead, they appear in the Flash menu.

- **View → Zoom In.** Tells Flash to enlarge your animation. This option's useful if you want to examine your artwork close-up.
- **View → Zoom Out.** Tells Flash to shrink your animation.
- **View → Magnification.** Displays a menu of percentage options you can choose from to tell Flash to enlarge or shrink your animation.
- **View → Bandwidth Profiler.** Creates a bandwidth profiler report.
- **View → Streaming Graph.** Tells Flash to display download data in stacked bars when it creates a bandwidth profiler report (Figure 18-8).
- **View → Frame By Frame Graph.** Tells Flash to display the download time for each frame separately when it creates a bandwidth profiler report (Figure 18-9).
- **View → Simulate Download.** Tells Flash to pretend to download your animation from a Web server based on the download settings you select using View → Download Settings.
- **View → Download Settings.** Displays a list of connection speeds, from 14.4 to T1, to test the download speed of your animation on a variety of different computers.
- **View → Quality.** Tells Flash to display your animation's artwork in one of three different quality modes: low, medium, or high. Flash assumes you want high quality unless you tell it differently. (Choosing low or medium quality doesn't reduce simulation download time, but reducing image quality in the Flash authoring environment does reduce your animation's file size, which in turn speeds up download time.)
- **View → Show Redraw Regions.** Displays borders around the moving images in your animation.

The Art of Debugging

Imagine, for an instant, that your animation isn't behaving the way you think it should. Testing it on the stage or in Flash Player, and then eyeballing the results, as described in the previous section, is a good place to start tracking down the problem. But if you've added ActionScript to your animation, chances are you need more firepower. You need to be able to examine the inner workings of your ActionScript code—the variables, instance names, methods, and so on—to help you figure out what's wrong. Debugging is one of those activities that's part art, part craft, and part science. Flash and ActionScript provide several tools that help you track down and eliminate those pesky bugs of all types. The tools at your disposal include:

- The **Check syntax** button catches the most obvious typos. If you've got too many parentheses in a line or you misplaced a comma or semi-colon, the syntax checker is likely to notice. Still, it lets lots of the bad guys through.

- The **Compiler Errors** panel is the next layer of defense against bugs. If there's a flaw in your code's logic (for example, a reference to some object that doesn't exist or is misnamed), a message is likely to appear in the Compiler Errors panel. Sometimes your animation will run anyway; other times it won't.
- The **Output** panel displays messages. Using the `trace()` statement, you can display the values of variable and object properties in the Output panel. So, you get to tell ActionScript what to report on.
- The **Debugger** is your debugging power tool. It's kind of like the diagnostic machine your mechanic connects to your car to see what's going on inside. The debugger combines the usefulness of the other tools with the all-important ability to stop your animation and code in its tracks. That gives you an opportunity to examine the critical variable and object values.

Note: The debugger for ActionScript 3.0 is different in a few ways from the debugger for ActionScript 2.0 and 1.0 code. For details on the older debugger, see *Flash CS3: The Missing Manual*.

You're likely to use the first three tools as you're writing and testing your animation. As your code gets more complex, with multiple timelines, multiple objects, and multiple functions and methods, you'll turn to the debugger. This section starts off with the quick and easy bug squashers, and then moves on to the more complex. The troublesome program attached to that diagnostic machine is called *draw_random_lines_begin fla* and you can find download it from the "Missing CD" page at <http://missingmanuals.com/cds>. If you'd like to see how the program is supposed to behave, check out *draw_random_lines_finished fla*.

UP TO SPEED

Deciphering the Actions Panel's Color Code

One quick way to spot problems in your ActionScript code is to examine the colors Flash uses to display your code in the Actions panel.

Right out of the box, Flash displays ActionScript keywords in blue, comments in light gray, text strings (text surrounded by quotes) in green, and stuff it doesn't recognize in black. So if you notice a function call or a property that appears black, you know there's a problem. A properly spelled function call or property should appear blue (unless it's a custom function), so if it's black, chances are your finger slipped.

If Flash's ActionScript coloring scheme is too subtle for your tastes, you can change the colors it uses.

To change colors:

1. Select Edit → Preferences (Windows) or Flash → Preferences (Mac).
2. In the Preferences panel that appears, select the ActionScript category. Make sure the checkbox next to "Code coloring" is turned on.
3. Click the color pickers next to Foreground, Keywords, Identifiers, Background, Comments, and Strings to choose different colors for each of these ActionScript code elements.

For example, if you have trouble making out the text strings in your scripts due to red-green color-blindness, you can change Strings to a different hue.

Using the Syntax Checker

The animation *draw_random_lines_begin.fla* isn't behaving the way it should—pretty ornery for a snippet of code that's only 16 lines long. It's supposed to draw lines on the screen from one random point to another. The lines are supposed to randomly vary in thickness, color, and transparency. There are four text fields in the display that are supposed to flash the X/Y coordinates of the points used to draw the lines. If you try to test the program Ctrl+Enter (⌘-Return on a Mac), a little text appears on the screen, but nothing much happens. The first step to putting things on track is to use the “Check syntax” button in the Actions panel, as shown in Figure 18-11.



