Code optimization is an evidence-based discipline. All optimizations detailed in Chapter 15, ActionScript Optimization were verified across different versions of AIR. The results from various tests carried out are provided here along with links to their source code.

At the time of writing, the most recent version of the AIR SDK (3.1) was overlaid onto Flash Professional CS5.5 and used. The tests were also carried out using AIR 2.0 to accommodate Flash Professional CS5 users.

There are no guarantees that these optimizations will apply to future versions of AIR. However, the source code for these tests has been provided if you wish to run them yourself.

The table below shows the four devices used for testing:

<table>
<thead>
<tr>
<th>Device and model</th>
<th>iOS version</th>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-generation iPod touch</td>
<td>3.1.3</td>
<td>412 MHz</td>
<td>128 MB</td>
</tr>
<tr>
<td>Third-generation iPod touch</td>
<td>5.0.1</td>
<td>600 MHz</td>
<td>256 MB</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>5.0.1</td>
<td>800 MHz</td>
<td>512 MB</td>
</tr>
<tr>
<td>iPad 2</td>
<td>5.0.1</td>
<td>1 GHz</td>
<td>512 MB</td>
</tr>
</tbody>
</table>

Each device was running iOS 5 with the exception of the first-generation iPod touch, which had iOS 3.1 installed. As older ARMv6 devices are not supported by more recent versions of AIR, the first-generation iPod touch was omitted from the AIR 3.1 tests.
The execution time of each test was used to compare the performance benefits of each optimization. The results shown throughout this chapter are an average taken from eight runs of each test.

Execution times are measured in milliseconds and shown throughout this Appendix.

### Declaring data types

Two tests were run to compare the performance benefits of using typed variables against un-typed variables.

Test A performs a series of operations on a collection of un-typed variables, whereas Test B performs the same operations using typed versions of those variables.

The following code snippet shows some of the un-typed variables from Test A being declared:

```javascript
var a = 1;
var c = 1.5;
var e = "hullo ";
```

You can see the same variables in Test B but with a data type being explicitly declared for each:

```javascript
var a:int = 1;
var c:Number = 1.5;
var e:String = "hullo ";
```

Each test performs 100,000 iterations.

The complete source code for both tests is available from the book's accompanying code bundle at appendix-b\declaring-data-types\.

Let us take a look at the performance measurements, starting with AIR 2.0 used by Flash Professional CS5.
AIR 2.0

The following table shows the times taken for each test to run:

<table>
<thead>
<tr>
<th>Device</th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td>1576</td>
<td>72</td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>590</td>
<td>38</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>477</td>
<td>28</td>
</tr>
<tr>
<td>iPad 2</td>
<td>177</td>
<td>14</td>
</tr>
</tbody>
</table>

Displaying the same data within the following chart really highlights the performance advantage of declaring each variable's type:
AIR 3.1

Similarly impressive gains are to be had when using AIR 3.1:

<table>
<thead>
<tr>
<th>Device</th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td>623</td>
<td>32</td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>498</td>
<td>25</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>167</td>
<td>18</td>
</tr>
</tbody>
</table>

The test results are also shown in the following chart:

Replacing objects with custom classes

Two tests were run to compare the difference in performance between accessing properties of an Object instance and properties of a custom class.

Test A accesses and updates properties belonging to an Object, whereas Test B performs the same operations on an equivalent custom class.

The following code snippet shows the Object from Test A being defined:

```javascript
var player:Object = {
  lives: 3,
  energy: 100,
  score: 0
};
```
Here’s the equivalent custom class used by Test B:

```javascript
public class Player {
    public var lives :uint = 3;
    public var energy:uint = 100;
    public var score :uint = 0;
}
```

Each test performs 100,000 iterations.

The complete source code for each test is available from the book’s accompanying code bundle at appendix-b\object-v-class\.

Let us take a look at the performance measurements from using both AIR 2.0 and AIR 3.1.

**AIR 2.0**

From the following table, it is clear that accessing a dynamic object is significantly slower than accessing an instance of a custom class:

<table>
<thead>
<tr>
<th>Device</th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td>2537</td>
<td>225</td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>646</td>
<td>95</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>483</td>
<td>74</td>
</tr>
<tr>
<td>iPad 2</td>
<td>299</td>
<td>36</td>
</tr>
</tbody>
</table>

The following chart shows the same data:
AIR 3.1

Now let us take a look at the results of the tests compiled from Flash Professional CS5.5 using AIR 3.1:

<table>
<thead>
<tr>
<th></th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>640</td>
<td>26</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>485</td>
<td>20</td>
</tr>
<tr>
<td>iPad 2</td>
<td>318</td>
<td>16</td>
</tr>
</tbody>
</table>

Again, there is a huge performance increase when accessing a custom class as opposed to a dynamic equivalent. The test results are also shown in the following chart:
Optimizing loops

Three tests were run to check for performance benefits when applying various optimizations to `for` loops. Each test iterates over an array of 100,000 elements.

