{ Chapter 11 }

Objects

If you've gotten this far in the book, you've seen objects in several places. Instances of movie clips are the most basic type of object in Flash, and the best to learn from as you can *see* them onscreen. You've also seen several of the scripting objects (namely, Math, String, Array, Selection, and the Key object). Although you've already learned a lot about objects, there's more!

The objects introduced in this chapter are not only particularly practical, but also they all require the formal rules of objects such as instantiation. In this way, they make previously explored objects seem very forgiving in comparison, because you'll be doing things that were never required in the other objects. Luckily, these objects are well worth the additional effort.

In this chapter, you will

- Learn the rules of these formal objects
- Use the Sound object to "attach" sounds that can be manipulated using scripts
- Use the Color object to tint clips on-the-fly
- Use the Date object to perform any imaginable calculation involving calendars or time
- Use the AttachMovieClip statement to effectively drag clips from the Library using ActionScript

Formal Rules of Objects

Most of these concepts will appear familiar. For example, by now you know that objects have properties, which are basically just variables that contain data. Some properties have visual representations, such as a clip's `_alpha` property for example. Although the value for this property can be ascertained (as in `theClip._alpha`) and changed (as in `theClip._alpha=10`), some properties can be ascertained only. The set of properties for any object type is specific to the object. For instance, only the Movie Clip object has an `_alpha` property. Other objects have other properties—but they're all the same in that they contain values that can, sometimes, be modified.

In addition to properties, objects can have methods, which are functions that are applied to unique instances of an object. Methods are processes, whereas properties are just static attributes. So far, this should be a review. The concept that's a little bit new is that formal objects must be instantiated. In the case of clip instances (that is, Movie Clip objects), you simply instantiate them by dragging them from the Library. After each object has been instantiated, it has its own unique set of properties and the potential to have methods, such as `nextFrame()`, applied to them individually. The formal objects require that you instantiate them using a constructor function. All constructor functions follow this pattern:

```
new Object();
```

For this chapter we'll see the following constructor functions that instantiate different formal object types:

```
new Sound();
new Color();
new Date();
```

(By the way, instantiating a new movie clip using ActionScript during runtime uses a different technique that's discussed in the "Attach Movie Clip" section later.)

The key to remember with instantiating objects is that you must store the object in a variable, so saying `new Sound()` doesn't really do anything. However, `mySound=new Sound()` creates a new instance (in the form of the Sound object) and places it into the variable `mySound`, whose value is of the object data type. From this point forward, you can treat `mySound` like any object, referring to properties (`mySound.someProperty`) or using methods (such as `mySound.someMethod()`) in the same way you would treat a clip instance.

Until you build your own objects, that's about all there is to it. The trick is that you must instantiate an object and place it in a variable before you can start doing stuff with it or doing stuff to it. Now we can see the nitty-gritty details of four objects.

Attach Sound

This section probably could be called "The Sound Object." However, if there's one step that's easy to forget when using the Sound object it's the attach sound step, so maybe the section title will help you remember. Here's the process you take in order to use the Sound object.

Sound Object Basics

The idea is that by using ActionScript, you will effectively drag a sound out of the Library and start using it in your movie. Items in the Library that aren't used anywhere in your movie normally don't export when you publish your movie (which is a good thing considering that unnecessary sounds will especially add to the filesize). After you import the sound you intend to "attach" into the Library, you'll need to override the no-export feature by setting the Linkage in the Library's option menu (as in Figure 11.1).

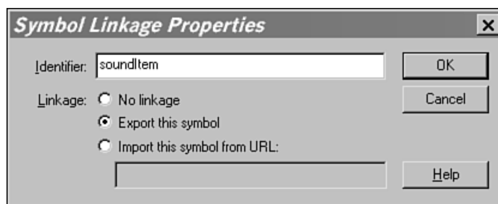

Figure 11.1 *A sound item is set to export (and given an identifier name) through the Library's Linkage option.*

All you need to do is select Export This Symbol and give it a unique identifier name (I'll use `"soundItem"` for my examples). Although we won't do it here, you can also Import This Symbol from URL but that requires you to build a shared library (a subject covered in my other book, *Sams Teach Yourself Flash 5 in 24 Hours*). In addition to causing your exported movie's filesize to grow, this sound will now download before subsequent frames load.

Now that we have a sound identified, we can start coding. You first instantiate a sound object and place it in a variable (I'm using `mySound`):

```
mySound=new Sound();
```

Before you can start using the Sound object methods shown in Figure 11.2, you need to attach the sound to this object.