Figure 18-11:
The Check syntax button looks like a check. One click, and Flash proofreads your code to find typo and punctuation errors.

1. With *draw_random_lines_begin.fla* open in Flash, select Window → Actions.

The Actions panel opens. The Actions panel can look different depending on how you've set up your workspace. It also remembers some of the settings, like which panels are open and closed, from the last time you used it.

2. If you don't see code in the Actions panel, in the animation's timeline, click Frame 1 in the “actions” layer.

The Actions panel shows the code associated with particular frames in the timeline. It's not a problem with this little snippet, but with larger animations, if you don't see the actions you want to debug, make sure you've selected frame that holds the code.

3. In the Actions panel, click the “Check syntax” button.

The “Check syntax” button looks like a checkmark. After a little deep thinking, ActionScript sends you a message like the one in Figure 18-12. If there's an error in your code, you're referred to the Compiler Errors panel.

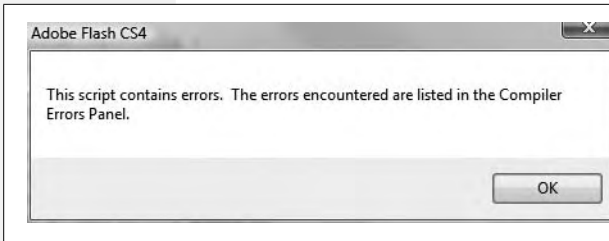


Figure 18-12:
When the Check syntax button uncovers a typo in your code, it sends you this message, which refers you to another message in the Compiler Errors panel.

4. In the Compiler Errors panel, double-click the error message.

If the Compiler Errors panel was closed or hidden, it opens when “Check syntax” finds an error. The Compiler Errors panel’s location may vary depending on how you’ve organized your workspace. If you’re using the Essentials workspace, it appears beneath your animation. (And if you’re wondering what the heck a compiler is, see the box on page 611.)

The message in the Compiler Errors panel looks like Figure 18-13. As helpful as these details are, sometimes your view of the issue and the compiler’s view aren’t coming from the same direction, so the messages may seem a bit cryptic. In this case, you’re told the error is on Line 7. The error’s description is:

1086: Syntax error: expecting semicolon before dot.

That’s a little on the cryptic side, but it means there’s probably something wrong with the way the line is punctuated. Double-clicking the error message puts your cursor in the offending line in the Actions panel.

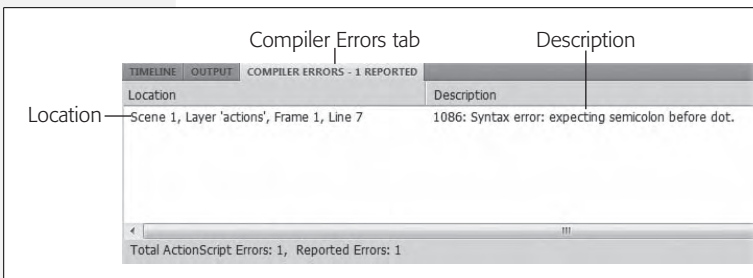


Figure 18-13:
The messages in the Compiler Errors panel have two parts. The Location part tells you where the error is and the Description part tells you how the compiler sees the error.

5. Examine the highlighted line for a syntax error.

Here’s where you get into the grunt work of debugging, looking through your code to find out where there might be a problem. Flash is a lot fussier than your third grade teacher about punctuation, so double-check to make sure there are commas between parameters and a semicolon at the end of the line. Parentheses are another place where it’s easy to make a mistake. In any statement (from the beginning of the line to the semicolon), there should be an equal number of left parentheses and right parentheses. And yes, they all need to be in the right

spots, too. But it's so easy to mess up on parentheses that counting is a legitimate debugging technique. In the code in line 7, there's one extra closing parenthesis before the comma.

```
var ptStart:Point = new Point(Math.random() * 550),Math.random() * 400);
```

6. Delete the error, and then click “Check syntax” again.

This time, a little box appears that happily says, “This script contains no syntax errors.” That's fine as far as it goes. With ActionScript 3.0 code, all the syntax checker does is check the punctuation and a few other details of your code. The Syntax checker works quickly because it doesn't actually compile your code. That means it doesn't catch nearly as many errors as you find when you test your animation using Ctrl+Enter (⌘-Return on a Mac).

UP TO SPEED

What's a Compiler and Why Does It Err?

When you write ActionScript code, you name objects and write statements in a language that's relatively understandable by humans. Your computer, however, speaks a different language altogether. When Flash compiles your ActionScript code, it translates the code from your human language to the computer's machine language. When Flash comes across statements that don't make sense, it says “Aha! A compiler error!” For example, one very common compiler error is the simple misspelling of an object's name.