Test A uses an iterator of type `Number` and queries the array's length when evaluating the loop's condition expression. The following code snippet illustrates this:

```javascript
for(var i:Number=0; i<array.length; i++)
{
    // Do something
}
```

Test B uses an iterator of type `int` and queries the array's length when evaluating the loop's condition expression:

```javascript
for(var i:int=0; i<array.length; i++)
{
    // Do something
}
```

Test C uses an iterator of type `int` and caches the array's length within a local variable, avoiding the need to query the array's length as part of the loop's condition expression:

```javascript
var len:int = array.length;
for(var i:int=0; i<len; i++)
{
    // Do something
}
```

The complete source code for each test is available from the book's accompanying code bundle at `appendix-b\loop-optimizations`.

We will start by taking a look at the performance measurements from running the three tests using AIR 2.0, before moving onto AIR 3.1.
AIR 2.0

The following table shows the times taken for each of the three tests to run:

<table>
<thead>
<tr>
<th></th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
<th>Test C (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td>746</td>
<td>723</td>
<td>34</td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>148</td>
<td>122</td>
<td>17</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>109</td>
<td>91</td>
<td>11</td>
</tr>
<tr>
<td>iPad 2</td>
<td>62</td>
<td>63</td>
<td>7</td>
</tr>
</tbody>
</table>

Whereas Test B provides relatively small gains in performance, Test C shows a significant improvement. This is visualized in the following chart:
AIR 3.1

Using AIR 3.1, Tests A and B perform significantly faster than their AIR 2.0 counterparts. It is clear that ADT is applying optimizations to the loops that aren’t being applied by AIR 2.0’s PFI tool. Test C shows that ActionScript optimization is still beneficial in AIR 3.1, but expect the gains to be modest.

<table>
<thead>
<tr>
<th>Device</th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
<th>Test C (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td>33</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>42</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>33</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>iPad 2</td>
<td>12</td>
<td>11</td>
<td>7</td>
</tr>
</tbody>
</table>

The test results are also shown in the following chart:
Replacing arrays with vectors

Four tests were run to evaluate the strength of using vectors over arrays. Each test creates and initializes a collection of 100,000 values.

Test A creates an empty array and uses a `for` loop to populate each index of the array. The following code block shows this:

```actionscript
code
const SIZE:int = 100000;
var collection:Array = [];
for(var i:int=0; i<SIZE; i++)
{
    collection[i] = 0;
}
```

Test B creates the array's size ahead of time. It then uses a `for` loop to initialize each element:

```actionscript
code
const SIZE:int = 100000;
var collection:Array = new Array(SIZE);
for(var i:int=0; i<SIZE; i++)
{
    collection[i] = 0;
}
```

Test C creates an empty vector and uses a `for` loop to populate each index of the vector:

```actionscript
code
const SIZE:int = 100000;
var collection:Vector.<int> = new Vector.<int>();
for(var i:int=0; i<SIZE; i++)
{
    collection[i] = 0;
}
```

Test D creates a fixed-sized vector containing 100,000 elements. It then uses a `for` loop to initialize each element:

```actionscript
code
const SIZE:int = 100000;
var collection:Vector.<int> = new Vector.<int>(SIZE, true);
for(var i:int=0; i<SIZE; i++)
{
    collection[i] = 0;
}
```
The complete source code for each test is available from the book's accompanying code bundle at appendix-b\arrays-v-vectors\.

Let us see how each test performs, starting with the results using Flash Professional CS5 and AIR 2.0.

### AIR 2.0

The following table shows that vectors (Tests C and D) have faster read and write access than arrays (Tests A and B). A fixed-size vector is particularly fast as it doesn't grow in size, preventing the need for additional memory allocations during its lifetime.

<table>
<thead>
<tr>
<th></th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
<th>Test C (ms)</th>
<th>Test D (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td>341</td>
<td>312</td>
<td>161</td>
<td>92</td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>109</td>
<td>106</td>
<td>61</td>
<td>44</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>85</td>
<td>86</td>
<td>47</td>
<td>33</td>
</tr>
<tr>
<td>iPad 2</td>
<td>49</td>
<td>50</td>
<td>29</td>
<td>17</td>
</tr>
</tbody>
</table>

The test results are also shown in the following chart:
AIR 3.1

When compiling with AIR 3.1, fixed-size vectors still have a significant performance benefit. However, it is clear that the gap has been bridged when comparing arrays (Tests A and B), and vectors that can dynamically grow in size (Test C).

<table>
<thead>
<tr>
<th></th>
<th>Test A (ms)</th>
<th>Test B (ms)</th>
<th>Test C (ms)</th>
<th>Test D (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1G iPod touch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3G iPod touch</td>
<td>99</td>
<td>84</td>
<td>78</td>
<td>37</td>
</tr>
<tr>
<td>iPhone 4</td>
<td>76</td>
<td>66</td>
<td>58</td>
<td>28</td>
</tr>
<tr>
<td>iPad 2</td>
<td>39</td>
<td>39</td>
<td>40</td>
<td>16</td>
</tr>
</tbody>
</table>

You can also see the execution times presented in the following chart: