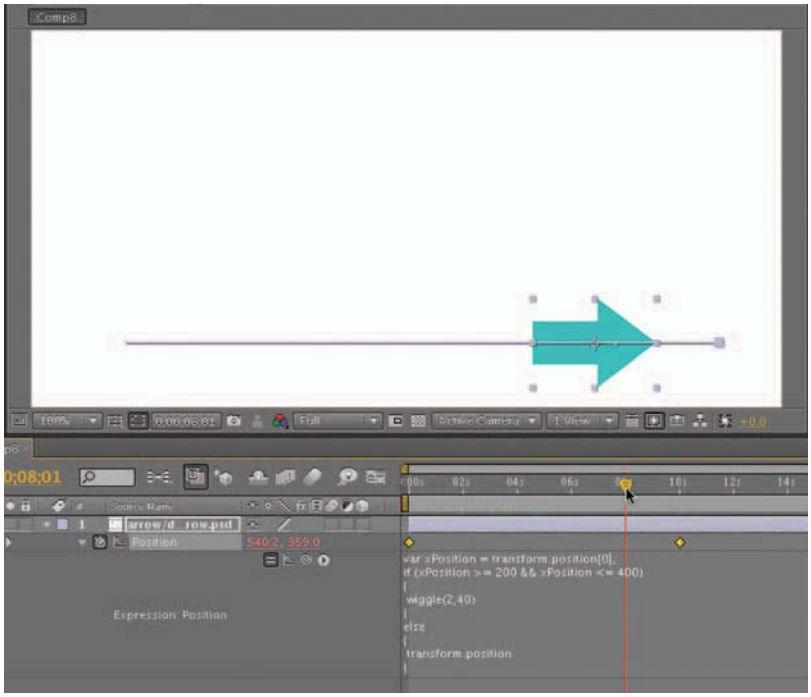
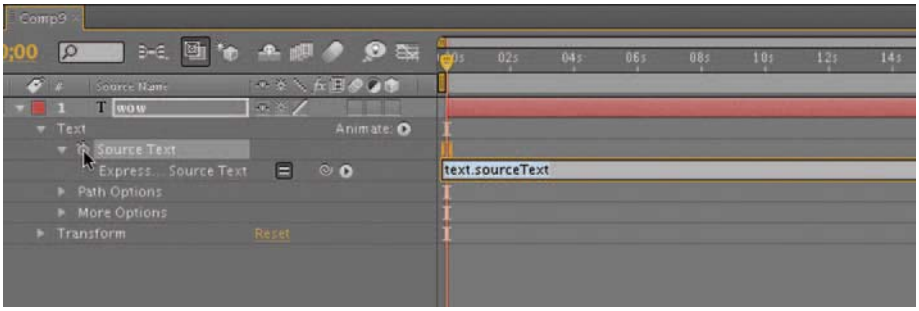