Flash and ActionScript are very literal. If there's a misplaced letter or even an error in capitalization in a word, that's an error. For example, if your program has a variable named *myBall* and you mistakenly type in *myball*, ActionScript sees that second reference as an undefined object and a compiler error. Anything that prevents the compiler from successfully identifying all the objects and values and performing all the methods in your program results in an error.

Using the Compiler Errors Panel

When you test your animation using Control → Test Movie or by pressing Ctrl+Enter (⌘-Return on a Mac), Flash creates a .swf file. That's the same as the finished file you distribute or put on a Web site so the world can see your animation. In the process, your ActionScript code is translated into a computer language that's smaller and faster than your ActionScript. The process of compiling your code is likely to catch mistakes that the “Check syntax” button misses. (See the box above for more details.)

Note: This example continues debugging the file *draw_random_lines_begin fla*. The entire process began on page 608.

1. Test your animation using Ctrl+Enter (⌘-Return on a Mac).

An error appears in the Compiler Errors panel, as shown in Figure 18-14. (It could be worse; sometimes you see seven or eight errors stacked up in the panel.) This error didn't appear when you clicked the “Check syntax” button in the previous exercise, because “Check syntax” doesn't compile the code. Obviously, the

compiler choked on something you're trying to feed it. The Compiler Errors panel reports that the location of the error is "Scene 1, Layer 'actions', Frame 1, Line 4." That's very helpful information, and what's more, when you double-click the error message in the panel, Flash zips you to the Actions panel and finds that point in your code. But before you double-click, read the description of the error:

1120: Access of undefined property sptLines.

That's also a good clue; explaining the problem, at least as far as the compiler sees it. The compiler thinks you're trying to make a change to a property that doesn't exist. That property is called sptLines.

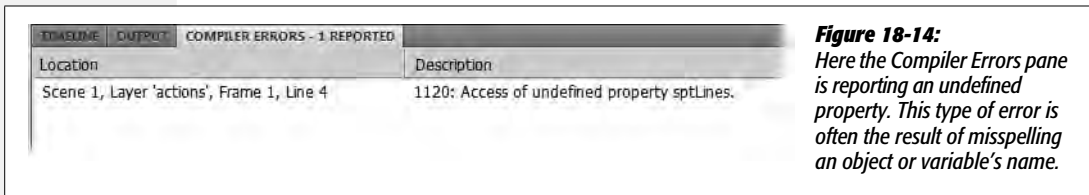


Figure 18-14: Here the Compiler Errors pane is reporting an undefined property. This type of error is often the result of misspelling an object or variable's name.

2. **Double-click the error message in the Compiler Errors panel.**

Flash zips you to the Actions panel and highlights line 4, which reads:

```
addChild(sptLines);
```

3. **Examine the highlighted line for a syntax error.**

At first this might seem a little puzzling. It's a simple statement that adds sptLines to the display list which should make its contents visible on the stage. The compiler error said something about a property, but this line doesn't seem to be changing a property. You know, however, that the compiler sees something it can't identify, and that's *sptLines*. It makes sense to check the code that precedes the error for previous references to *sptLines*. There aren't any, and that's exactly the problem. The variable *sprtLines* is defined in the first line of the code, and the reference to *sptLines* is a typo.

4. **In line 4, correct the spelling of sprtLines, and then test the animation.**

This time, *draw_random_lines_begin fla* is a little more entertaining. The animation draws random lines on the stage. The lines vary in thickness, color, and transparency. But the text boxes, like the one that reads *start X*, don't change or provide any other information. Sounds like there's more debugging to do.

Note: The animation *draw_random_lines_begin fla* uses a very slow frame rate of 2 fps so that each line appears slowly on the screen. If you want to speed up the action, go to Modify → Document and change the frame rate.

Note: Stop the *draw_random_lines_begin.fla* when you're sufficiently entertained. Because there's no automatic end to the animation, Flash can conceivably draw so many lines that your computer will run out of memory trying to display them.

Using the Output Panel and *trace()* Statement

The ActionScript *trace()* statement is one of the easiest ways to debug your programs—and it delivers a lot of bang for your debugging buck. You get to tell ActionScript exactly what variable value or object you want to keep track of, and Flash obligingly sends the details to the Output panel. *Trace()* is such an important code writing tool that it's used throughout the ActionScript examples in this book. Here's another good example of the way you can use *trace()* to understand why your program isn't behaving as expected.

Note: This example continues debugging the file *draw_random_lines_begin.fla*. The entire process begins on page 608.

When you tested *draw_random_lines_begin.fla* in the last step on page 612, the drawing lines part of the program worked, but the text fields didn't display information about the points used to start and end the lines. Text fields display strings of text in your animation. There are a few different types of text fields and you can format them in a number of ways. (For all the details, see page 538.) In this case, the text fields show some of the information that you want displayed, but not all of it.

