

# SQUASH 'EM IF YOU'VE GOT 'EM: THE BUG HUNT

## CHAPTER OUTLINE

### Bugs 1

Traces 2

Flash Tracer 4

The Debugger 4

### Performance and Optimization 7

The FrameRateProfiler Class 7

The MemoryProfiler Class 11

The Sample Package 14

### Summary 18

As you get close to completing a project, whether on your own or with a team, it will usually (and should *always*) go through some form of quality assurance (QA). During this process, those testing the game should look for bugs, performance issues (both CPU and network related), and general playability. Playtesting a game to refine a mechanic and make it more fun is an entire process unto itself, which we will not cover in this chapter. We *will* cover ways to speed up the process of debugging code and tools to help you optimize your code for better performance.

## Bugs

Unless you are a programming wunderkind (in which case, why are you reading this book?), you're going to have bugs crop up in your work from time to time. Flash's compiler and runtime engine will catch a lot of errors due to incorrect syntax, misspellings, and a host of other common coding mistakes. These are the bugs that give you immediate feedback and are usually very simple to correct. It's the bugs where nothing is *technically* wrong with the code but your game behaves incorrectly that programmers never want to encounter. Examples of these types of bugs might be degrading performance over time, memory leak/bloat issues, or even simply

unexpected output from your game engine. To address these kinds of issues, Flash has a set of tools we can use—and expand on—to speed up debugging. We'll look at a number of these tools and how to use them in your work.

### Traces

The most basic form of displaying information from your code is through the use of a *trace* statement. It prints any arguments you give it to the output window in Flash. Any objects that are passed to the method have their *toString* method called to determine how to display them. This is why generic objects will trace [object Object]. Arrays will trace their contents as a comma-delimited string (as if the *join* method had been called). When creating custom game objects, it's not a bad idea to override the *toString* method and display custom returns as well. For instance, if you have a player Sprite on the screen, a helpful *toString* return might be a list of its vital stats:

```
Player <name> - Health: 90%, Ammo: 30
```

Similarly, the main game engine object might list out important information:

```
Game Status - Time: 2:49, Enemies: 5, Projectiles: 2, Score: 1200
```

All objects have a *toString* method available to them for overriding. Here's how you might write such an override.

```
override public function toString():String {
    var status:String = "Game Status - ";
    status += "Time: " + gameTime + ", ";
    status += "Enemies: " + enemyList.length + ", ";
    status += "Projectiles: " + projectileList.length + ", ";
    status += "Score: " + score;
    return status;
}
```

Defining these *toString* substitutions makes the information coming back from a trace much more relevant when you're trying to solve a problem. However, once you get a bunch of traces on objects going (which happens very quickly if you don't comment out a trace after you're done with it), it can be a mess to try to sort through the very limited output window to find what you're looking for. This is where extending the trace functionality to a new class makes sense. Because trace is a top-level function, there's no way to override or extend it in the same way we might do with a class. However, we can create a class of static methods and properties that will increase the usefulness of traces. We'll call the class *TraceUtil*.

```

package {

    public class TraceUtil {

        public static const INFO:String = "info";
        public static const WARNING:String = "warning";
        public static const ERROR:String = "error";

        public static var throwErrors:Boolean = false;
        public static var filter:Array = [INFO, WARNING, ERROR];

        public function TraceUtil() {
            throw new Error("TraceUtil class cannot be
                instantiated.");
        }

        public static function Trace(message:String,
            category:String = INFO):void {
            if (category == ERROR && throwErrors)
                throw new Error(message);
            if (!filter || !filter.length || filter.
                indexOf(category) > -1)
                trace(message);
        }

        public static function TraceObject(message:String,
            object:Object, category:String = INFO):void {
            Trace(message, category);
            Trace(" " + object, category);
            for (var i:String in object) {
                Trace(" "+ i + ": " + object[i], category);
            }
        }
    }
}

```

This class will allow us to attach a filter to our traces as we perform them so that we can later cull out less important ones. I have predefined three different levels of priority in this class: INFO traces are those that are purely informational and less crucial (like a notification that a file has finished loading), WARNING traces are for situations that the game knows how to deal with but shouldn't have gotten into for one reason or another, and ERROR traces should be used to track down errors inside try/catch blocks or in places where you don't want to actually throw an error but you do want to be notified of a problem. By adding these strings (or any

others you wanted to create) to the filter array, only messages that correspond to these trace types will be displayed.