STRING MANIPULATION

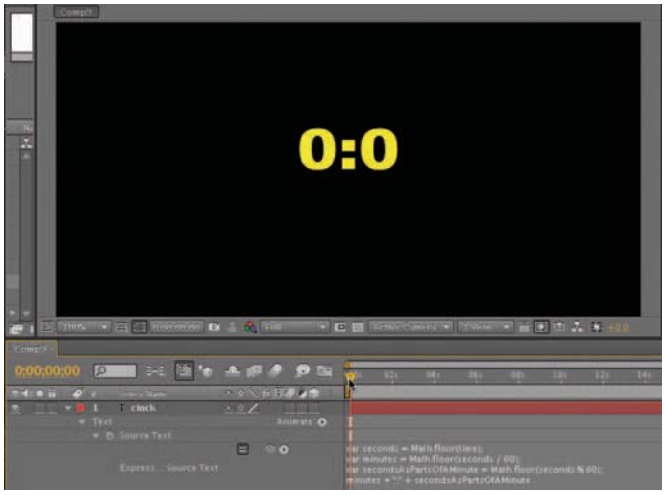
A Counter

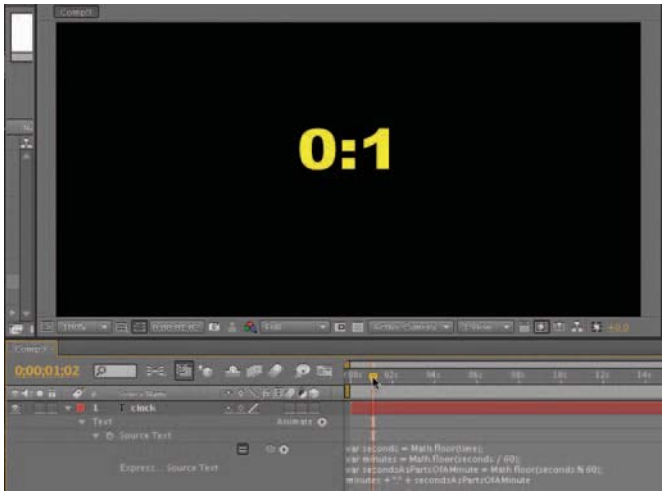
Some of the most creative Expressions manipulate Type Tool text. You always start these Expressions by twirling open a type layer, accessing the Source Text property, and Option clicking (PC: Alt clicking) its stopwatch.

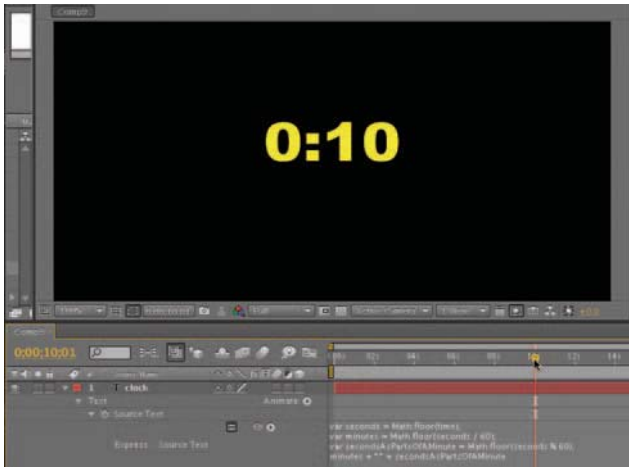


After doing that, in Chapter07.aep, Comp9, I was able to make a minute second counter, similar to the famous one on the show *24*, by adding this Expression to a text layer's Source-Text property (it doesn't matter what the text says initially):

```
var seconds = Math.floor(time);
var minutes = Math.floor(seconds/60);
var secondsAsPartsOfAMinute = Math.floor(seconds % 60);
minutes + ":" + secondsAsPartsOfAMinute
```







AE already gives you the seconds since the beginning of the Comp (the location of the CTI) in the property called time. But time is a bit too accurate for my purposes. It stores fractions of a second. So I rounded it down to the nearest second, using Math.floor().

Next, I calculated minutes. Because there are 60 seconds in a minute, the total number of minutes (at any given time) is seconds divided by 60.

The only problem now is the fact that seconds won't tick over to 0 after counting up to 59. They'll just keep going. So if I just used minutes and seconds as is, I'd get times like 4:257—4 minutes and 257 seconds. I needed to make sure that seconds always stay within the bounds of 0 and 59.

So I created a new variable called secondsAsPartsOfAMinute, and, inside it, I stored the remainder of seconds divided by 60. Remember, % gives you the remainder of a division problem. In other words, it gives you what's left over when you divide a big number into even chunks of 60. You can forget all those chunks, because they're already accounted for in minutes. What's left over is the extra seconds you need to show.