The *trace()* statement works kind of like a text field. You put the information that you want displayed inside of the *trace()* statement's parentheses. If you want to display a string of text, put it inside of quotes, like this:

```
trace("show this text");
```

When your ActionScript code gets to that line, the words “show this text” appear in the Output panel (without the quotes). If you have a variable named *strMsg*, and its value is “show this text”, then you can write a *trace()* statement like this:

```
trace(strMsg);
```

This statement would also send the words “show this text” to the Output panel. The Output panel isn't at all fussy about the data types. Put the name of a variable, an instance of an object, or just about anything inside of a *trace()* statement, and something is bound to appear in the Output panel. If the value is a number, that's what you see. If it's a reference to an object, you'll see the object's name.

Following are some steps to gain a little insight into the problem with the text fields in the file *draw_random_lines_begin.fla*.

1. Test the animation as it worked at the end of the previous section.

When you test the animation, you see the random lines drawn properly, but you see only part of the text that should be displayed in text fields, as shown in Figure 18-15. Don't forget to stop the animation (Control → Stop) or close the window (File → Close).

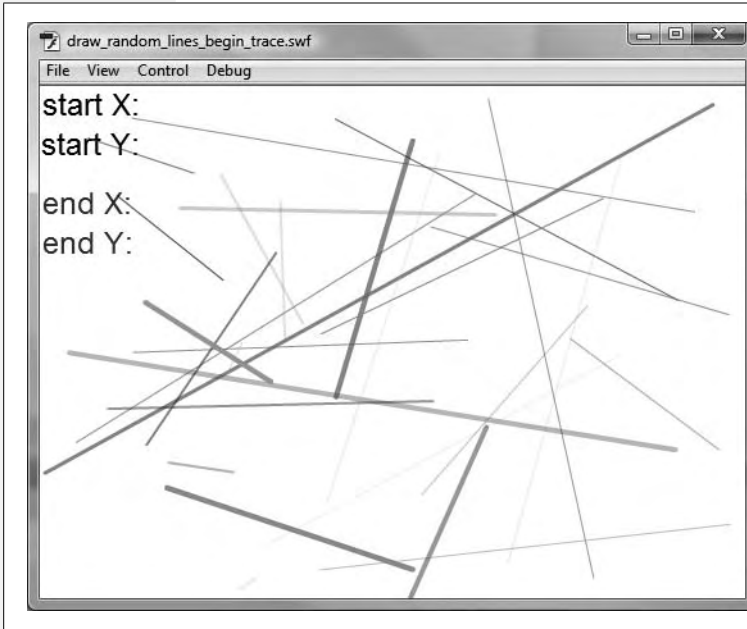


Figure 18-15:
When you test the animation draw_random_lines_begin.fla, it displays lines on the stage, but it doesn't display the point coordinates for the lines as it should.

2. On the stage, click the text fields, and then, in the Properties panel, check the names of the text fields.

In your Flash document, the text fields on the stage show text, like “start X” and “start Y.” When you select a text field, you see its name at the top of the Properties panel. For example, the text field with the text “start X” is named txtStartX. The others are named txtStartY, txtEndX, and txtEndY. These are the names of misbehaving text fields, so you'll look for references to them in your code.

3. Select Window → Actions to open the Actions panel.

The Actions panel displays this code, which amounts to only 16 lines:

```

1  var sprtLines:Sprite = new Sprite();
2
3  drawRandomLine();
4  addChild(sprtLines);
5
6  function drawRandomLine():void {
7      var ptStart:Point = new Point(Math.random() * 550,Math.random() * 400);

```

```

8     var ptEnd:Point = new Point(Math.random() * 550,Math.random() * 400);
9     sprtLines.graphics.lineStyle(Math.random()*5,Math.random()*0xFFFFFF,
    Math.random());
10    sprtLines.graphics.moveTo(ptStart.x,ptStart.y);
11    sprtLines.graphics.lineTo(ptEnd.x,ptEnd.y);
12    txtStartX.text = "start X: " + ptStart.x;
13    txtStartY.text = "start Y: " + ptStart.y;
14    txtEndX.text = "end X: " + ptEnd.x;
15    txtEndY.text = "end Y: " + ptEnd.y;
16 }

```

4. Search for lines with references to the misbehaving text fields: `txtStartX`, `txtStartY`, `txtEndX`, and `txtEndY`.

In lines 12 through 15, values are assigned to the text fields' properties. Each value is made of two parts, a string literal with text like "start X:", and then the string concatenation operator (+) and a reference to an object's property, like `ptStart.x`. Looking back up in the code, you see on line 7 that the data type for `ptStart` is `Point`. The reference `ptStart.x` is a reference to the X property of a point. That value is a number. Still, there's nothing apparently wrong with the code.