The first method in `TraceUtil` is called `Trace`, capitalized because of the name conflict with the trace statement itself. It simply traces out the given message, assuming the category matches what's in the filter array. In addition, if you're in the throes of debugging a serious issue, you can set the `throwErrors` flag to true and the method will throw an actual error when a message with that type is sent. When you're done testing and don't want runtime errors thrown, you can turn this flag back off.

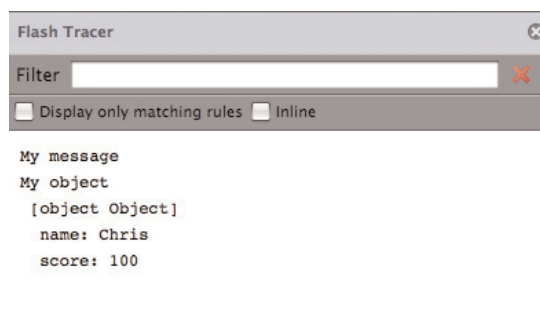
The other method in this class is called `TraceObject`. It is intended for use with dynamic objects; it relies on a *for...in* loop to iterate through the given object and trace out its properties on individual lines. In addition to normal dynamic objects, this will also work for Arrays and Vectors. It will format the properties, so they are indented slightly. This simple formatting will make scrubbing through the output window considerably easier.

## Flash Tracer

Unfortunately, as useful as traces can be, they're technically only available inside Flash. Luckily, a savvy programmer and Flash user Alessandro Crugnola created an extension for Firefox that works in conjunction with the Flash debug player to display traces in a handy window side by side with your content. It works on both Mac and Windows and is an excellent tool for debugging once you've left Flash and are in the browser. You can find a link to download Flash Tracer on this book's Web site, [www.flashgamebook.com](http://www.flashgamebook.com). Figure BC1.1 shows a shot of Flash Tracer displaying the traces from the `TraceUtil.fla` example.

## The Debugger

Sometimes traces are insufficient or too time-consuming to implement for solving a particular issue. Perhaps you want to view a



**Figure BC1.1** Flash Tracer allows you to view all your Flash traces inside the Firefox using the debug player.

number of values in an object at once or watch an object change as it is processed by a method. Traces will work here but are a brute force and messy solution. This is where Flash's debugger comes in very handy. If you've used the debugger in Flash versions prior to CS3, you probably just rolled your eyes. This is because the debugger in older versions of Flash was, sadly, buggy. It would not always activate correctly, it was hard to track down values, and it had a very poor UI for navigating the information when you *could* get it to work. For AS3, Adobe re-worked the debugger and it is a much more useful tool now. If you're unfamiliar with the debugger, it allows you to step through your code line by line as it is running to see exactly where a problem starts or an error occurs. You can define the point at which your code stops running automatically and switches to a line-by-line mode. This is known as a break point. You add it to your code in Flash by simply clicking to the left of a line number. You will see a red dot appears next to the line. This indicates that when the Flash player reaches this line, it will turn over control to you. Figure BC1.2 shows how this break point looks inside of Flash.

Once you've set a break point, you can run your SWF in debug mode by selecting Debug → Debug Movie from the main toolbar.

Once you've started the debug player, it will run until it encounters an error or a break point. It will then show the current stack of methods, as well as any variables relevant to the scope of the break point. In Fig. BC1.4, you can see how this information looks inside the Trace method from earlier in this chapter. We can see the values in the TraceUtil class, such as the filter array and throwErrors flag, and the values in the method we're currently inside (the message and category). Now try setting a break point in a piece of code and explore the debugger further on your own.