Finally, I output minutes and secondsAsPartsOfAMinute, separated by a colon, as the source text:

```
minutes + ":" + secondsAsPartsOfAMinute
```

There are a couple of problems here. First of all, the length of the text will keep changing, because the number of digits will change. For instance, 2:2 is only three characters long; 2:13 is four characters long. Anyway, who writes times like 2:2? We're taught to pad single-digits with a 0, like this: 2:02.

I fixed this problem by creating a function called addZeros:

```
function addZeros(rawNumber)
{
  var finalNumber;
  if (rawNumber < 10)
  {
    finalNumber = "0" + rawNumber;
  }
  else
  {
    finalNumber = rawNumber;
  }
  return finalNumber;
}
```

It's pretty simple: When you call it, you give it a number, like this: addZeros(5) or addZeros(16). That number is stored in the variable rawNumber.

Let's say rawNumber is 5. If it is, then the if will be true, because 5 *is* less than 10. So a new variable, finalNumber, will be set to "0" plus the 5.

You might be thinking that 0 + 5 is 5. But I'm not adding 0 and 5. I'm adding the string "0", which is not a number, to 5. Which gives me the string "05". In JavaScript, if you add a string to a number, the result is always a string. (As opposed to when you add two numbers together, which always results in a number.)

At the end of the function, I return finalNumber. So in this example, I'd be returning 05—5 in, 05 out.

If I call the function like this

```
addZeros(16)
```

then rawNumber will be set to 16, and the if will be false; 16 is not less than 10. So, the else section will run. finalNumber will be set to rawNumber. Because rawNumber is 16, finalNumber will be 16 too—16 in, 16 out.

This cool little function pads a number with a 0 only if the original number is less than 10.

(Incidentally, you may wonder what's going on with this statement:

```
var finalNumber;
```

It's perfectly legal to define a variable without assigning it a value. It's also not required. I could have just typed finalNumber the first time I actually had use for it. But I often like to define all my variables at the top of an Expression or

function. To me, it's clearer. It's like laying all your cards on the table: "Here are all the variables I'll be using, even if I'm not ready to give some of them values yet.")

In Chapter07.aep, Comp10, I rewrote my earlier counter, this time including my addZero function:

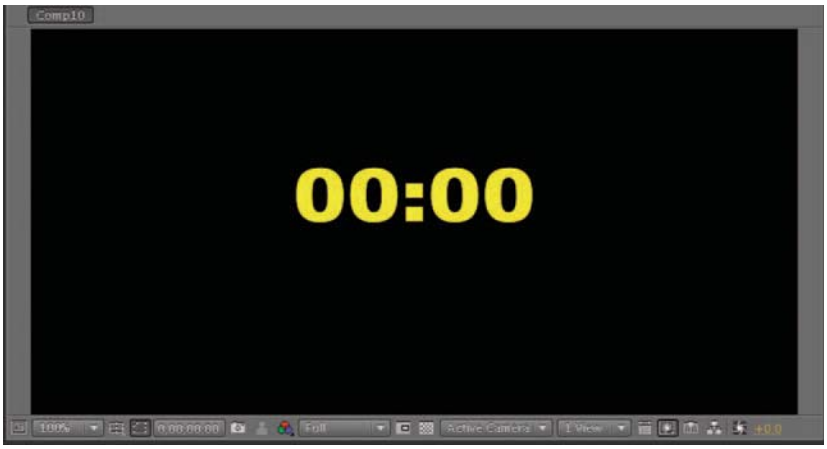
```
var seconds = Math.floor(time);
var minutes = Math.floor(seconds/60);
var secondsAsPartsOfAMinute = Math.floor(seconds % 60);
function addZeros(rawNumber)
{
  var finalNumber;
  if (rawNumber < 10)
  {
    finalNumber = "0" + rawNumber;
  }
  else
  {
    finalNumber = rawNumber;
  }
  return finalNumber;
}
addZeros(minutes) + ":" + addZeros(secondsAsPartsOfAMinute)
```

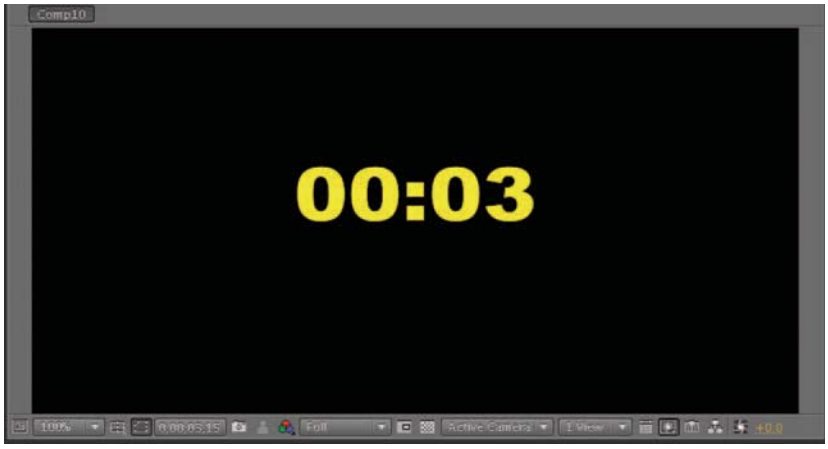

Notice that it's the same as before, except that instead of outputting

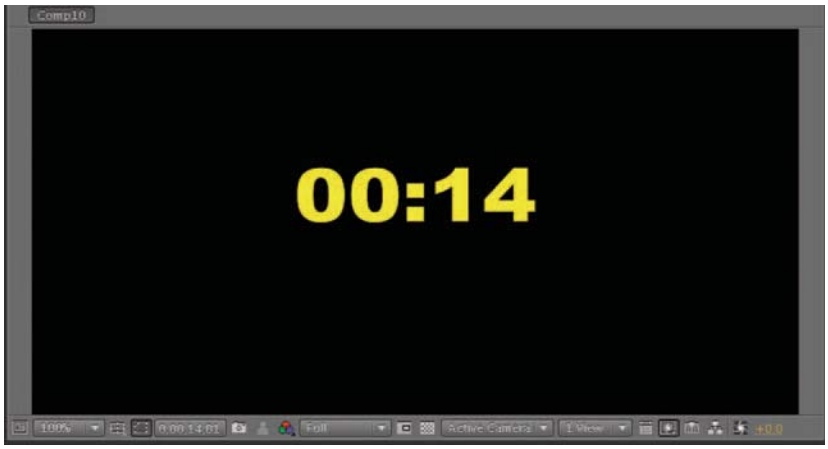
```
minutes + ":" + secondsAsPartsOfAMinute
```

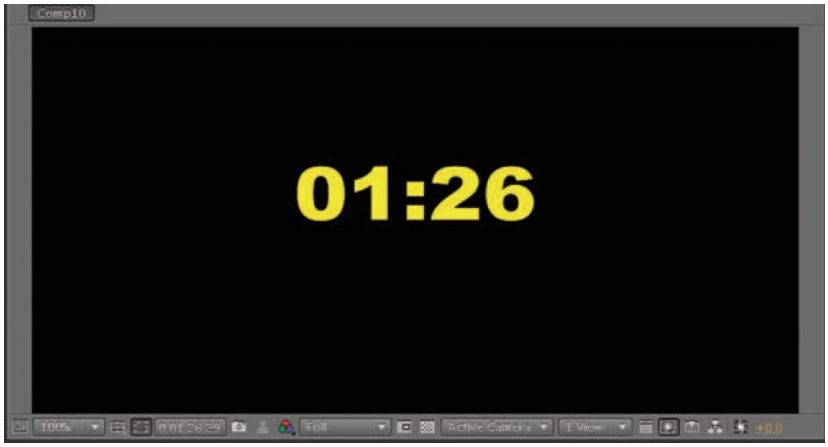
I output

```
addZeros(minutes) + ":" + addZeros(secondsAsPartsOfAMinute)
```




Comp10
00:03


Comp10
00:14


Comp10
01:26

In Chapter07.aep, Comp11, I rewrote the function to make it a bit more compact. Here's the old version:

```
function addZeros(rawNumber)
{
  var finalNumber;
  if (rawNumber < 10)
  {
    finalNumber = "0" + rawNumber;
  }
  else
  {
    finalNumber = rawNumber;
  }
  return finalNumber;
}
```

Here's the new version:

```
function addZeros(rawNumber)
{
   if (rawNumber < 10) return "0" + rawNumber;
   return "" + rawNumber;
}
```


Where's the else? Well, I eliminated it by utilizing a JavaScript trick. A function will quit running (and return a result) when the first return statement runs.

Let's say rawNumber happens to be 3. In that case, if (rawNumber < 3) is true, the immediately following return statement will run: return "0" + rawNumber;

The JavaScript trick I mentioned will make the function quit at that point, so the following statement—return "" + rawNumber;—will never run.

On the other hand, if rawNumber is 12, the if statement will be false, so the return statement after it won't run. In that case, the function will move onto the next statement—the second return statement—and *it* will run. Either one return will run or the other will run, never both.

If rawNumber is 10 or greater, the second return statement will run:

```
return "" + rawNumber
```

But why did I write it that way, instead of this way?

```
return rawNumber
```

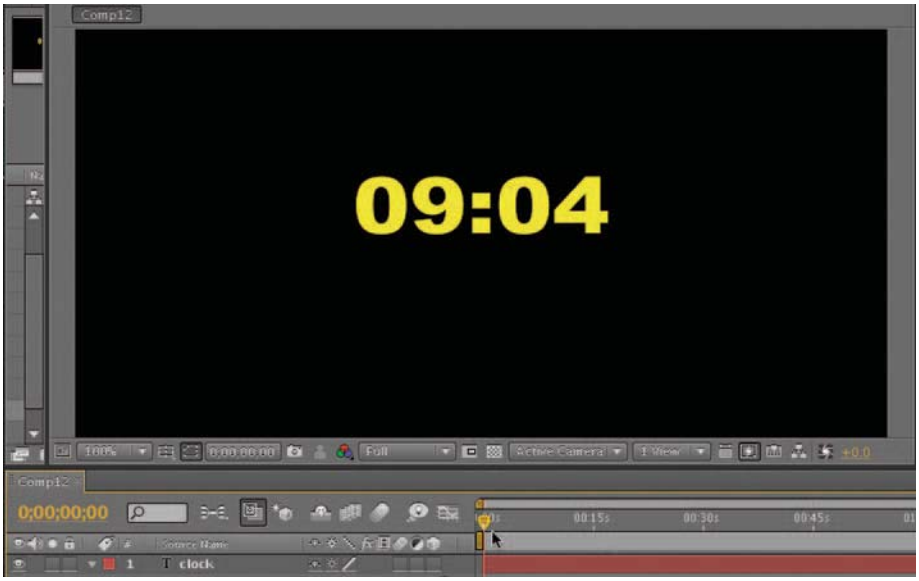
After all, if rawNumber is 12, I want to just return that 12, without any extra 0. But 12 is a number. I can't return a number, because the result is going to control a Source Text property. And the source text must be a string. So I'm adding an empty string—""—to rawNumber. That converts 12 to "12".

There's still one problem with the Expression. It's always going to start with 00:00. What if I want it to start with 09:04 and then tick up from there? In fact, that's what I make it do in my final rewrite, which is in Chapter07.aep, Comp12:

```
var startingMinute = 9;
var startingSecond = 4;

var seconds = Math.floor(time) + startingSecond;
var minutes = Math.floor(seconds/60) + startingMinute;
var secondsAsPartsOfAMinute = Math.floor(seconds % 60);
function addZeros(rawNumber)

{
    if (rawNumber < 10) return "0" + rawNumber;
    return "" + rawNumber;
}
addZeros(minutes) + ":" + addZeros(secondsAsPartsOfAMinute);
```

Words from a List

A client gave me a list of foods. He wanted one word from the list to be displayed every second-and-a-half. So I whipped up this Expression, which you can see in Chapter07.aep, Comp13:

```
var food = ["eggs","bacon","cheese","soup","cake","ice
cream","apples","ham"];
var rate = 1.5;
var i = Math.floor(time/rate);

if (i >= food.length)
{
  i = Math.floor(i % food.length);
}

food[i]
```

I retyped my client's list as an array called food:

```
var food = ["eggs","bacon","cheese","soup","cake","ice
cream","apples","ham"];
```

Remember, you can access array items via an index. So, food[0] is "eggs" and food[2] is "cheese".

Because a new food needs to display every 1.5 seconds, I needed my index to increment like this:

```
second        index
0             0
1             0
1.5           1
2             1
2.5           2
```

Here's the code I used to achieve that result:

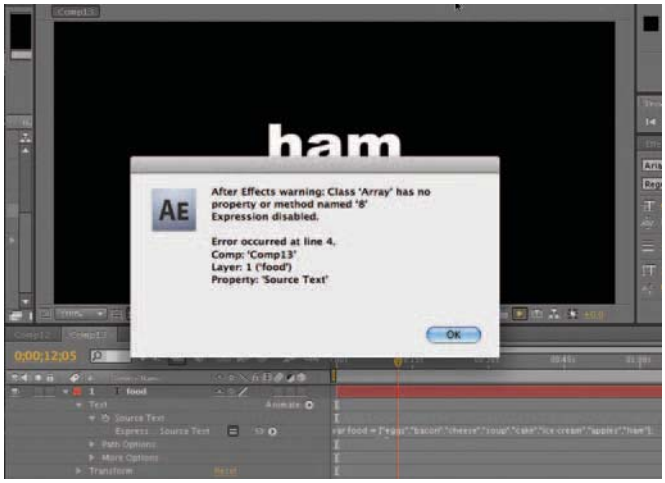
```
var rate = 1.5;
var i = Math.floor(time/rate);
```

I simply divided time by 1.5 and rounded the result down to the nearest whole number. I could have stopped there, using the result as is for the index. Had I done that, the Expression would have looked like this:

```
var food = ["eggs","bacon","cheese","soup","cake","ice
cream","apples","ham"];
var rate = 1.5;
var i = Math.floor(time/rate);

food[i]
```

The only problem is that i, the index, will eventually be bigger than the number of items in the array. There are eight items in the array, and the final one's index number is 7 (because the last, food[7], is ham). What happens when i is set to 8? Food[8] doesn't exist!



I need to deal with out-of-bounds indexes. And I need to ask myself, "What do I want to happen when the Expression reaches the last item on the list?" In this case, I decided that I wanted it to start over, displaying the first item again, and then the second, and so on. So I need i to increment like this: 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3, 4, 5, 6, 7, and so on. Here's how I kept i within those bounds:

```
if (i >= food.length)
{
  i = Math.floor(i % food.length);
}
```

If i gets bigger (or equal to) the length of the array (eight), change i so that it's equal to the remainder of itself divided by the array's length (eight). Once again, I only want what's left over.

Had I wanted the Expression to hold on the last item (ham) once it reached the end, I would have written my if statement as follows:

```
if (i >= food.length)
{
  i = food.length - 1;
}
```

food.length is 8, so food.length − 1 = 7. The final item in the array (ham) is item 7.

In Chapter07.aep, Comp14, I rewrote the Expression to make it more concise:

```
var food = ["eggs","bacon","cheese","soup","cake","ice
cream","apples","ham"];
var rate = 1.5;
var i = Math.floor(time/rate) % food.length;
food[i]
```