5. Insert a line at line 13, and then type the following `trace()` statement:

```
trace("start X: " + ptStart.x);
```

The text inside the parentheses is exactly the text that's supposed to appear in the text field. In fact, you can copy and paste to create the line. Using copy and paste is a good technique for an operation like this, because you'll be sure the text in the two statements is identical.

Tip: If you don't see your `trace()` statement in the Output panel, select File → Publish Settings → Flash, and make sure the "Omit trace actions" checkbox is turned off.

6. Test your animation using `Ctrl+Enter` (⌘-Return on a Mac) and examine the Output panel.

When you run your animation, the Output window starts to fill up with lines like:

```

start X: 549.6419722447172
start X: 499.13692246191204
start X: 239.57312640268356
start X: 64.5334855420515

```

Comparing the Output panel details, you see that your text fields display the string literals, like `start X`, but they aren't displaying the numbers with all those

decimal places. Those long numbers are generated by the `random()` method used earlier in the code. For example, the value for `ptStart.x` is created in line 7 with the statement:

```
var ptStart:Point = new Point(Math.random() * 550,Math.random() * 400);
```

This line creates a new variable called `ptStart`. Its data type is `Point`, which includes `x` and `y` properties. At the same time that the new instance of `Point` is being created, values are assigned to those `x` and `y` properties. Instead of providing specific numbers, you want to provide random numbers that change every time the `drawRandomLine()` method runs. So the statement uses a method that's provided by the `Math` class. The section of the statement that reads `Math.random() * 550` is in effect saying, give me a random number between zero and 550 (the width of the stage). Likewise, the next bit of code is providing a number for the `y` property that matches the height of the stage. `Math.random()` is providing a number with a little more precision than is necessary for this snippet of a program. In fact, the number is too long to fit in the text field.

7. Select the `txtStartX` text field on the stage and change the width property to 550.

The width of the text field expands so it's the length of the stage.

8. Test your animation using `Ctrl+Enter` (`⌘-Return` on a Mac).

When the animation runs, the entire number is displayed in the `txtStartX` text field, as shown in Figure 18-16. The `x` and `y` properties aren't displayed in the other text fields.

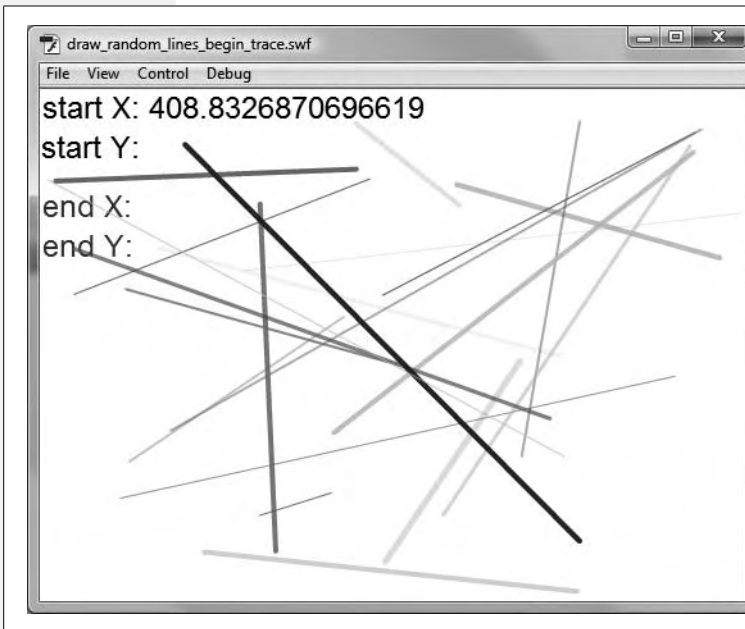


Figure 18-16:
After changing the width of the text field, Flash displays the very long number that represents the `x` property for your line's starting point.

Mystery solved. You figured out what's gone wrong with the code. The solution is a little less than satisfactory—you don't really need a number that long for this animation. Fortunately, that gives you an opportunity to explore the Debugger in action, as you'll see in the next section.

You can delete the `trace()` statement from your code or you can leave it in. It doesn't change the appearance of the animation in any way. A third alternative is to use a handy programmer's trick called *commenting out*; see the box below.

TRICK OF THE TRADE

Comment Me Out

If you're planning to keep working on your ActionScript code and think you'll need to reuse these `trace()` statements at some point down the road, you don't have to delete them, and then type them in again later. Instead, you can "comment them out" by placing two slashes in front of each line, like this:

```
// trace("start X: " + ptStart.x);
```

When you stick two slashes at the beginning of a line of ActionScript code, Flash ignores everything it finds on the line

after those slashes. In other words, it treats the code as if it were a plain old comment. Later, when you want to use that `trace()` statement again, all you have to do is remove the slashes and you're back in business.

It's a good idea to remove or comment out `trace()` statements when you no longer need them. Not only will you avoid cluttering up your Output panel, there's a performance boost, too.

Using the Debugger

When you need as much debugging muscle as Flash can provide, click Debug → Debug Movie. You may think you've fired up a different program, but actually, Flash has merely closed some panels, opened others, and rearranged your view of your animation (Figure 18-18). It also automatically compiles and runs your animation. Your first visit to the debugger can be a little intimidating, but don't worry. Look around for familiar landmarks, and you soon figure out the purposes of the multiple panels and the messages within.

- The **Debug Console** in the upper-left corner shows DVD-like Play and Stop buttons (Figure 18-17) and that's exactly what you do with the buttons in the debug console. You use them to move forward and backward through your code. You can also open the Debug Console using menus: Window → Debug Panels → Debug Console.
- The **Variables** panel below the Debug Console is where you really learn what's going on in your program. You see variable and object names on the left and their related values on the right. There are probably a lot of unfamiliar words in there, because this panel keeps track of every property for every object in your animation. You don't need to worry about many of these, because Flash takes care of them perfectly well. But when something goes wrong, look up the name

of the offending text box, variable name, or object in this list, and you'll be on your way to a solution. You can also open the Variables panel using menus: Window → Debug Panels → Variables.

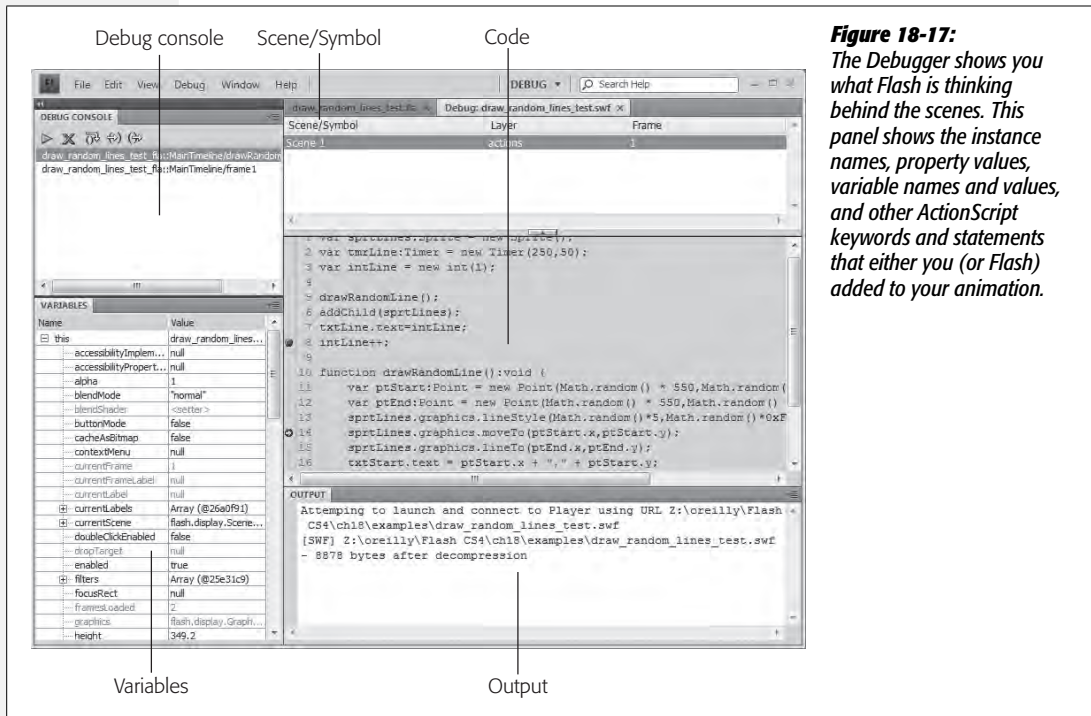


Figure 18-17: The Debugger shows you what Flash is thinking behind the scenes. This panel shows the instance names, property values, variable names and values, and other ActionScript keywords and statements that either you (or Flash) added to your animation.

- In the upper-right corner, the **Scene/Symbol** panel is pretty straightforward. It tracks your animation's current position in the main timeline, scene, or symbol timeline. You see the name of the scene or symbol, the layer that contains code, and the frame number.
- The middle panel shows the **ActionScript** code you wrote, similar to the Actions panel. You can force your program to stop at certain places in your code to give you a chance to inspect the inner workings of the objects. More on that in the next section: *Setting and Using Breakpoints*.
- At the bottom is the **Output** panel, covered earlier in this chapter (page 613). If you've used the `trace()` statement while you were writing code and experimenting with ActionScript, you know how helpful the Output panel can be.

Setting and Using Breakpoints

One of the most important debugging tools in any well-stocked ActionScript programmer's arsenal is the *breakpoint*. A breakpoint is an artificial stopping point—sort of a roadblock—that you can insert into your ActionScript code to stop Flash Player in its tracks. Setting breakpoints lets you examine your animation at different points during playback so that you can pinpoint where a bug first happens.

Flash lets you set breakpoints at specific lines in your ActionScript code. Setting a breakpoint lets you play the animation only up until Flash encounters that breakpoint. The instant Flash encounters a line with a breakpoint, it immediately stops the animation so that you can either examine object property values (as described in the previous section) or step through the remaining code in your action slowly, line by line, watching what happens as you go.

Setting breakpoints is a great way to track down logic errors in your ActionScript code. For example, say you've created a chunk of code containing a lot of *if* and *switch* conditionals or *while* and *for* looping statements. Stopping playback just before you enter that long stretch of code lets you follow Flash as it works through the statements one at a time. By stepping through statements in the order Flash actually executes them (as opposed to the order you thought Flash was supposed to execute them), you may find, for example, the cause of your problem is that Flash never reaches the *else* section of your *if...else* statement, or never performs any of the statements inside your *while* block because the *while* condition is never met.

Note: For more information on using *if...else*, *do...while*, and other logical statements in ActionScript, check out Colin Mook's *Essential ActionScript 2.0* (O'Reilly) or *Essential ActionScript 3.0* (O'Reilly). Both books have detailed coverage of more advanced ActionScript topics that are beyond the scope of this book.

So far, this chapter has shown how to clean up the buggy code in the file *draw_random_lines_begin fla*. In this section, you can make further improvements to the animation while learning how to stop your animation and code in its tracks and examine individual properties using the debugger.

Note: This example continues debugging the file *draw_random_lines_begin fla*. The entire process began on page 608.

To get started, follow these steps:

1. With *draw_random_lines_begin fla* open in Flash, click **Debug** → **Debug Movie**.

A Flash Player window opens and begins playing your animation. The panels in Flash change to show the Debug Console, the Variables panel, the Scene/Symbol panel, your code, and the Output panel. There's no information showing in either the Debug Console or the Variables panel at this point. If you followed the steps in the previous example, some information appears in the Output panel.

2. In the Flash Player window, select the toggle **Ctrl+Enter** (⌘-P on a Mac).

The animation stops playing.

3. In the panel with the ActionScript code, click to the left of the line numbers 7 and 8.

A red dot appears next to the line number, indicating a breakpoint.

4. In the Flash Player window, press Ctrl+Enter (⌘-P on a Mac).

The Flash Player may be hidden by the debugger. If necessary, use Alt+Tab (or ⌘-Tab on a Mac) to bring it to the front. When you press Ctrl+Enter, the animation runs for a moment, and then stops. A small arrow appears in the breakpoint next to line 7, indicating that the animation is at this point in the ActionScript code. You see more details in both the Debug Console (Figure 18-18) and the Variables panel.

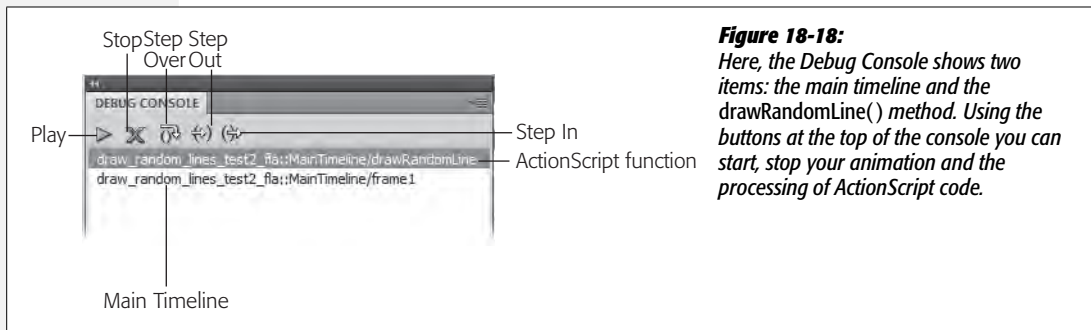


Figure 18-18:

Here, the Debug Console shows two items: the main timeline and the `drawRandomLine()` method. Using the buttons at the top of the console you can start, stop your animation and the processing of ActionScript code.

There are two items displayed in the Debug Console. You may need to drag the right edge of the panel to read the entire lines. The line at the top references the `drawRandomLine()` method in your code. The bottom line references the main timeline in the animation. When you click different items in the Debug Console, the items listed in the Variables panel change. Click the top line before examining the Variables panel in the next step.

5. In the Variables panel, click the plus sign next to the word “this”, and then scroll the panel to view all the variables.

Initially, there are three items in the variables panel: *this*, *ptStart*, and *ptEnd*. *This* refers to the main timeline; *ptStart* and *ptEnd* are variables inside of the `drawRandomLine()` function. At this point, the value for both variables is undefined. When you click the + button next to *this*, a list expands beneath showing all the properties and variables related to the main timeline. Some of the items are familiar, like *alpha*, *height*, and *width*. Others may be a bit mysterious, especially if they’re properties you haven’t yet used in ActionScript: in Flash, there are a lot of preset values, which you may never need to worry about. If you look carefully in the list, you find the names of the text fields in your animation: *txtStartX*, *txtStartY*, *txtEndX*, and *txtEndY*.