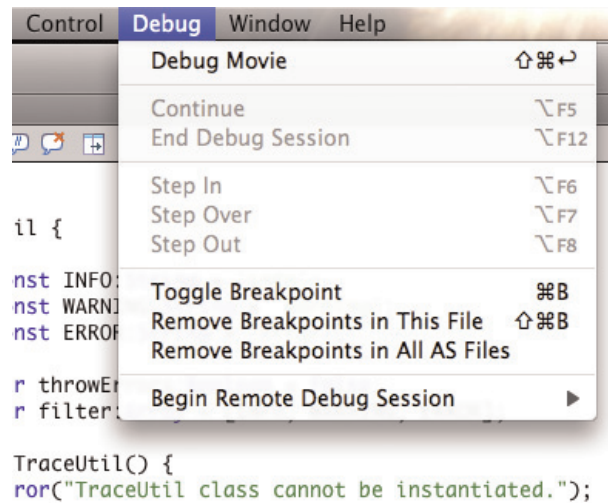
Like traces, debugging is very useful inside of Flash, but often errors and problems don't crop up until you're in the final phases of a project, often once a game is in the browser. If you have the Flash debug player installed in your browser (as any self-respecting Flash developer *should*), you can debug a SWF remotely with Flash

```

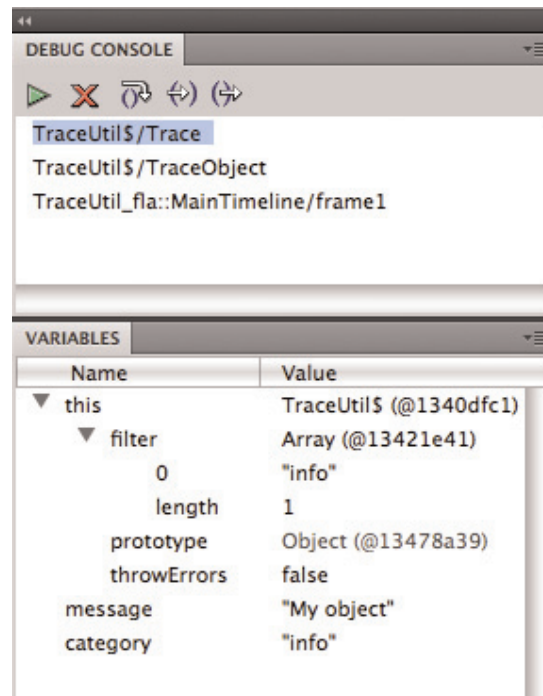
12     public function TraceUtil() {
13         throw new Error("TraceUtil class cannot be instantiated.");
14     }
15
16     public static function Trace(message:String, category:String = INFO):void {
17         if (category == ERROR && throwErrors)
18             throw new Error(message);
19         if (!filter || !filter.length || filter.indexOf(category) > -1)
20             trace(message);
21     }
    
```

**Figure BC1.2** When you click to the left of a line number in the Flash code editor, it adds a break point that will be caught by the debugger.

open in the background. To do this, you must first compile the SWF in debug mode; this includes some extra data (such as break points) that the debugger can pick up. You may have noticed that the size of a SWF in debug mode is larger than one in normal mode. Next, deploy your SWF on a remote server to test. Go into Flash and select Debug → Begin Remote Debug Session → ActionScript 3.0, per figure BC1.3.



**Figure BC1.3** Start a SWF in debug mode by selecting the Debug menu from the main toolbar.



**Figure BC1.4** The method stack in the debugger and the variable browser.

You'll notice at this point the output window goes into an idle mode, waiting for a connection. Finally, load the SWF in a browser and Flash will connect to it automatically. Although this can be a tedious series of steps to follow every time you deploy new builds to a server, it is invaluable when a bug occurs in the browser but not inside Flash.

## Performance and Optimization

In addition to traditional debugging (fixing errors), an important component of game QA is performance testing. This includes simply playing the game on a variety of different machines to determine the lowest threshold for your system requirements. If your game is going to be in front of a wide audience, or kids (who often have older, hand-me-down computers), it needs to be able to hold on even modest machines. Every game's needs are different—a puzzle game is likely going to need less CPU power and memory than a side-scrolling action game.

The catch to performance testing is on the surface, all you have is a “feeling” for how the game is supposed to play. Obviously, if a game is maxing out your CPU usage every moment it's running, you have a problem, but usually the differences are subtler. You can only eyeball the framerate, and most system memory viewers don't separate browsers into discrete threads, so it can be difficult to get an accurate gauge of just how the game is performing. In this section, we'll create a couple of tools you can use to add basic framerate and memory monitoring to your game. We'll combine both classes into an FLA called PerformanceProfiler, which you can open from the Bonus Chapter 1 examples folder.