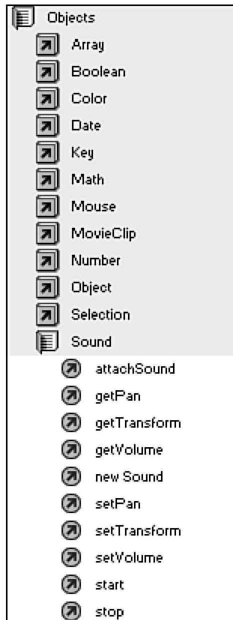

Figure 11.2 *The list of Sound object methods is short but they're powerful.*

The `attachSound` method lets you specify which sound (in the Library) you want to associate with this object.

```
mySound.attachSound("soundItem");
```

Notice that `"soundItem"` matches the name we gave the imported sound in the linkage settings. At this point, we can now start playing with the other methods. You'll likely want to start the sound so that you can hear all the changes you might make to the sound later.

```
mySound.start();
```

This will start the sound playing from the beginning of the sound file. Two optional parameters are included in this method. If you want to cut in and start

the sound (not at the beginning), you can specify the number of seconds into the sound that you want to begin. That is, `mySound.start(10)` will start the sound 10 seconds in from the start. (Of course you shouldn't use this to skip past silence at the beginning of your sound: Any silence should have been removed before import as it adds unnecessarily to the filesize.) The second optional parameter lets you specify how many times the sound should loop. To make a sound play almost continuously, use `mySound.start(0,9999999)`. Notice that you still need something in the first parameter that specifies the delay until start time in order to use the second parameter.

The opposite of `start()` is obviously `stop()`. Use `mySound.stop()` to stop the sound. This is not "pause" in that if you later use `mySound.start()`, it will start over from the beginning. There are only a few other methods (shown in Figure 11.2), so let's look at them all.

Advanced Sound Controls

Although starting or stopping a sound isn't really fancy, you'll likely need to at least start a sound before using the other methods. While a sound is playing you can easily adjust its volume using `mySound.setVolume(`*`level`*`)` where "level" is an integer between `0` and `100`. Effectively this is a percent of the volume the user has her computer system and speakers set to. That is, `mySound.setVolume(100)` will play the sound at the full 100% of the user's computer settings. Conversely, `mySound.setVolume(0)` will make the sound silent. Keep in mind that `setVolume(0)` is different from `stop()` because the sound continues to play but at a volume of `0`. The default sound level is `100` and then you can lower it using `setVolume()`. If you ever need to ascertain the current level, just use `getVolume()`.

In addition to changing the volume, you can use pan to affect the balance between the left and right channels. Similar to the way a camera can pan left and right, you can cause the sound to seem to originate from the left or right. The `setPan()` method accepts a parameter ranging from `-100` (to pan all the way to the left) to `100` (to pan right). When the sound is sent to both speakers equally, the pan is `0`. So, when a sound is playing, you can use `mySound.setPan(-50)` and it will sound as if your audio has moved to the left. You can actually set the pan (as you can set the volume) even when a sound isn't playing, but you'll always need a Sound object on which to use the `setPan()` method. If you ever need to ascertain the current pan use `getPan()`.

Finally, the last Sound object method is called `setTransform()` (and its sister `getTransform()`). On the surface, this method appears very similar to `setPan()` because it controls how much sound is going to each channel. But it's actually a combination of setting the volume, setting the pan, and exactly which portion of the audio goes to each speaker. There are four factors that you specify when using the `setTransform()` method: how much left channel sound you want going to the left speaker (referred to as `ll` and ranging from `-100` to `100`); how much left channel is going to the right speaker (`lr`); how much right is going to the right speaker (`rr`); and how much right is going to the left speaker (`rl`). Through this method you have very fine control. By the way, all this changing of volume and pan actually overrides settings previously made through the `setVolume()` and `setPan()` methods.

Now for the funky part. Specifying the four settings (`ll`, `lr`, `rr`, and `rl`) would probably be easiest if you simply provided four parameters when invoking the `setTransform()` method. But it doesn't work that way. Instead, `setTranform()` accepts a single parameter in the form of *another* object that has four properties. The process involves first creating a generic object in a variable, setting the four properties (`ll`, `lr`, `rr`, and `rl`), and finally passing that variable (data type "object") as the parameter when calling `setTransform()`. Here's how you might do it:

```
transObj=new Object();
transObj.ll=100;
transObj.lr=0;
transObj.rr=100;
transObj.rl=0;
mySound.setTransform(transObj);
```

This script effectively sets the balance equal (left going to left and right going to right are both 100).

