PROGRAMMER TO PROGRAMMER™

PROFESSIONAL JavaScript 2nd Edition

Mark Baartse, Stuart Conway, Jean-Luc David, Sing Li, Nigel McFarlane, Sean Palmer
Jon Stephens, Margie Virdell, Stephen Williams, Paul Wilton, Cliff Wootton, Jeff Yates

wrox
Programmer to Programmer

Summary of Contents

21

The BBC News Online Ticker

In this chapter, we are going to review an application of JavaScript in the creation of a news ticker. This is the one used on the front page of the BBC News Online web site at http://www.bbc.co.uk/news. Here is a screenshot showing the ticker at the top of the middle column:

This was originally designed and implemented in 1999 at a time when the installed base of browsers was somewhat different to what is available now. When it was being developed, certain design choices were made, which now need to be readdressed given the progress that has been made since then. When the ticker was first implemented, these were considered to be the important criteria:

- It should work on IE without requiring Java.
- It should work on IE across all platforms (principally Windows and Macintosh).
- It must use an existing data feed which was driving the Java applet without requiring additional editorial work to create a new data feed.
- Some of the required dynamism was known to be unavailable in Netscape 4.x, so could continue to use the Java applet.
- DOM wasn't nearly standardized and deployed enough to use.
- It must not use an additional frame.
- Netscape 6 was not available.
- The market share of Netscape was declining and the future expectation was that usage of that browser would diminish and potentially vanish altogether.

From the viewpoint of 2001, we now need to reevaluate those objectives and consider some changes to the original design criteria:

- Netscape 6 and Opera now both support `IFRAME` tags, rendering our IE-only selection mechanism less than perfect.
- The JavaScript code that was carefully written (and compromised) to address the portability issues of IE across its various platforms needs to be revised somewhat to work on Opera and Netscape 6.
- DOM standardization is much improved but regrettably, much of it is still non-portable due to the way the browsers treat whitespace and empty text nodes in the object model.

The early part of this chapter, will describe how the ticker works, its internals, and why it was implemented the way it was. It is necessary to discuss the compromises you need to make when building a real world JavaScript application like this.

At the outset, you begin with a somewhat ideal implementation that works fine on a single platform and browser. As you test the code on more platforms and browser versions, you gradually reveal more deficiencies in the browsers, meaning that you need to modify code until it works on both old and new browsers and whatever platforms that browser is supported by. In the end, you arrive at a code base that is somewhat less pretty than that which you started with. Also, there are all sorts of extra code fragments and sub-optimal coding techniques that become necessary.

That's roughly where we are with the live deployed ticker as it is on the News Online site. It works on the platforms and browsers it was originally intended for, but new browsers are available which mandate some corrective code to enable the ticker to run on them as well.

Things have moved on and usage statistics tell, us that Netscape 4.x usage is still declining, while Netscape 6 usage is very small, but increasing. It is not yet a serious challenger to Microsoft, but it has the potential to gain significant market share. Opera browser usage is also small, but increasing, indicating that some people are considering alternatives to the Microsoft browser. We have a long way to go however before IE is no longer the dominant force in browser usage.

In the latter parts of this chapter, rather than simply review the ticker application as it was designed in 1999, we'll try to bring it up to date by remodeling the basic implementation so that it works with a wider variety of contemporary browsers.

Why do it in JavaScript?

One of the popular uses of *Java* is to build animated effects that can be embedded into a web page. Many applets provide button rollovers and menu handlers, or are used for news headline scrolls and tickers.

The major problem with all of this, is that there is a certain penalty involved in deploying these applets. The problem is that for many browser and platform combinations, the time taken for the Java Virtual Machine (JVM) to start up can be a nuisance. Also, if the applet requires a later version of the JVM, then that has to be installed as well. In its worst-case scenario, the JVM start-up can completely suspend the page loading and drawing process, meaning that the user can sometimes stare at an empty or unchanging screen for 30 seconds or more.

Using frames can alleviate this, but can still be frustrating. So, the challenge is to create a news ticker using JavaScript. It needs to be dynamically data driven and start-up very quickly.

When we look at an example like this, there is a temptation to suggest that there must surely be a very simple solution. We will see though that the result is something that did evolve from a simple solution, but has become complex as a necessity. The complexity comes from discovering shortcomings in the support of dynamic effects on different versions of browsers. Different platforms also exhibit failure modes that force us to code round the problem, sometimes in a sub-optimal way.

