15
ActionScript Optimization

In this chapter, we will cover:

- Declaring data types
- Replacing Objects with custom classes
- Optimizing loops
- Replacing arrays with vectors
- Recycling instances
- Using static members and methods
- Using callback methods
- Taking advantage of event bubbling

Introduction

Given the hardware capabilities of even modest computers and laptops, Flash developers targeting the desktop rarely have to worry about performance. However, it is a very different story when developing for iOS. An un-optimized application can often ask too much of the device, causing sluggish performance and in some cases, even leading to a crash.

While graphics rendering is a common bottleneck that can impact performance, another that is often overlooked is the application's code itself. Poorly written ActionScript can have a detrimental effect on performance and can unnecessarily consume vast amounts of memory. Many of the features of ActionScript 3.0, which make it so easy to use, are, unfortunately, some of the biggest culprits.

In this chapter, we will cover a range of optimizations that can be applied to your ActionScript. We will see how even the simplest of changes can provide improvements in execution speed and also learn how to minimize memory usage. The exact performance gain obtained from a recipe will vary depending on the version of AIR being used, the devices you wish to target, and the context within which the optimization is being applied.

It is important to note that there are many online resources and articles related to ActionScript optimization. However, be aware that most of these articles pertain to the Flash and AIR players for desktop and may not be applicable to AIR for iOS. Remember, AIR for iOS compiles an application's ActionScript into native ARM machine code rather than relying on a virtual machine. Although often similar, **Ahead of Time** (**AOT**) compilation and **Just in Time** (**JIT**) compilation don't always provide the same performance characteristics. It is, therefore, vitally important that you measure the performance of your ActionScript to assess if gains have been had from any potential optimizations. Measuring the performance of your code changes is fundamental to code optimization. Also, never assume that an optimization will hold true for future releases of AIR, as Adobe may change the architecture of the ADT compiler.

It is generally accepted that most code optimizations should be performed as a last resort. After all, significantly larger improvements can be had from selecting more efficient algorithms within your code. However, much of what is covered in this chapter requires so little effort, and is regarded as good practice, that it may well be worth applying as you develop.

Declaring data types

ActionScript 3.0 allows you to selectively opt out of compile-time type checking. However, no self-respecting developer worth their salt should ever leave a variable un-typed without good reason. Doing so can hurt an app's run-time performance.

Let us look at an example and see how to correct it.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe1\recipe.fla` into Flash Professional.

How to do it...

Apply the following steps to your document class:

1. Within the class' constructor you'll find the following variable declarations:

   ```
   var a = 1;
   var b = 1.5;
   var c = "Hello World";
   ```

2. Use type annotation to explicitly assign the most appropriate data type for each:

   ```
   var a:int = 1;
   var b:Number = 1.5;
   var c:String = "Hello World";
   ```

3. Save your changes.

How it works...

Using type annotation is not only one of the simplest changes you can make to your ActionScript, but also produces the single biggest improvement in run-time performance. Working with typed variables can often result in a five to ten-fold increase in performance.

It is not just local variables that can benefit either. Where possible, you should explicitly declare a data type for all variables used within your code, no matter their scope—global variables, class members, constants, static variables, and parameters are all prime candidates.

It is also important that you select the most appropriate data type for each variable. For example, don't use a `Number` if you know you will only be working with integer values.

Performance measurements for this recipe's optimizations can be found in Appendix B. The code used for testing can also be found in the book's accompanying code bundle at `appendix-b\declaring-data-types\`.

There's more...

Here are a few more pieces of advice when declaring variables.

Explicitly opting out

On occasions you may intentionally wish to use an un-typed variable. You can explicitly declare a variable as un-typed by using the special asterisk annotation—`:*`. Here is an example:

```
var a:* = 1;
var b:* = 1.5;
var c:* = "Hello World";
```

This snippet of ActionScript is functionally identical to our original three un-typed variables. If you wish to declare a variable as un-typed, then it is better to explicitly state your intentions for reasons of readability and to avoid any compiler warnings.

Unsigned integers

ActionScript 3.0 supports both the `int` and `uint` types. While it is important that you carefully select between the two to ensure the maximum range of values for your variable, selecting one type over the other will, for the most part, have no discernable effect on performance.