This assumes that `mySound` is already instantiated (and playing if you want to hear anything). After you have the variable (`transObj` in this case) that contains an object, you can change any of its four properties and then invoke the last line (`mySound.setTransform(transObj)`) to hear that change. Assuming that the `transObj` exists, you can send all the left channel's audio to the right speaker (and vice versa) by using the following code:

```
transObj.ll=0;
transObj.lr=100;
transObj.rr=0;
```

```
transObj.rl=100;
mySound.setTransform(transObj);
```

To make a stereo sound play as if it were mono, use this code:

```
transObj.ll=50;
transObj.lr=50;
transObj.rr=50;
transObj.rl=50;
mySound.setTransform(transObj);
```

Translated, this code says send half the left channel's sound to the left channel, and the other half to the right. Then send half the right channel's sound to the right and the other half to the left. The result is all sounds are evenly distributed to both speakers and it sounds mono.

Finally, if you need to ascertain the current transform, use `getTransform()`. The only tricky thing is that this returns another object. If you want to then specifically target one of the four properties, you can by using the dot syntax techniques of which you're so familiar. For example, to find out what percent of the left channel is going to the left speaker, use `mySound.getTransform().ll`. If you don't want to keep calling the `getTransform()` method, you can use code such as the following:

```
curTrans=mySound.getTransform();
trace("Left speaker is playing "+ curTrans.ll + "% of the left channel");
trace("Right speaker is playing "+ curTrans.lr + "% of the left channel");
trace("Right speaker is playing "+ curTrans.rr + "% of the right
channel");
trace("Left speaker is playing "+ curTrans.rl + "% of the right channel");
```

Controlling Multiple Sounds

I left out an optional parameter when first introducing the Sound object constructor function (`new Sound()`). Think of the parameter as the way to attach a sound to an instance. Then that instance and attached sound is independently controllable just like any other property of that clip. If you provide a reference to a movie clip as the parameter, the sound will be independently controllable. Otherwise all sound objects' volume will be the same. For example, here's how you can start playing two sounds and then control their respective volume levels:

```
sound1=new Sound(clip1);
sound1.attachSound("music");
```

```
sound1.start();
sound2=new Sound(clip2);
sound2.attachSound("narration");
sound2.start();
sound1.setVolume(50);
sound2.setVolume(80);
```

You'll need two clips onstage (`clip1` and `clip2`); two sounds in the Library with linkage set and identifiers (`"music"` and `"narration"`). When the sounds start, you'll hear their respective sounds change when calling `sound1.setVolume(`*`toWhat`*`)` and `sound2.setVolume(`*`toWhat`*`)`. It's weird because you'd think by having the two sound objects stored in two separate variables (`sound1` and `sound2`) you'd have independent control. Just remember, though, you need to attach the sound to a specific clip instance (by providing the clip as a parameter) to have such control. Lastly, variables (as always) are indeed part of the timeline where they're created (so you'll need to apply all that you know about targeting if you want to refer to them from other timelines). But interestingly, including a clip reference in the `new Sound()` constructor has no impact on targeting (so you don't need to worry about it).

The Sound object is pretty awesome. Unfortunately, you can't ascertain the total length of a sound or determine the current position in a sound while it plays. However, in conjunction with the `getTimer()` function, you can get pretty close. That is, if you know how long a sound is (because you imported it) and you know when a sound started (because you started it), you can store the start time in a variable (such as `startTime=getTimer()`) when you start the sound and then calculate the elapsed time in milliseconds any time by using the expression `getTimer()-startTime`. If you know the sound is 10 seconds long (10,000 milliseconds) and you find that `getTimer()-startTime` is greater than `10000`, then you know the sound has expired.

To use the Sound object, you just need to remember these steps:

1. Import a sound and set its linkage to export. Also, give the sound a unique identifier.
2. Instantiate the Sound object and store it in a variable by using the "new" constructor: `mySound=new Sound()`.

3. Attach a sound by referring to the identifier name given in step 1: `mySound.attachSound("identifier")`.
4. Start the sound and then use any of the other methods as you wish: `mySound.start()`.
5. Finally, when you're sure that you won't need the sound anymore, you can delete the variable containing the object: `delete mySound`. Although I don't believe a few unused sound objects will bring your movie's performance to a crawl, just as any variables, there's no reason to have more than you're using. (By the way, be sure to `stop()` the sound before you delete the variable, or you'll lose control of the sound.)

Color

Through scripting, you can use the Color object to apply color effects on clip instances the same way you can manually use the Effect panel. The process is analogous to using the Sound object. The Color object requires that you first instantiate an object through `new Color("clipToTint")` in which `"clipToTint"` is the clip you want to affect, and then use one of the two methods—`setRGB()` and `setTransform()`—to cause the clip to change. It really is that simple. It's just when you want to perform elaborate effects, there are additional details—as you'll see.

Simple Coloring

Here's the simple version of coloring a clip using the Color object. First, instantiate the Color object *and* specify a target clip:

```
myColor=new Color("theClip");
```

The variable `myColor` now contains the object, so the clip instance named `theClip` will be affected when we do the next step. (Notice that even though the clip is referenced between quotes, you can still use target paths as long as you remember the quotes.) At this point, you can color the clip using the `setRGB()` method. To tint it pure red, use

```
myColor.setRGB(0xff0000);
```

For green,

```
myColor.setRGB(0x00ff00);
```

Notice that the parameter used for the `setRGB()` method is in the form of a hexadecimal color reference. The first two characters `0x` act as a warning to Flash that what follows is in the hexadecimal format. So that's it! As long as you know the hex value for the color you intend to use, this works great. By the way if you want to learn more about hexadecimal color references, the easiest way is by exploring Flash's Mixer panel shown in Figure 11.3.

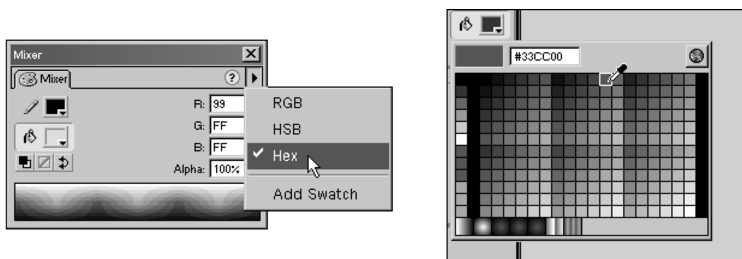

Figure 11.3 *You can change Flash's Mixer panel (left) to Hex or simply view Hex values any time you select a color swatch (right).*

Using RGB Values

In addition to providing a hex value (after `0x`) as the `setRGB()` method's parameter, you can provide a number between 0 and 16,777,215. The following three paragraphs include a detailed explanation of how you can specify colors in an intuitive (RGB) manner. 24-bit color includes 8 bits for each of the three colors red, green, and blue. This means that there are 256 shades for each color (0-255) because each binary digit is either "on" or "off." Eight binary digits—1, 2, 4, 8, 16, 32, 64, 128—all "on" adds up to 255. If they're all off, it adds up to zero. The highest number you can represent with twenty-four binary digits (three 8-bit colors) is 16,777,215.

An interesting method is used to relate 16 million different values to three colors. Blue always gets the first 8 bits (1 through 8 bits or 0-255). A value of 0 is no blue, and 255 is 100% blue. Something such as 128 is only 50% blue. To add green to the equation, 8 bits are still used, but they start at 9 and go through 16. So instead of ranging from 0-255 in one-step increments, green is defined with numbers between 256 and 65,280, which is 256 steps of 256 each. So every notch of green is 256; 256 is one notch of green, 512 is two notches of green, and so on. A number such as 522 is two notches of green and 10 notches of blue.

The way to see the breakdown of blue and green is to first extract the round 256 increments (256 goes into 522 twice; then the left over 10 is used for blue). Think about if blue went from 0-99 (in steps of 1) and green went from 100 to 1000 (in steps of 100). Because you extract the largest steps first and then the remainder, a number such as 600 would be a shade of green 6 units deep (and no blue), but a number such as 630 would be 6 units green and 30 units of blue. It actually works just like this except that instead of being based on 1s and 100s, it's based on 8 bit and 16 bit. Of course, red gets the last 8 bits, which means that it steps from 65,280 to 16,711,680 in 256 steps of 65,536 each. All this means is that it's next to impossible to intuitively specify colors using RGB—but we'll find a way.

The reason the previous concept is so difficult to understand is that we like to think of digits going from 0-9 (that is, in base 10). In our base-10 system, the far right digit is for "ones" (0-9), the second digit is for "hundreds" (0-9 again, but representing how many "hundreds"), and so on. Hexadecimal values do it in three pairs of characters RRBBGG. For example, the first two characters "RR" represents a number between 0-255 for red. The three 256-shade values for R, G, B are in the 24-bit system; they're just hard to derive. If you think in binary, though, it's probably easiest. Using 8 digits (for 8 bits), you can represent any number from 0-255. For example, 00000001 is 1, 00000010 is 2, and 00000011 is three. Each position in the 8-digit number represents a bit. To read the previous binary numbers, consider the far right digit as the "ones" (0-1), the second digit as the "twos" (0-1 representing how many "twos"), the third digit is for the "fours", and so on. Therefore you can count (in binary) 001, 010, 011, 100, 101, 111. Check it out... 1, 2, 3, 4, 5, 6 in binary!

For a 24-bit color, you need only to have 24 binary digits. The eight at the far right represent 0-255 for blue, the middle eight represent 0-255 for green, and the leftmost eight digits represent 0-255 for red (see Figure 11.4).