### The FrameRateProfiler Class

Most Flash games don't consistently run at a perfect fixed framerate. The rate fluctuates based on how much information the Flash player is trying to process at that moment and what else is going on in the browser (and the rest of the system, for that matter). When you profile a game for framerate performance, you want to take a sample over a period of time and average it. To do this, we will create a simple Sprite-based component that will allow us to monitor the framerate right on the Stage.

```
package {  
  
    import flash.display.Sprite;  
    import flash.events.Event;  
    import flash.utils.getTimer;  
    import flash.text.TextField;
```

```
public class FrameRateProfiler extends Sprite {  
  
    private var _previousTime:int;  
    private var _sampleSize:int = 30;  
    private var _sample:Vector.<Number>;  
  
    [Inspectable(defaultValue = 1,name = "Decimal  
    Precision")]  
    public var precision:uint = 1;  
  
    public var textField:TextField;
```

To start the class, we set up variables to store the number of samples we'll collect, as well as a `Vector` object to keep track of them and the level of decimal precision we want to display when viewing the framerate. Since this is going to be a component, we'll expose the precision variable, so this can be set from inside the Flash component inspector. We expose it using the `Inspectable` metadata tag.

```
public function FrameRateProfiler() {  
    addEventListener(Event.ADDED_TO_STAGE, addedToStage,  
        false, 0, true);  
    addEventListener(Event.REMOVED_FROM_STAGE,  
        removedFromStage, false, 0, true);  
    _sample = new Vector.<Number>();  
}  
  
[Inspectable(defaultValue = 30, name = "Sample Size")]  
public function set sampleSize(value:int):void {  
    _sampleSize = Math.max(1, value);  
}  
  
public function get sampleSize():int { return _sampleSize; }  
  
[Inspectable(defaultValue = 0x000000, type = "Color", name = "Text  
    Color")]  
public function set color(value:uint):void {  
    textField.textColor = value;  
}  
  
public function get color():uint { return textField.textColor; }
```

In the constructor, we simply initialize the `Vector` and assign listeners, so the component knows when it is added to and removed from the Stage. Next, we expose two other accessor methods for defining the sample size (number of frames we'll use to get our average) and the color of the textfield. This way the same

component can be used on a variety of different backgrounds and still be readable.

```
private function addedToStage(e:Event):void {
    addEventListener(Event.ENTER_FRAME, onEnterFrame, false,
        0, true);
    _previousTime = getTimer();
}

private function removedFromStage(e:Event):void {
    removeEventListener(Event.ENTER_FRAME, onEnterFrame);
}

private function onEnterFrame(e:Event):void {
    var newTime:int = getTimer();
    var rate:Number = 1000/(newTime - _previousTime);
    _sample.push(rate);
    if (_sample.length > _sampleSize) _sample.shift();
    _previousTime = newTime;
    var avg:Number = 0;
    for each (var value:Number in _sample)
        avg += value;
    avg /= _sample.length;
    textField.text = avg.toFixed(precision);
}
```

When the component is added to the Stage, the `onEnterFrame` script starts getting called every frame cycle. Each time it fires, we determine the amount of time between frames and divide it into 1000 ms to determine the framerate. This rate is then added to the samples Vector. Once the Vector has exceeded the sample size we specified earlier, older elements at the front are removed. The text-field is then updated with an average of the sample. Here's the class in its entirety:

```
package {

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.utils.getTimer;
    import flash.text.TextField;

    public class FrameRateProfiler extends Sprite {

        private var _previousTime:int;
        private var _sampleSize:int = 30;
        private var _sample:Vector.<Number>;
```

```
[Inspectable(defaultValue = 1, name = "Decimal
Precision")]
public var precision:uint = 1;

public var textField:TextField;

public function FrameRateProfiler() {
    addEventListener(Event.ADDED_TO_STAGE,
        addToStage, false, 0, true);
    addEventListener(Event.REMOVED_FROM_STAGE,
        removeFromStage, false, 0, true);
    _sample = new Vector.<Number>();
}

[Inspectable(defaultValue = 30, name = "Sample Size")]
public function set sampleSize(value:int):void {
    _sampleSize = Math.max(1, value);
}

public function get sampleSize():int
{ return _sampleSize; }

[Inspectable(defaultValue = 0x000000, type = "Color",
name = "Text Color")]
public function set color(value:uint):void {
    textField.textColor = value;
}

public function get color():uint { return textField.
    textColor; }

private function addToStage(e:Event):void {
    addEventListener(Event.ENTER_FRAME,
        onEnterFrame, false, 0, true);
    _previousTime = getTimer();
}

private function removeFromStage(e:Event):void {
    removeEventListener(Event.ENTER_FRAME,
        onEnterFrame);
}

private function onEnterFrame(e:Event):void {
    var newTime:int = getTimer();
    var rate:Number = 1000/(newTime -
        _previousTime);
    _sample.push(rate);
}
```

```

        if (_sample.length > _sampleSize) _sample.shift();
        _previousTime = newTime;
        var avg:Number = 0;
        for each (var value:Number in _sample)
            avg += value;
        avg /= _sample.length;
        textField.text = avg.toFixed(precision);
    }
}
}