See also

- *Replacing Objects with custom classes*
- *Optimizing loops*

Replacing Objects with custom classes

Being a dynamic language, it is often tempting to use ActionScript's `Object` type to represent data. Although convenient to use, accessing properties of `Object` instances can be hugely inefficient and should be avoided where possible.

This recipe will show you how to replace an `Object` instance with a faster and more appropriate alternative.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe2\recipe.fla` into Flash Professional.

How to do it...

Perform the following optimizations:

1. Within the constructor, you will find the following variable declaration:

   ```
   var player:Object = {
     lives: 3,
     energy: 100,
     score: 0
   };
   ```

   It uses an `Object` to represent a player from a game and contains properties that represent the player's number of lives, amount of energy, and current score. A default value has been set for each property.

   Let us replace the `Object` definition with a custom class.

2. First, create a new ActionScript 3.0 class and name it `Player`.

3. Add the following code to it:

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

4. Save the class file as `Player.as` within the FLA's root folder.
5. Now move back to the document class and replace the code within its constructor with the following:

   ```
   var player:Player = new Player();
   ```

6. Save your document class.

You have successfully replaced your `Object` with a class-based alternative.

How it works...

The properties belonging to both versions can be accessed identically. This is shown in the following code snippet, where the `player` variable could just as easily be an `Object` or instance of the custom `Player` class:

```
player.score += 100;
player.energy = 50;
player.lives--;
```

However, performing such operations on an instance of the `Player` class will be significantly faster than the `Object` instance. This is due to the fact that the class' typed member variables exhibit superior runtime performance when being accessed compared to the `Object` instance's un-typed equivalents.

When an object's properties are known up-front, there is very little reason to use the `Object` type over a custom class. Also, when writing a custom class, remember to explicitly declare each member variable's data type. After all, this is where the performance advantage comes from.

You should endeavor to only use the dynamic features of ActionScript 3.0 where appropriate. For example, an `Object` would be the correct type to use if there was a requirement to add properties dynamically at runtime.

Performance measurements for this recipe's optimizations can be found in Appendix B. The code used for testing can also be found in the book's accompanying code bundle at `appendix-b\object-v-class\`.

See also

- *Declaring data types*

Optimizing loops

Loops are heavily relied upon, making them an obvious target when optimizing your ActionScript. Being the most often used, we will apply some simple optimizations to a `for` loop. In particular, we will focus on the use of a `for` loop to iterate over an array of data.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe3\recipe.fla` into Flash Professional.

How to do it...

Take a look at the code within your document class.

1. The class' constructor contains the following loop:

   ```
   var array:Array = new Array(1000000);
   for(var i:Number = 0; i < array.length; i++)
   {
     // Do something
   }
   ```

   The array's length dictates the number of loops that will take place, with the iterator variable, `i`, being incremented at the end of each looping sequence. However, take a look at the variable's type. Although the iterator will always represent an integer value, a `Number` type is actually being used to represent it.

2. Change the iterator variable's type from a `Number` to an `int`:

   ```
   var array:Array = new Array(1000000);
   for(var i:int = 0; i < array.length; i++)
   {
     // Do something
   }
   ```

3. Also notice that the array's length is being queried on each iteration as part of the `for` loop's condition expression. Rather than evaluating this statement within the loop, a saving can be made by storing the array's length outside the loop and re-using it. To do this, make the following changes:

   ```
   var array:Array = new Array(1000000);
   var len:int = array.length;
   for(var i:int = 0; i < len; i++)
   {
     // Do something
   }
   ```

4. Save your document class' changes.

How it works...

Using the `int` type for a loop's iterator variable will always be faster than using the `Number` type. This is because `int` is quicker than other types for the addition and subtraction operations typically performed upon iterators. There really is no excuse for not selecting the correct type. Try to get into the habit of doing this as you develop, rather than later in the project's lifecycle.

Just as important is the time taken to evaluate the loop's condition statement. If you know this statement will return the same result on each iteration, then store its value within a local variable outside the loop and re-use it. This is also true of other expressions being evaluated within your loop. Move them outside the loop if their values do not change.