6. Click a second time to close the list.

The lists closes, leaving just the three items showing: *this*, *ptStart*, and *ptEnd*.

7. Click the green Continue button in the Debug Console.

Flash moves ahead just one step in the code because there’s a breakpoint at line 8. In the Variables panel, you see that the *ptStart* item has changed. The value is

no longer undefined; there's some weird-looking number there. And there's a + button next to the name.

When you place a breakpoint in your code, the debugger stops before that line is executed. That's why, with a breakpoint at line 8, `ptStart` has newly assigned values, but `ptEnd` is still undefined.

8. Click the + button next to `ptStart` and examine its properties.

As explained on page 570, the `Point` class has three properties: `length`, `x`, and `y`. Now that `ptStart` is defined, the variables panel shows values for each. More of those are really long decimal numbers.

9. Double-click the `x` value for `ptStart` and type `300`. Then, double-click the `y` value and type `200`.

One of the extremely handy features of the Debugger is that you can change values of properties when the animation is stopped at a breakpoint.

10. Click the green Continue button.

The animation movies through its two frames, and then it runs the `drawRandomLine()` method again. It stops again at line 7 in the code.

11. Examine the animation in the Flash Player window.

The two text fields at the top of the animation show the values you entered in the Variables panel: `300` and `200` as shown in Figure 18-19. This small test proves that if you round the numbers to whole numbers, they'll fit in the text fields.

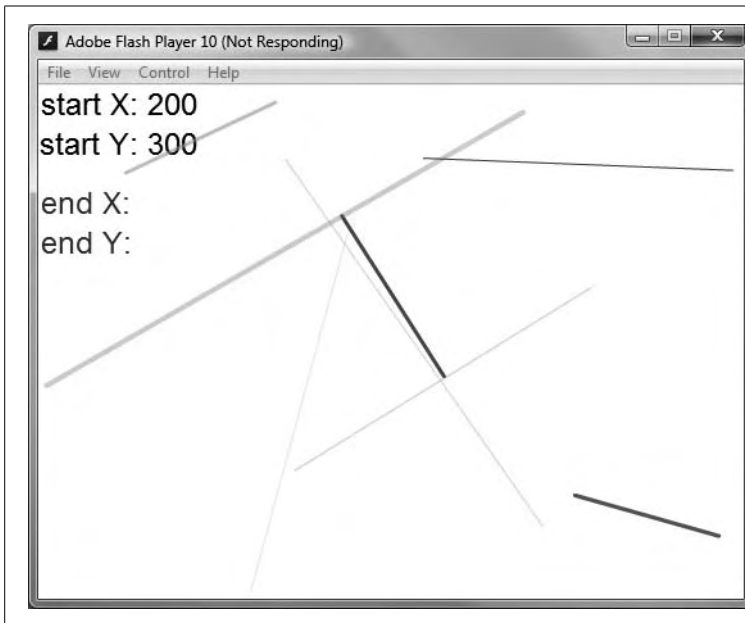


Figure 18-19:

When you change values in the Debugger, you can see the results in your animation. Here, the values for two of the text fields have been changed. Before the change, the numbers were too long to be displayed by the text fields.

12. In the Debug Console, click the red X, also known as the End Debug Session button.

Flash restores your project to its appearance before you entered the debugger.

So, one solution to making the numbers fit in the text fields is to round them off to whole numbers. You can do that with another method that's part of the Math class. Here's what the finished code looks like. The two bold lines show the changes using the *Math.round()* method.

```
var sprtLines:Sprite = new Sprite();

drawRandomLine();
addChild(sprtLines);

function drawRandomLine():void {
    var ptStart:Point = new Point(Math.round(Math.random() * 550),Math.
round(Math.random() * 400));
    var ptEnd:Point = new Point(Math.round(Math.random() * 550),Math.
round(Math.random() * 400));
    sprtLines.graphics.lineStyle(Math.random()*5,Math.random()*0xFFFFF,
Math.random());
    sprtLines.graphics.moveTo(ptStart.x,ptStart.y);
    sprtLines.graphics.lineTo(ptEnd.x,ptEnd.y);
    txtStartX.text = "start X: " + ptStart.x;
    trace("start X: " + ptStart.x);
    txtStartY.text = "start Y: " + ptStart.y;
    txtEndX.text = "end X: " + ptEnd.x;
    txtEndY.text = "end Y: " + ptEnd.y;
}
```

Now the animation runs as it was intended. It draws lines that are random in position, color, thickness, and transparency. The X/Y coordinates for the start and end of the lines is shown in upper-left corner of the animation. Not only do the whole numbers fit in the text fields, they're a little easier to discern than the monster decimals.

Note: Flash lets you debug your animations remotely after you've uploaded them to a Web server. This book doesn't cover remote debugging, but you can find out more about it in Flash's help files.