```

If you open the `PerformanceProfiler.fla` file, you can see how this class is attached to a clip in the library called `FrameRate Profiler`, as well as in the component definition for that same clip. Once on the Stage, you can set the three exposed properties in the Component Inspector panel, which is accessible from the Window toolbar. Next we'll look at a similar class for profiling memory usage.

## The MemoryProfiler Class

In this class, we'll create a component much like our `FrameRateProfiler`, which will simply monitor the amount of memory the Flash player is using over time. This will allow you to see any large spikes in memory usage, which are directly caused by Flash or your game.

```

package {

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;
    import flash.system.System;
    import flash.text.TextField;

    public class MemoryProfiler extends Sprite {

        private var _timer:Timer;
        private var _interval:Number = 5;

        public var textField:TextField;

        public function MemoryProfiler() {
            addEventListener(Event.ADDED_TO_STAGE,
                addToStage, false, 0, true);
            addEventListener(Event.REMOVED_FROM_STAGE,
                removeFromStage, false, 0, true);
            _timer = new Timer(interval * 1000);
        }
    }
}

```

Because memory does not fluctuate at the same frequency as framerate, we only need to check the memory usage every few seconds. Much more often than that and you'll be taxing the processor more than you need to do for the same result. By default, this component checks every five seconds.

```
[Inspectable(defaultValue = 5,name = "Sample Interval")]
public function set interval(value:Number):void {
    _interval = Math.max(1, value);
    _timer.delay = _interval * 1000;
}

public function get interval():Number { return _interval; }

[Inspectable(defaultValue = 0x000000, type = "Color", name = "Text
Color")]
public function set color(value:uint):void {
    textField.textColor = value;
}

public function get color():uint { return textField.textColor; }
```

Like the previous class, we expose two values to the Component Inspector: the interval at which we want to check the system memory and the color of the textfield.

```
private function addedToStage(e:Event):void {
    _timer.addEventListener(TimerEvent.TIMER, onTimer, false,
        0, true);
    _timer.start();
    onTimer(null);
}

private function removedFromStage(e:Event):void {
    _timer.stop();
}

private function onTimer(e:TimerEvent):void {
    var memoryUsed:Number = System.totalMemory/1024;
    var memoryUnit:String = "k";
    if (memoryUsed > 1024) {
        memoryUsed /= 1024;
        memoryUnit = "mb";
    }
    textField.text = "Memory Used: " + memoryUsed.toFixed(1) +
        memoryUnit;
}
```

As before, adding and removing the component starts and stops the Timer, respectively. Every time the Timer object runs, it converts the amount of system memory used to kilobytes (k). If the amount of memory used is about 1024 k (1 MB), we use megabytes instead. Here is the class in its entirety, for context:

```
package {

    import flash.display.Sprite;
    import flash.events.Event;
    import flash.events.TimerEvent;
    import flash.utils.Timer;
    import flash.system.System;
    import flash.text.TextField;

    public class MemoryProfiler extends Sprite {

        private var _timer:Timer;
        private var _interval:Number = 5;

        public var textField:TextField;

        public function MemoryProfiler() {
            addEventListener(Event.ADDED_TO_STAGE,
                addedToStage, false, 0, true);
            addEventListener(Event.REMOVED_FROM_STAGE,
                removedFromStage, false, 0, true);
            _timer = new Timer(interval * 1000);
        }

        [Inspectable(defaultValue = 5, name = "Sample
            Interval")]
        public function set interval(value:Number):void {
            _interval = Math.max(1, value);
            _timer.delay = _interval * 1000;
        }

        public function get interval():Number
        { return _interval; }

        [Inspectable(defaultValue = 0x000000, type = "Color",
            name = "Text Color")]
        public function set color(value:uint):void {
            textField.textColor = value;
        }

        public function get color():uint { return textField.
            textColor; }
    }
}
```

```
private function addedToStage(e:Event):void {
    _timer.addEventListener(TimerEvent.TIMER,
        onTimer, false, 0, true);
    _timer.start();
    onTimer(null);
}

private function removedFromStage(e:Event):void {
    _timer.stop();
}

private function onTimer(e:TimerEvent):void {
    var memoryUsed:Number = System.totalMemory/
        1024;
    var memoryUnit:String = "k";
    if (memoryUsed > 1024) {
        memoryUsed /= 1024;
        memoryUnit = "mb";
    }
    textField.text = "Memory Used: " + memoryUsed.
        toFixed(1) + memoryUnit;
}
}
```

Together in the PerformanceProfiler document, these components provide a lot of useful information. You could even convert them to precompiled clips (by right-clicking on them in the library and selecting Covert to Compiled Clip), at which point they would be entirely self-contained and portable to any file, regardless of where the class AS files are. When combined with the Activity Monitor on a Mac, or the Task Manager on Windows, you can use these two components to easily do real-time, statistical monitoring of your game on multiple machines.

## The Sample Package

Sometimes in the process of debugging and optimization, you're able to narrow down the source of a performance drain to a handful of culprits. Game engines that manage a lot of data (like a large action game) often generate a lot of hefty objects in memory to store all the data. Prior to Flash CS4, it was not possible to get information about how much memory objects actually consumed, other than as a sum total. In CS4, Adobe introduced a set of classes that were originally part of the Flex Builder debugging set. This package and its primary class are known as the Sampler. This is because its primary function is to take samples of memory data to determine which methods are called the most and which objects are eating the

most resources. It's worth noting that these classes and methods only function as expected within the debug Flash player, and they are also compatible with certain versions of Flash player 9 if you are still targeting that platform.

Sampling memory can become complicated, and the sampler package is capable of being used to write a full in-Flash debugging tool for monitoring performance. That said, we'll just be taking a quick look at a couple of the more useful "quick" methods that can be used when trying to track down memory leaks.

The *getSize* method at the root level of the sampler package accepts any object as a parameter and returns the amount of memory that object consumes in bytes. In the Bonus Chapter 1 examples folder, you can find a file called *getSizeExample fla*. All it has is a little code on the first frame to demonstrate memory usage.

```
import flash.sampler.*;

var arr1:Array = new Array(1, 2, 3, 4, 5);
var arr2:Array = new Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
var arr3:Array = new Array();

trace("Array 1:",getSize(arr1),"bytes");
trace("Array 2:",getSize(arr2),"bytes");
trace("Array 3:",getSize(arr3),"bytes");
```

In this example, there are three different arrays, each with a different number of values in them. The traces output the following:

```
Array 1: 60 bytes
Array 2: 80 bytes
Array 3: 40 bytes
```

As you might expect, Array 2 has twice as many elements as Array 1 and takes up twice as much space over the empty Array 3. This is also helpful because it shows the cost of an object before it has even been populated with any data. Doing a little bit of math reveals that each Number (or integer, in this case) consumes 4 bytes of memory. This number may be small on its own but considers that most objects are far more complicated and contain many numbers, strings, Booleans, and other objects. Every little piece of data adds up and being conscientious of it at the onset of development will help prevent problems later. To put it in perspective, refer back to the MixUp game from Chapter 13. The main game object was around 500 bytes, but each individual square of bitmap data was 100 k (12 for a total of 1.2 MB). Code will almost always be less than raw assets, such as sounds or images.

Another useful set of methods in the Sampler package are *getInvocationCount*, *getSetterInvocationCount*, and *getterInvocationCount*. These three return the number of times a method

(or getter/setter) has been called on an object. This information can be helpful because it acts as an indicator for where your time spent optimizing code should be spent. Say you have a game that involves AI making decisions about where enemy players move during gameplay. The logic involved is likely going to poll a number of methods frequently. Taking a measurement of the frequency with which these methods are called at the end of a game will give you a summary you can use to determine where to focus your time. I've created a `SamplerUtil` class, similar to `TraceUtil`, which provides a static method for getting this information easily. It also makes use of an overridden `toString` method to make the returned data easily readable. However, before we look at the function for polling we should look at a helper class I also created to store the information retrieved about the methods.

```
internal class MethodObject extends Object{

    public var name:String;
    public var count:int;
    public var getCount:int;
    public var setCount:int;

    public function MethodObject(name:String,
                                  count:int = 0,
                                  getCount:int = 0,
                                  setCount:int = 0) {

        this.name = name;
        this.count = count;
        this.getCount = getCount;
        this.setCount = setCount;
    }

    public function toString():String {
        if (count > 0) {
            return "Method " + name + " called " + count +
                " times.";
        } else {
            return "Accessor " + name + " set " + setCount + "
                times and gotten " + getCount + " times.";
        }
    }
}
```

The `MethodObject` class is simply a data container. We could have also used a generic object, but because we'll be creating a bunch of these, it is better to statically type the properties we will be using. The `toString` method is also easier to add in this format. Each of these objects keeps track of how many times a method or

getter/setter is called. The *toString* method checks to see what type of function it is based on the number of types of calls and returns an appropriately formatted string. It's worth noting at this point that if an accessor function is read only (just a *get*, no *set*), the *setCount* will be -1. Now we'll look at the method that uses these objects.

```
package {

    import flash.sampler.*;

    public class SamplerUtil {

        public static function pollMethods(obj:Object):Array {
            var methods:Array = new Array();
            for each (var name:QName in getMemberNames(obj)) {
                var methodObject:MethodObject;
                if (isGetterSetter(obj, name)) {
                    methodObject = new MethodObject
                        (name.localName,

0,

                    getGetterInvocationCount(obj, name),

                    getSetterInvocationCount(obj, name));
                } else {
                    methodObject = new MethodObject
                        (name.localName,

                    getInvocationCount(obj, name));
                }
                if (methodObject.count > 0 ||
                    methodObject.getCount > 0 ||
                    methodObject.setCount > 0)
                    methods.push(methodObject);
            }
            methods.toString = function() {
                return this.join("\n");
            }
            return methods;
        }
    }
}
```

When the *pollMethods* function is called, it creates an array and then iterates through the passed object with the *getMemberNames* method of the *Sampler* package. This method will return all public and private member variables of a class object. What is returned is a list of *QName* objects.

## QNAME?

The QName (or Qualified Name) class is usually associated with XML, as it is part of the E4X standard. We won't cover its use with XML, but in regular ActionScript, it stores a properly packaged reference to the name of a specific member variable. To get the name of the variable as we're used to looking at it, you simply specify the `localName` property of a QName object. Most of the time you likely won't deal with QNames, unless you're working very heavily with XML.

A new `MethodObject` is created for each QName object returned, storing its `localName` property and the number of times it was invoked, either as a method or a getter/setter. Since we're only interested in methods, not just plain properties, we don't bother to add to the array any objects that don't have at least one invocation. When the loop is finished running, we have an array of only those methods that were called at least once, and it can easily be sorted based on the "count," "getCount," or "setCount" properties of each `MethodObject`. As a last step before returning the array, we format the `toString` method of the array to use line breaks to separate each element (as opposed to the default commas). This will make any trace statements with the array automatically formatted for readability.

I encourage you to continue exploring the rest of the `Sampler` package and use it to create helpful debugging tools. Hopefully, Adobe will continue to provide us with more information we can use in future versions of the Flash player. My fingers are crossed for CPU and GPU usage polling.

## Summary

Debugging and optimization are extremely important tasks in game development that should never be omitted because of lack of time (or any other reason, for that matter). If your game is buggy or sluggish, people won't want to play it.