Both these minor changes will increase the execution speed of the `for` loop, but whether or not any meaningful performance gains can be made really depends on the context within which the loop is used. For example, there may be very little benefit from optimizing a loop that has iterated over, say 50 times. Similarly, optimizing a 10,000 iteration loop, could gain little if it is called only once during application start up.

Instead, invest your time and energy optimizing loops that matter. Nested loops, and in particular inner loops (they are executed the most number of times), are prime targets for optimization. This is also true of loops that are executed continuously throughout an application's lifetime. Even tiny improvements can lead to significant reductions in CPU usage.

In such cases, focus on the block of statements to be executed within the loop, and make each statement as lean as possible. This goes for any methods that are called from within the block too. Considering the number of iterations that can be performed during a loop, any saving made here can be significant and can far outweigh the savings made from this recipe's trivial code example.

Performance measurements for this recipe's optimizations can be found in Appendix B. The code used for testing can also be found in the book's accompanying code bundle at `appendix-b\loop-optimizations\`.

There's more...

You may also find the following points regarding loops of some value.

Accessing arrays

The `Array` class is optimized for integer indexing. This further strengthens the case for using a loop iterator of type `int` rather than `Number`. Take the following as an example, where a `for` loop is used to initialize the elements of an array:

```
var array:Array = new Array(1000000);
var len:int = array.length;
for(var i:int = 0; i < len; i++)
{
  array[i] = 0;
}
```

Using an `int` instead of a `Number` allows significantly faster access to the array's elements. If the `Number` data type was used, then it would first have to be downcast to an `int` before each operation could take place.

Performance measurement

You can measure the performance of your ActionScript by timing how long it takes to execute. This can be achieved by using the `flash.utils.getTimer()` package-level function. Here is an example that times how long it takes an un-optimized `for` loop to execute:

```
var startTime:int = getTimer();
var array:Array = new Array(1000000);
for(var i:Number = 0; i < array.length; i++)
{
  // Do something
}
var endTime:int = getTimer();
var duration:int = endTime - startTime;
```

A `getTimer()` call is made immediately before and after the loop. The results from both are used to calculate the number of milliseconds the loop took to run. This simple technique will allow you to see if any attempted optimizations have actually been beneficial.

Remember to first import `getTimer()` before using it within a class:

```
import flash.utils.getTimer;
```

You can find additional detail regarding the `getTimer()` function by searching for `flash.utils.getTimer()` from Adobe Community Help.

See also

- *Declaring data types*

Replacing arrays with vectors

Arrays are ideal for storing collections of data. However, when performance is critical, the `Array` class may not always be the most appropriate choice.

In this recipe, you will learn how to use the `Vector` class in place of the `Array` class.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe4\recipe.fla` into Flash Professional.

How to do it...

Make changes to your document class by performing the following steps:

1. Take a look at the class' constructor. It contains the following ActionScript:

   ```
   const SIZE:int = 1000000;
   var collection:Array = [];
   for(var i:int = 0; i < SIZE; i++)
   {
     collection[i] = 0;
   }
   ```

   The code above creates an array and populates it with a million elements.

2. The code can be optimized slightly by declaring the array's size ahead of time. Do this by making the following change:

   ```
   const SIZE:int = 1000000;
   var collection:Array = new Array(SIZE);
   for(var i:int = 0; i < SIZE; i++)
   {
     collection[i] = 0;
   }
   ```

3. Things, however, can be improved further by using the `Vector` class. Change your constructor to use the `Vector` class with an integer base type:

   ```
   const SIZE:int = 1000000;
   var collection:Vector.<int> = new Vector.<int>();
   for(var i:int = 0; i < SIZE; i++)
   {
     collection[i] = 0;
   }
   ```

4. Just like an array, a vector can be optimized by specifying its intended length. If the vector's size is fixed, then declare this too for an additional performance gain. Make these final changes to your code:

   ```
   const SIZE:int = 1000000;
   var collection:Vector.<int> = new Vector.<int>(SIZE, true);
   for(var i:int = 0; i < SIZE; i++)
   {
     collection[i] = 0;
   }
   ```

5. Save the document class.

How it works...