Figure 11.4 *A binary representation of a 24-bit number includes eight digits for each color.*

Finally, I can show you a quick way to convert RGB values (of 0-255 each) into binary at the same time that they can be used in the `setRGB()` method. That is, how do you turn `r=255`, `g=255`, and `b=255` (which is white) into a binary series of 24 ones or zeroes (that can, in turn, be used as the parameter passed when invoking `setRGB()`)? Assuming that `r`, `g`, and `b` are variables containing a number between `0` and `255`, you can use a bitwise shift operator to specify how many digits to the left you want the binary number to shift. That is, `5<<8` takes the binary version of `5` (`101`) and shifts it eight spots to the left (`10100000000`—that's `101` with eight zeros). This is exactly how to shift the value for green up eight places (or `g<<8`). Red needs to be shifted 16 places, so `r<<16` is used. Finally, the combined form looks as follows:

```
myColor.setRGB(r<<16 | g<<8 | b);
```

Notice that `b` (the value for blue) doesn't need to be shifted. The result of the entire expression in the parentheses is a binary number representing RGB by using eight digits for each color. In practice, you just need to make sure that your values for `r`, `g`, and `b` are between `0` and `255`; then simply use the previous method call as is.

Using the Color Transform Method

Naturally, you probably aren't satisfied with only 16 million different possible colors—you probably want to change the alpha of a color too. After all, I said you can use scripting to achieve the same results that the Effect panel can—and just look at all the things you can do with the Advanced option in the Effect panel in Figure 11.5. The `setTransform()` method allows you to modify any clip that has been associated with the color object in the same way the Advanced option of the Effect panel does. Of course, you could always just use the familiar `theClip._alpha=70` if you ever need to change the alpha of a tinted clip, but `setTransform()` can do even more than that.

Actually, if you understand the interface of the Advanced option of the Effect panel (which is easiest when applying an effect to a clip containing a raster graphic), you'll better understand how to use `setTransform()`. Just like using the `setTransform()` on the Sound object, you'll need to pass an object as a parameter. You first create a generic object, set its properties according to the effect you want, and then pass it when invoking `setTransform()`. The generic object has eight properties that correlate directly to the eight settings in the Effect panel. Of course we're not using the panel (that is the manual way), we're doing this with

scripting—but it helps to consider these eight properties in relation to the panel (see Figure 11.6).

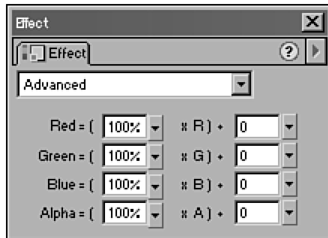

Figure 11.5 *The Advanced option in the Effect panel gives you fine control over tinting (especially with bitmaps).*

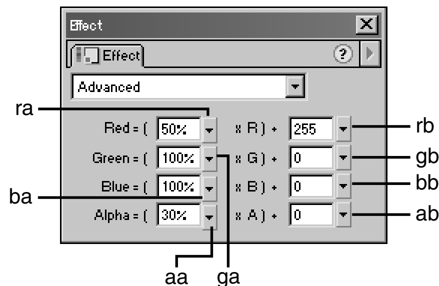

Figure 11.6 *The settings in the Advanced Effect panel are the same for the generic object passed to the* `setTransform()` *method.*

Here's a code sequence you might use to tint a clip 50% red and 30% alpha:

```
transObj=new Object()
transObj.ra = 50;
transObj.rb = 255;
transObj.ga = 100;
transObj.gb = 0;
transObj.ba = 100;
transObj.bb = 0;
transObj.aa = 30;
transObj.ab = 0;
myColor=new Color("theClip");
myColor.setTransform(transObj);
```

I see `setTransform()` as having two main benefits. You can change alpha of a clip and you can control subtle color shifts that are most apparent when the clip being colored contains a raster graphic (such as .bmp or .jpg). The fact that there are other ways to control alpha makes me think that the only real value for `setTransform()` is when manipulating raster graphics. The kinds of effects you can make are pretty cool, though.

Before we move on, let me just mention the two other methods `getRGB()` and `getTransform()`. The `getRGB()` method returns (in base 10) the last color value used on the object referenced. That is, `myColor.getRGB()` will return the color for the clip associated with the object stored in the variable `myColor`: specifically, a number between 0 and 16,777,215. By the way, a handy way to translate that number to binary is by using the `toString()` method but by providing a parameter. That is, `myColor.getRGB().toString(2)` will return (in the form of a string) the color value represented in binary. You can even use `toString(16)` to convert the number being operated on to hexadecimal. Check it out by including a Dynamic Text field onstage containing a variable value and use: `value=myColor.getRGB().toString(16)`. Finally, realize that the value returned when you use `getTransform()` is an object with eight properties. For example, if you want to ascertain the current alpha percentage, use `myColor.getTransform().aa` because `aa` is the property containing alpha percentage.