Simplicity is possible if you can bind your target audience with code that works on a very restricted platform and browser combination. However, portability dictates that we need to go the other way and accommodate many versions of many browsers on many platforms. That clearly must add complexity given the current state of the JavaScript and DOM implementations.

The Platform Issue

Right away, we bump into some difficult problems. Any dynamic effects are not likely to be browser independent. This kind of dynamism on Netscape prior to version 6, is going to rely on layers and may not be feasible at all, (that's assuming we are going to totally disregard browsers earlier than Netscape 4). With layers, we might be able to create some dynamic effects, but that would have to take place within the main content area of the page. Constraining the boundaries of the layer to fit exactly the right location and size, were also considered to be quite a difficult problem to solve in the context of the News Online front page where the ticker was intended to be used. On those grounds, and also being aware that even in 1999 layers were already considered a deprecated feature, we decided not to attempt to provide Netscape 4.x JavaScript driven tickers. We already had the Java ticker that was working on Netscape anyway.

On IE, we had Dynamic HTML techniques available to us to replace the content of a tagged block of HTML marked-up text.

Looking at the traffic figures for the BBC News Online site, we established at the outset that Netscape 4 accounted for less than 15% of the traffic. Most of our traffic was attributable to IE. This itself was quite a shock, as it clearly indicates the extent to which Microsoft now dominates the browser marketplace and the extent to which Netscape have lost market share.

Looking again at the figures 2 years on, Netscape 6 penetration is still very small at around 0.3% (as of summer 2001). Making special provision for Netscape 6, is very hard to justify from a commercial or resource allocation point of view. In the meantime, Netscape 4 browsers are rapidly declining in use. Arguably, the Holy Grail of a single platform is mostly within reach at the expense of it all being under Microsoft's control. Even that is a vain hope, because IE is implemented quite differently on Macintosh and Windows, and will only be available on platforms that Microsoft sees commercial benefit in supporting. There is no long-term alternative other than approaching the problem from a standards-driven perspective. In 1999, that was not viable, but it is now looking more attractive.

All of this dictated the design goal of getting this ticker working on IE and basically disregarding the other browsers since they could continue to use the Java applet, albeit with the 30 second delay for the Java VM.

Insetting the Ticker Into the page

We need to find a way to embed the ticker into a page so that our target browser invokes the JavaScript version, while any others call in the Java version.

Remember, that this decision was being made in 1999 when only IE supported the `IFRAME` tag and all other browsers ignored it. Yes, we did expect that other browsers might implement it in the future and now that future has arrived, we will review this later on in the chapter, when we develop a new ticker for contemporary browsers. For now, we are looking at the live deployed ticker.

The IFRAME tag was assembled in such a way as to call in the ticker HTML document only on IE. The other browsers would ignore the IFRAME tag and see the HTML contained within it. This fragment of HTML is taken from the News Online front page:

```
<IFRAME SRC='/ticker/ticker.stm' WIDTH='315' HEIGHT='30'
   SCROLLING='no' FRAMEBORDER='0'>
   <APPLET CODE='ticker.class' CODEBASE='/java/'
          WIDTH='300' HEIGHT='50'>
      <PARAM NAME='bgcolor'     VALUE='255,255,255'></PARAM>
      <PARAM NAME='linkcolor'   VALUE='255,0,0'></PARAM>
      <PARAM NAME='textcolor'   VALUE='0,0,0'></PARAM>
      <PARAM NAME='SectionID'   VALUE='252'></PARAM>
      <PARAM NAME='LanguageID'  VALUE='3'></PARAM>
      <PARAM NAME='RegionID'    VALUE='-1'></PARAM>
      <PARAM NAME='SubRegionID' VALUE='-1'></PARAM>
   </APPLET>
</IFRAME>
```

Conveniently, IE ignores the HTML between the IFRAME and /IFRAME tags, while still implementing the IFRAME. Netscape ignores the IFRAME tags and sees the APPLET element inside. The applet parameters are not important to us here and are provided to allow the tickers to be customized for different language variants of News Online.

So, the ticker runs within a frame that is placed at the top of the page (or wherever we like).