A vector is an array whose elements are all of the same data type. It provides faster read and write access than the `Array` class.

The size of both an array and a vector can be declared ahead of time. This ensures that the memory required is allocated up-front, which can lead to performance gains.

To understand why, consider either a `Vector` or `Array` that is created without its length being explicitly declared. When this is the case, some memory is allocated in anticipation of additional elements being added. As the array or vector grows in size, that allocation will become exhausted, forcing an additional allocation. However, these memory allocations can impact performance. This is particularly true of large arrays or vectors where many allocations are likely to take place as more elements are added. A single allocation during instantiation will always be faster than multiple smaller allocations.

An array's initial size can be declared by passing the required size to the constructor:

```
var collection:Array = new Array(100000);
```

Similarly, a vector's initial size can be passed as its constructor's first parameter. Unlike an array, a vector can also be fixed-length, meaning its size can't change. This can be specified by passing `true` as the constructor's second parameter. Here is an example:

```
var collection:Vector.<int> = new Vector.<int>(100000, true);
```

While specifying an array's size helps performance, the largest gains can be had by using the `Vector` class instead.

Since all elements of this recipe's array were integers, it was an ideal candidate to be replaced with a vector. The data type selected for the vector is known as its base type, and is specified using postfix parameter syntax:

```
var collection:Vector.<int> = new Vector.<int>(100000, true);
```

Any primitive built-in class or custom class can be specified as the base type.

Additionally, the `Vector` class provides the same API as the `Array` class and the square bracket operator (`[]`) can still be used to set or retrieve elements. So using a vector in place of an array should be relatively straightforward.

If you are aiming for maximum performance, then use vectors in your iOS projects. There is also the added benefit of type safety that comes with using the `Vector` class—the compiler will catch any attempts to insert instances that don't match the vector's base type.

Performance measurements for this recipe's optimizations can be found in Appendix B. The code used for testing can also be found in the book's accompanying code bundle at `appendix-b\arrays-v-vectors\`.

For more information and a complete list of APIs, perform a search for both `Array` and `Vector` within Adobe Community Help.

There's more...

While vectors have performance benefits, you still need to be careful when working with them.

Methods that return a new vector

Some methods of the `Vector` class create and return a new vector. Such methods should be used with caution as additional memory will be required and the request will consume additional CPU cycles.

The methods that result in a new vector being returned are `concat()`, `filter()`, `map()`, and `slice()`. The same is also true of these same four methods when working with the `Array` class.

If you must use any of these methods, then try to avoid using them within loops as the repeated memory allocations can seriously impair your application's performance.

See also

- *Recycling instances*

Recycling instances

Memory allocation is expensive, so too is garbage collection. Poor memory management within your app can impact performance.

Let us look at an example of this and see how to minimize the number of memory allocations.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe5\recipe.fla` into Flash Professional.

How to do it...

Perform the following steps:

1. The code within the class' constructor creates an array containing a thousand sprites. It then loops through the array and randomly sets the color of each sprite using a `ColorTransform` instance:

```
const SIZE:int = 1000;
var sprites:Array = new Array();
for(var i:int = 0; i < SIZE; i++)
{
  sprites[i] = new Sprite();
}

for(i = 0; i < SIZE; i++)
{
  var colTransform:ColorTransform = new ColorTransform();
  colTransform.color = Math.floor(0xFFFFFF * Math.random());
  sprites[i].transform.colorTransform = colTransform;
}
```

2. This code is extremely inefficient as a new `ColorTransform` object is created upon each iteration of the loop. Optimize this by creating the `ColorTransform` object outside the loop and re-use it:

```
const SIZE:int = 1000;
var sprites:Array = new Array();
for(var i:int = 0; i < SIZE; i++)
```

```
{
  sprites[i] = new Sprite();
}

var colTransform:ColorTransform = new ColorTransform();
for(i = 0; i < SIZE; i++)
{
  colTransform.color = Math.floor(0xFFFFFF * Math.random());
  sprites[i].transform.colorTransform = colTransform;
}
```

3. Save your modified document class.

How it works...

Creating a new object on each iteration of a loop when the same instance can be used again is extremely inefficient. Not only can it consume vast amounts of memory, but precious CPU cycles will go to waste managing that memory. First there is the time taken to allocate the memory required for each instance. Then there are the resources consumed by the garbage collector as it decides what instances are no longer being used. Finally, there is a performance hit when the previously allocated blocks of memory are released back to the OS.

Reusing the same object minimizes the amount of memory required to perform the task and reduces the load on the garbage collector. When targeting Flash on desktop, memory consumption is rarely a consideration. However, it really is a different story on mobile. Whereas desktop computers have vast amounts of memory and can also utilize virtual memory, iOS devices don't have such luxuries. Intelligent memory management is critical to the success of your app.

There's more...

What about arrays and vectors?

Reusing arrays and vectors

Arrays are particularly bad news when memory and CPU resources are a concern. This is especially true if you are dealing extensively with them throughout the lifetime of your application. Rather than creating new instances each time an array is required, consider re-using a previously created array. This will give you access to an already allocated block of memory, removing the need for the OS to allocate more. To clear an array's content before re-use, simply set its `length` property to `0`.

Similarly, where possible, `Vector` objects should be recycled too.

The Garbage Collector

The **Garbage Collector** is a background process that is responsible for reclaiming memory consumed by objects that are no longer active. Periodic checks are made and any memory that can be released back to the operating system is released.

While garbage collection frees much needed memory for your application, it can be a CPU-intensive task. To make matters worse, you have limited control over the timing of these checks meaning garbage collection could take place during performance-critical moments.

To minimize the risk of this happening, try to re-use objects rather than disposing of them. It will result in fewer memory allocations and de-allocations, which will save valuable CPU cycles and improve your application's run-time performance.

Forcing garbage collection

While you can't force garbage collection to take place, AIR 3.0 introduced the `System.pauseForGCIfCollectionImminent()` method. This static method provides a hint to the garbage collector at runtime, increasing or decreasing the likelihood that garbage collection will occur at that moment in time.

The method takes a single parameter of type `Number` between `0` and `1.0`, which is used to indicate how urgently you would like the next garbage collection cycle to take place. A value closer to `1.0` is more likely to trigger garbage collection, whereas a lower number is less likely.

Object pooling

This recipe's example code is a subset of a design pattern commonly known as **Object pooling**. Object pooling is advantageous when the instantiation of a particular object is expensive and the frequency of instantiations is high.

An object pool is a cache consisting of pre-instantiated objects. Rather than creating a new instance, one is taken from the object pool and initialized. When an instance is no longer required, it is placed back in the pool for use later.

By using an object pool, you limit the number of memory allocations and de-allocations used throughout the lifetime of your application, which in turn reduces the amount of garbage collection that may take place.

Object pools are often used in games to manage many things, including enemies, projectiles, and other game-world objects.

Using static members and methods

Occasionally variables and methods are conceptually associated with a class rather than an instance of a class. When this is the case, a memory saving can be made by declaring such variables as static. The more instantiations that are made of a class containing static fields, the greater the memory savings will become.

In this recipe, we will see how to apply the `static` keyword to a class.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe6\recipe.fla` into Flash Professional.

How to do it...

We will take the FLA's document class and apply some simple code optimizations to it:

1. Open the document class. It creates one thousand instances of a class named `Rectangle` and stores each within a vector array:

```
var shapes:Vector.<Rectangle> = new <Rectangle>[];
for(var i:int = 0; i < 1000; i++)
{
  shapes.push(new Rectangle());
}
```

> Don't confuse the `Rectangle` class used in this recipe with Flash's own `flash.geom.Rectangle` class. They are unrelated.

2. The `Rectangle` class can be found in the FLA's root folder. Open `Rectangle.as` within Flash Professional and take a look at it:

```
package {

  public class Rectangle {

    private const NAME:String = "Rectangle";
    private const SIDES:uint = 4;

    public function describe():String {
      return "A " + NAME + " has " + SIDES + " sides.";
    }
  }
}
```

3. The value of the NAME and SIDES constants clearly won't change across any instances of the class and can, therefore, be declared as static. Make the following changes:

```
package {

  public class Rectangle {

    static private const NAME:String = "Rectangle";
    static private const SIDES:uint = 4;

    public function describe():String {
      return "A " + NAME + " has " + SIDES + " sides.";
    }
  }
}
```

4. The same is also true of the describe() method. No matter which instance it is called from, the same string will be returned. Let us declare it as static too:

```
package {

  public class Rectangle {

    private const NAME:String = "Rectangle";
    private const SIDES:uint = 4;

    static public function describe():String {
      return "A " + NAME + " has " + SIDES + " sides.";
    }
  }
}
```

5. Save your modified document class.

How it works...

Static variables only occupy memory once, rather than once for each instance of the class. They are useful when storing information that is shared across every instance of a class, while static methods are useful for encapsulating functionality that doesn't affect individual instances.

For this recipe, each instance of the Rectangle class originally consumed 24 bytes. After applying the static keyword to both member variables, each instance required only 16 bytes. Overall, a saving of 8,000 bytes was made across all instances held within the array. The savings are likely to be much larger for real-world applications that make use of significantly more static variables.

You cannot access static variables or static methods through an instance of the class. For example, the following attempt will result in a compiler error:

```
var rectangle:Rectangle = shapes[0];
var description:String = rectangle.describe();
```

Instead, access must be performed through the class itself:

```
var description:String = Rectangle.describe();
```

Spend time identifying member variables and methods that are conceptually associated with a class and declare them as static. Neglecting to do so can needlessly waste memory.

There's more...

This recipe has shown how the `static` keyword can save valuable memory, but how can you actually measure memory consumption within your application?

Measuring memory consumption

The Flash Player API provides the `flash.sampler.getSize()` package-level function, which returns the number of bytes being used by a specified object.

The following example shows how to determine the size of the 65th element within an array:

```
var size:Number = getSize(shapes[64]);
```

Also, remember to first import `getSize()` before using it within a class:

```
import flash.sampler.getSize;
```

In order to use the `getSize()` function, you must deploy a debug build of your app. This is true of all classes and methods in the `flash.sampler` package.

Be careful when measuring the size of an array or vector. When an element represents an object rather than a primitive type, the call to `getSize()` will only count the memory required to store the object's reference rather than the object itself. For example, to calculate the memory consumed by the vector of `Rectangle` objects from this recipe, you may be tempted to write the following ActionScript:

```
var size:Number = getSize(shapes);
```

However, you actually need to call `getSize()` on each of the vector's elements instead:

```
var totalSize:Number = 0;
for(var i:int = 0; i<shapes.length; i++)
{
  totalSize += getSize(shapes[i]);
}
```

You can find additional detail regarding the `getSize()` function by performing a search for `flash.sampler.getSize()` from Adobe Community Help.

See also

- *AIR for iOS deployment settings*, *Chapter 2*
- *Remote debugging*, *Chapter 3*

Using callback methods

Flash's event model can be expensive to use. For each event that is dispatched, an `Event` object must be created and tracked. In many cases, a callback method can be employed instead and reduces code execution time.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe7\recipe.fla` into Flash Professional.

How to do it...

The FLA's ActionScript uses Flash's event model. We will make the required changes to use a callback instead:

1. Open the FLA's document class. Within the constructor a `Dice` object is created and a listener for its `Dice.ROLLED` event is added. You can see this in the following code:

```
package {

  import flash.display.MovieClip;
  import flash.events.Event;

  public class Main extends MovieClip {

    private var dice:Dice;

    public function Main() {
      dice = new Dice();
      dice.addEventListener(Dice.ROLLED, rolled);
      dice.roll();
    }

    private function rolled(e:Event):void {
      output.text = ("rolled a " + dice.face);
    }
  }
}
```

2. Now open the `Dice` class, which can be found in the same folder as the FLA. It contains a `roll()` method, which throws a random number between one and six, then dispatches its `ROLLED` event to any listeners. Here is the ActionScript for it:

```
package {

  import flash.events.EventDispatcher;
  import flash.events.Event;

  public class Dice extends EventDispatcher {

    static public const ROLLED:String = "rolled";

    public var face:int = 1;

    public function roll():void {
      face = Math.ceil(Math.random() * 6);
      dispatchEvent(new Event(ROLLED));
    }
  }
}
```

3. Let us make some changes to the `Dice` class. First remove the following code related to Flash's event model:

```
package {

  import flash.events.EventDispatcher;
  import flash.events.Event;

  public class Dice extends EventDispatcher {

    static public const ROLLED:String = "rolled";

    public var face:int = 1;

    public function roll():void {
      face = Math.ceil(Math.random() * 6);
      dispatchEvent(new Event(ROLLED));
    }
  }
}
```

4. Also, the `Dice` class no longer needs to extend `EventDispatcher`. Change the class definition from:

   ```
   public class Dice extends EventDispatcher {
   ```

   to:

   ```
   public class Dice {
   ```

5. Now add the following code that will allow a callback method to be used instead:

   ```
   package {

     public class Dice {

       public var face:int = 1;
       private var callback:Function;

       public function addRolledCallback(func:Function):void {
         callback = func;
       }

       public function roll():void {
         face = Math.ceil(Math.random() * 6);
         if(callback != null)
         {
           callback();
         }
       }
     }
   }
   ```

6. Save your changes.
7. With the `Dice` class complete, move back to the document class and make the following adjustments to it:

   ```
   package {

     import flash.display.MovieClip;

     public class Main extends MovieClip {

       private var dice:Dice;

       public function Main() {
         dice = new Dice();
         dice.addRolledCallback(rolled);
   ```

```
          dice.roll();
        }

        private function rolled():void {
          output.text = ("rolled a " + dice.face);
        }
      }
    }
```

8. Save the document class.

How it works...

In this recipe, we adjusted the `Dice` class to store a reference to a method from the document class. Upon calling its `roll()` method, the `Dice` object simulates being rolled and then calls the method that was passed to it. This provides an effective alternative to employing Flash's event model.

Callbacks do have their limitations. Whereas Flash's event model permits multiple listeners to be added for the `Dice.ROLLED` event, only one callback method can be registered. Of course, you could get around this limitation by adapting the `Dice` class to store multiple callbacks within an array. However, for many situations, a single callback will do fine and has significantly fewer overheads compared to other approaches.

There's more...

Other than using callbacks, there are other alternatives to Flash's native event model. Perhaps the most popular is AS3 Signals.

AS3 Signals

Signals is a lightweight messaging API by Robert Penner that has been inspired by both Signals & Slots in Qt and the event model found in C#. It is easy to use and can provide some performance benefits over Flash's native event model. AS3 Signals is open source and can be downloaded from `https://github.com/robertpenner/as3-signals`.

See also

- *Using an update loop*, *Chapter 3*
- *Taking advantage of event bubbling*

Taking advantage of event bubbling

Working with Flash's event model can negatively impact code execution speed. This is especially true when managing event listeners for a large collection of interactive objects. However, by taking advantage of event bubbling, it is possible to listen for events from a parent clip rather than each individual object.

In this recipe, we will begin with a code example that registers multiple event listeners and then work through the steps required to reduce it to just one.

Getting ready

From the book's accompanying code bundle, open `chapter15\recipe8\recipe.fla` into Flash Professional.

You will find a library symbol named `Wheel` in the FLA's library. The clip has been exported for ActionScript allowing it to be used by this recipe's example code.

How to do it...

We will optimize this recipe's ActionScript with a few simple line changes.

1. Open the document class. The class creates a grid of wheels and removes each when they are tapped by the user. To manage this, a `MOUSE_UP` event listener is registered with each individual wheel. When the wheel is tapped, it is removed from the display list and its event listener is explicitly removed.

   You can see the code for this as follows:

```
package {

  import flash.display.MovieClip;
  import flash.events.Event;
  import flash.events.MouseEvent;

  public class Main extends MovieClip {

    private var wheels:Array = [];

    public function Main() {
      const START_X:Number = 15;
      const START_Y:Number = 7;
      const GRID_W:int = 5;
      const GRID_H:int = 8;
      var wheel:Wheel;
      for(var x:int=0; x < GRID_W; x++)
      {
        for(var y:int=0; y < GRID_H; y++)
        {
          wheel = new Wheel();
          wheel.x = START_X + (wheel.width * x);
          wheel.y = START_Y + (wheel.height * y);
          wheel.addEventListener(MouseEvent.MOUSE_UP,
            removeWheel);
          addChild(wheel);
          wheels.push(wheel);
        }
      }
    }

    private function removeWheel(e:Event):void {
      var wheel:Wheel = e.target as Wheel;
      if(wheel != null)
      {
        wheel.removeEventListener(MouseEvent.MOUSE_UP,
          removeWheel);
        removeChild(wheel);
      }
    }
  }
}
```

2. This class can be re-written to use a single event listener by taking advantage of the `MOUSE_UP` event's bubbling phase. Rather than adding an event listener to each individual `Wheel` instance, we can capture each object's `MOUSE_UP` event as the event bubbles up to the stage.

   To accomplish this, first remove the existing calls to the `addEventListener()` and `removeEventListener()` methods:

```
public function Main() {
  const START_X:Number = 15;
  const START_Y:Number = 7;
  const GRID_W:int = 5;
  const GRID_H:int = 8;
  var wheel:Wheel;
  for(var x:int=0; x < GRID_W; x++)
  {
    for(var y:int=0; y < GRID_H; y++)
    {
      wheel = new Wheel();
```

```
          wheel.x = START_X + (wheel.width * x);
          wheel.y = START_Y + (wheel.height * y);
          wheel.addEventListener(MouseEvent.MOUSE_UP,
            removeWheel);
          addChild(wheel);
          wheels.push(wheel);
        }
      }
    }

    private function removeWheel(e:Event):void {
      var wheel:Wheel = e.target as Wheel;
      if(wheel != null)
      {
        wheel.removeEventListener(MouseEvent.MOUSE_UP,
          removeWheel);
        removeChild(wheel);
      }
    }
```

3. Finally, listen for the stage receiving `MOUSE_UP` events from any of the `Wheel` instances:

```
public function Main() {
  const START_X:Number = 15;
  const START_Y:Number = 7;
  const GRID_W:int = 5;
  const GRID_H:int = 8;
  var wheel:Wheel;
  for(var x:int=0; x < GRID_W; x++)
  {
    for(var y:int=0; y < GRID_H; y++)
    {
      wheel = new Wheel();
      wheel.x = START_X + (wheel.width * x);
      wheel.y = START_Y + (wheel.height * y);
      addChild(wheel);
      wheels.push(wheel);
    }
  }

  addEventListener(MouseEvent.MOUSE_UP, removeWheel);
}
```

4. Save the changes to your document class.

How it works...

ActionScript's event flow has three phases: the capture phase, the target phase, and the bubbling phase. During the bubbling phase, the event flows from the target interactive object back up the display list hierarchy. This makes it possible to listen for an event within the target object's parent and is particularly useful when listening for events from multiple objects within the same container.

For this recipe's code example, the stage acts as the parent for each of the `Wheel` instances and has an event listener added to it. As each object is pressed, its event bubbles up to the stage where its event handler is called. From the handler, it is possible to determine the target object that was pressed and remove it from the display list. And unlike the original version of the code, since a listener wasn't directly added to each wheel, there is no need to call the `removeEventListener()` method after removing a wheel from the display list.

Whereas forty individual calls to both `addEventListener()` and `removeEventListener()` methods were made during the lifetime of the un-optimized version of this recipe's code, after optimization, only a single call to `addEventListener()` was required.

Incorrect management of event listeners can lead to memory leaks—forgetting to remove listeners that are no longer required is a common mistake. Minimizing the number of listeners within your application reduces the complexity of your code and can help avoid memory leaks as well as improve performance.

There's more...

There is another optimization that can often be made.

Preventing event propagation

After handling an event, it is possible to prevent it propagating further up the display hierarchy by calling its `stopPropagation()` method. This wasn't required for this recipe's example as the event listener was registered with the stage, but it is a valuable optimization that you are likely to rely on for many of your AIR for iOS projects.

See also

- *Using callback methods*