Date

The Date object gives you an easy way to store specific dates, ascertain the current date (and time), and find out details about any date (such as its day of the week). For example, I know I was born on a Wednesday—not because I remember, but because I can check it with the Date object. Basically, I created a new instance of the Date object with my birthday as the initial value. Then I used a method that returns the day of the week. Another interesting application is to repeatedly reassign a variable a new Date object (and use the current date and time for the initial value) and then you can display all the details of the current time (using a clock or calendar). It's even possible to accurately find the difference (in number of days) between two dates, and you don't need to know which are leap years or how many days any particular month "hath." (You know—"Thirty days hath September…")

Instantiating a Date

Similar to the Color and Sound objects, you always start by instantiating the Date object and then you can use methods on it. The variable you use to hold a Date object contains a snap shot of a moment in time. That is, a variable that contains the Date data type is only holding one moment in time. When you create an instance of the Date object, you can specify that moment (year, month, hour, second, and even millisecond if you want); or if you don't specify any date, you're given a date that matches the setting of your user's computer clock. Here's the form to create an instance with the current time:

```
now=new Date();
```

The variable `now` contains a Date object with the current time. You can provide up to seven optional parameters (to specify year, month, day, hour, minute, second, and millisecond). For example, this is how you create an instance that contains the U.S. Independence Day (July 4, 1776):

```
indyDay=new Date(1776, 6, 4);
```

That is, the year 1776, the month July (counting January as 0, February as 1), the fourth date in the month (which—surprisingly—starts counting with 1). I left out some optional parameters: hour, which counts from 0 (midnight) to 23 (11 p.m.); minutes (0-59 for every hour); seconds (0-59 per minute); and, milliseconds (of which there are 1,000 per second). Because the seven parameters are optional, you can leave them off if you want (though the order is important with the first parameter always referring to year, the second to month, and so on).

Manipulating Dates

After you've created a variable that holds your Date object, you can manipulate and view it through the various methods. Although quite a few methods are available (see Figure 11.7), there are only two general types—methods that "get" information from the date and methods that change or "set" elements within dates. Let's walk through some operations to get a handle on both types of methods.

Figure 11.7 *Although there are many methods for the Date object, they fall into two general categories—those that get values and those that set values.*

Getting Information from Dates

Several methods "get" specific information from a date. For example, the `getDay()` method returns the day of the week. However, because it returns a number between `0` (for Sunday) and `6` (for Saturday), you might first create an array with all the days of the week:

```
dayNames=["Sunday","Monday","Tuesday","Wednesday","Thursday","Friday",
"Saturday"];
```

Then you can easily determine the day of the week that the U.S. constitution was signed:

```
trace("Signatures were made on a "+dayNames[indyDay.getDay()]);
```

Because `indyDay.getDay()` returns a `4`, the expression `dayNames[4]` would return `"Thursday"`. It's almost as though it doesn't matter that `getDay()` starts counting with Sunday as `0` because when grabbing data from an array, we count the same way. (This isn't to say that it will never mess you up.)

Other methods are similar to `getDay()` such as `getYear()` (and its better half `getFullYear()`), `getMonth()`, and `getDate()` (which returns the number of the day in the month). It's unlikely that you'd really *need* these to ascertain the year, month, or date for a Date object that you created by specifying the date. However, they can be particularly useful when you're not sure of the date. For example, let's say that you want your Flash movie to display information about the current date in a Dynamic Text field. You can start with

```
now=new Date();
```

Then, if your text field contains a variable called `message`, you can use the following code:

```
monthNum=now.getMonth()+1;
dateNum=now.getDate();
yearNum=now.getFullYear();
message=monthNum+"/"+dateNum+"/"+yearNum;
```

By the way, `getYear()` returns the number of years since 1900 (so if you do a `getYear()` on a date in the year 2001, you'll get `101`). The method `getFullYear()` returns a 4-digit number (which naturally renders your Flash movie non-Y10K-compliant—but I wouldn't worry about it).

If you want to display the date in a format that's a little more wordy than 3/31/2001 (as previous), you can use a quick-and-dirty technique involving the `toString()` method. When used on a Date object, `toString()` returns the full date and time in the form:

Sat Mar 31 17:03:57 GMT-0800 2001

Although this is kind of nice, if you want something more readable, you could use a function such as this one:

```
function getNiceDate(whatDate){
  var dayNames=["Sunday",
  ➥"Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"];
  var monthNames=["January","February","March","April","May",
  ➥"June","July","August","September","October","November","December"];
  var day=dayNames[whatDate.getDay()];
  var month=monthNames[whatDate.getMonth()];
  var date=whatDate.getDate();
  var year=whatDate.getFullYear();
  return (day+" "+month+" "+date+", "+year);
}
```

Just pass an actual Date object for a parameter, and you'll get a string back that follows a more traditional form than what you get with `toString()`. That is, `trace(getNiceDate(indyDay))` will result in `"Thursday July 4, 1776"` displayed in the output window.

The data maintained in a Date object is detailed down to the millisecond. However, because instantiating a new Date object (with `new Date()`) only takes a snap shot of the current time, in order to make a clock (that doesn't appear frozen in time), you'll need to repeatedly re-instantiate the date object. The ideal place to do this is inside an enterFrame clip event. So, if you have a Dynamic Text field (associated with a variable `theTime`) in a movie clip and you attach the following code to the clip instance, you'll have a nice digital clock (as shown in Figure 11.8).

Figure 11.8 *You can easily make this digital clock display with a Dynamic Text field and in a Movie Clip.*

```
onClipEvent (enterFrame) {
  now=new Date();
  seconds=now.getSeconds();
  if (seconds<10){
    seconds="0"+seconds;
  }
```

```
  minutes=now.getMinutes();
  if (minutes<10){
    minutes="0"+minutes;
  }
  hours=now.getHours();
  amPm="AM";
  if (hours<10){
    hours="0"+hours;
  }else if (hours>12){
    amPm="PM"
    hours=hours-12;
    if (hours<10){
      hours="0"+hours;
    }
  }
  theTime=hours+":"+minutes+":"+seconds+" "+amPM;
}
```

The `now` variable is reassigned a new instance of the Date object and then used in most of the subsequent calculations. The `seconds` variable is set by applying the `getSeconds()` function to `now`. If `seconds` is less than 10 we just add a "0" in front of it so it still displays using two characters. The `minutes` variable is similar to `seconds`. In the case of `hours`, determining the actual hour is straightforward enough (`hours=now.getHours()`). I first assume it's AM (by setting `amPm` to "AM"), but if hours is not less than 10, I check if it's greater than 12...in which case I say `amPm` is "PM" and then take 12 off (that is, if it's 14:00 you subtract 12 to get 2 PM). Finally, I build the string called `theTime` that's displayed in a Dynamic Text field.

Naturally, the ugly part of this code is the error checking. That is, I go through extra work to make sure that numbers less than ten appear with a zero to their left. The important part to remember in this example is that the variable now is continually reassigned a new Date object (12 times a second if the frame rate is set to 12 fps). I don't think you'll see a performance hit from this code executing so frequently—but even if you did, it's a clock, so you'll want it to update frequently.

Setting Values in Dates

So far we've looked at using the Date object to store a moment in time and then use methods to peek inside. Although we haven't explored all these methods that "get" values returned, there's another set of methods that "set" values. These allow you to change any attribute of a date stored in a variable. For example,

if you want to take today's date and find out the month and date for a day exactly two weeks from now, you can use the `setDate()` method. The `setDate()` method will change the date in the attached object to whatever number you provide as a parameter. If you provide "today plus 14 days" (that is, `getDate()+14`), you'll find the answer. Here's the code:

```
now=new Date();
fortNight=new Date();
fortNight.setDate(now.getDate()+14);
trace("Two weeks from now is: "+ fortNight.toString());
```

Notice that I could have simply used the `setDate()` method on my original date object (`now`). That is, `now.setDate(now.getDate()+14)`. Instead of getting confused with a variable called "now" that actually contained a date in the future, I came up with another variable name (`fortNight`). But it's important that before I try setting `fortNight`'s date, I had to instantiate the variable as a Date object (in the second line of code). Simply, you can only use methods on objects. Another important point is the `setDate()` method actually changes the object being operated on. This is performing an assignment without the equal sign. Finally, the cool part about `setDate()` (and all the other "set" methods) is that other elements in the object being operated on automatically update accordingly. That is, if you "setDate" to today's date plus 40 days (`now.setDate(now.getDate()+40)`), you'll find the object's month (found through `getMonth()`) has changed. Similarly, you'll find the year changes when you "setMonth" to the current month plus 13.

There's one last method that I want to describe. When you use the method `getTime()` on any Date object, the elapsed milliseconds between January 1st 1970 and the object being operated on will be returned. This might seem like a useless piece of trivia but it might come up on a quiz show some time. It also happens to be the most direct way to determine the difference between two dates. For example, if you knew one person was born five days after January 1, 1970 and another person was born 200 days after January 1, 1970, it's simple to calculate the difference in their two ages as 195 days. It's not that you care how many days apart from January 1st 1970 each birthday is—it's just a common reference point. In the following code sample, you see that we never really take much note as to how many milliseconds have past since 1970, we just find the difference between two dates. In fact, one of the dates used occurred before the magic 1970 date.

```
birthday = new Date(1969, 1, 12);
bicentennial = new Date(1976, 6, 4);
difference = Math.abs(birthday.getTime()-bicentennial.getTime());
millisecondsPerDay = 1000*60*60*24;
difference = Math.floor(difference/millisecondsPerDay);
trace ("Birthday was "+difference+" days before or after the
bicentennial");
```

Notice that no one really cares how many milliseconds have elapsed since January 1st 1970 (or, even that `getTime()` could result in a negative number if the date being operated on was earlier). Instead of calculating whether a birthday was before or after the bicentennial, I just calculated the absolute value of the difference. Absolute value (`Math.abs()`) always returns a non-negative number. Finally, to convert milliseconds into days, I divided by `1000*60*60*24` (which is based on the fact that there are 1000 milliseconds every second, 60 seconds every minute, 60 minutes every hour, and 24 hours each day). Instead of just dividing the `difference` by `millisecondsPerDay` I use `Math.floor()` to make sure to just extract the integer portion of the number. That is, I don't want to know that it's been 2698.958333 days—2698 is plenty. Using this same basic technique, you can accurately calculate the difference between any two days.

Attach Movie Clip

To be perfectly accurate, we've already discussed the Movie Clip object. However, by using the technique that follows, you can effectively drag instances of clips on to the stage entirely through scripting. This is almost identical to the Sound object. And, just like how you have to remember the `attachSound()` step with the Sound object, you must remember the `attachMovie()` step here.

If you want to use scripting to cause a clip instance to appear onstage during runtime, you must first set the linkage for that clip (as we did for sounds) and come up with a unique identifier. Then, all you do is call the `attachMovie()` function using this form:

```
targetPath.attachMovie("identifier", "newInstanceName", depth);
```

Where `"targetPath"` is a path to where the new clip will reside (like `_root`), `"identifier"` is the name you gave the clip through its linkage, `"newInstanceName"` assigns it an instance name (as if you typed it in manually through the Instance panel), and `depth` is the level number. (Most clips are on level 0, but when loading movies you can specify higher numbers and the clips

will appear on top of others.) For example, if I have a clip who's identifier is `"box"`, I could use

```
_root.attachMovie("box", "box1",0);
```

This will place an instance of the clip in the Library whose identifier name is set to `"box"` onstage. The clip's instance name will be `box1`. The following code will position and change the `_alpha` property of the clip:

```
_root.box1._x=190;
_root.box1._y=33;
_root.box1._alpha=50;
```

Looks pretty familiar, eh? Well, to explain this any further would probably insult your intelligence. We covered all the bases of Movie Clips in Chapter 7, "The Movie Clip Object." The only trick to remember here is the identifier that's set through the Library item's Linkage. Additionally, you can't put more than one clip on the same level. If you attach a clip and specify level 0, you can't put any other clips in that same level (nor can you load movies into that same level). Also, if you want to remove a clip that's been created using the `attachMovie()` function, you can use `removeMovieClip()` which is a *method* of the clip, so the form is

```
targetPath.instanceName.removeMovieClip();
```

Notice that you apply the `removeMovieClip()` method on a clip reference, the same way that you use any method, not on the identifier. For example, to remove the clip created previously, use the following:

```
_root.box1.removeMovieClip();
```

Finally, there's another confusingly similar method called `duplicateMovieClip()`. All you need to specify is the new instance name for the clip and the level number. For example, you can duplicate the `box1` clip with

```
_root.box1.duplicateMovieClip("box2", 1 );
```

This method requires that an instance has already been instantiated (otherwise, you'd have no object to apply this method to). The good news, however, is the `duplicateMovieClip()` method doesn't require that you've previously specified the linkage and given the library item an identifier. Also, any scripts attached to the clip that's duplicated are contained in the duplicate. By the way, `removeMovieClip()` works the same way with clips created through the `duplicateMovieClip()` method.

Summary

We’ve looked at three traditional objects made for Flash—Sound, Color, and Date. In each of these, you first need to create an instance of the object (by putting it in a variable) and then you can use any of the object’s methods. The three objects introduced in this chapter are a good representation of “formal” objects. So many other objects in Flash have special conditions that let you get away without instantiating them (the Math object and String object in particular). Also, we got to see a generic object when creating the parameter for the Sound and Color object’s `setTransform()` method. Creating generic objects will be fully explored in the next chapter when we create our own custom objects. Our old friend the Movie Clip object was also touched on in this chapter when we looked at the process of “attaching” a movie clip. When you attach both a movie clip and a sound, you just have to remember to provide an identifier through the library item’s linkage option.

If you ever have trouble grasping concepts about objects in general, remember that you can always think about what makes an instance of a clip an object. It has a set of properties that can be varied from instance to instance. And just like any object, there are a host of methods that can be applied to individual instances of clips. The other objects explored in this chapter are still objects; they just are not really visible.