How It Works

The workings are fairly simple:

- Test that we are running in the correct browser
- Initialize any data structures
- Construct an interval timer that plays out the ticker text
- On each cycle, pick up a new headline and play it out
- After a pre-defined number of cycles, reload the ticker page

We will now take a closer look at these steps to consider the design choices and compromises that were made.

Browser Test

During the initial page loading, several functions are declared, and a browser test forces a redirect to an alternative ticker written in Java if the browser is other than IE or older than IE 4. Here is the fragment of JavaScript code that carries out the test and redirect:

```
// --- Check for old browser and force load the applet
theBrowserVersion = parseInt(navigator.appVersion);

if (theBrowserVersion < 4)
{
   location.href = "/ticker/ticker_applet.252.htm";
}
```

Remember, that this is assumed to only be running on IE. Now that Opera and Netscape 6 support IFRAME tags, this browser test requires some modification. The smallest change we could make to the ticker script, is to test for non-IE browsers here and force the applet to load, but that's not a very nice way to solve the problem. It is far better to get the ticker to work on those other browsers instead.

Ticker Startup

Once the BODY is fully loaded, we can trigger the ticker start-up. It is important to wait until this time, because we need the BODY loading to be complete, so that we can extract the story text from the DIV blocks at the end of the page.

The ticker startup is called like this:

```
<BODY onLoad="startTicker();">
```

Here, is that startup function code:

```
// --- Only run for V4 browsers (check browser again here -
//   some old browsers won't do this inline)
   function startTicker()
   {
      theBrowserVersion = parseInt(navigator.appVersion);

      if (theBrowserVersion < 4)
      {
         location.href = "/ticker/ticker_applet.252.htm";
          return;
      }

// ------ Check and fixup incoming data block
         if(!document.body.children.incoming.children.properties)
         {
            document.all.incoming.innerHTML = "… default data …";
         }

// ------ Set up initial values - behavior
         theCharacterTimeout = 50;
         theStoryTimeout     = 5000;
         theWidgetOne        = "_";
         theWidgetTwo        = "-";

// ------ Set up initial values - content
         theStoryState       = 1;
         theItemCount        = document.all.itemcount.innerText;
         theCurrentStory     = -1;
         theCurrentLength    = 0;
         theLeadString       = "  …  ";
         theSpaceFiller      = "    …  <BR><BR><BR>";

// ------ Begin the ticker
         runTheTicker();
      }
```

Due to some browsers that support IFRAME tags not also supporting the earlier inline browser test and redirect, we also do that again here to catch them. This is mostly for the benefit of preventing IE 3 browsers from trying to run the ticker.

The next fragment of code checks that the data in the DIV blocks actually exists. This is a work around for a problem that occurred in the publishing system where an empty data feed was routed to the server. It resulted in the DIV blocks being present, but containing no data. This tests for that and reconstructs some default data if necessary. The code has been simplified to save a little space.

There are two blocks of global variable initialization. They are separated simply, because one set describes how the ticker works, while the other relates to the ticker data. The values in those variables control the ticker as follows:

Variable	Purpose
theCharacterTimeout	This is the time in milliseconds between drawing one character and the next. The value is used to determine the delay before calling the next cycle of the ticker.
theStoryTimeout	When a headline is complete, this delay in milliseconds determines how long the story dwells on the screen, before the next story begins to tick out.
theWidgetOne	The bouncing golf ball effect requires two characters to be alternately appended to the end of the ticker. This is the first.
theWidgetTwo	This is the other golf ball effect character. The two characters need to differ in their perceived shape. One needs to be elevated while the other needs to be low down. An underscore and a minus sign work quite well together. A more gross effect can be obtained with an underscore and an asterisk. You could experiment with HTML character entities as well.
theStoryState	The story state indicates whether we are currently playing out the headline text as a ticker, or whether it is completed.
theItemCount	The total number of stories available is stored here.
theCurrentStory	This determines which one of several stories is the one currently being played out.
theCurrentLength	The current ticker length enumerator variable.
theLeadString	In the News Online ticker, we must indent the text to avoid overwriting the 'Latest:' text that is part of the background. That spacer must be a series of HTML character entities that represent non-breaking-spaces. This is important, so that the mouse can enter and click on any part of the ticker.
theSpaceFiller	The as yet unfilled part of the ticker must also be filled with non-breaking space characters so that it can be clicked on.

The last line in the ticker startup function runs the first cycle of the ticker.

Main Run Loop

Each time the ticker cycles, the main run loop invokes this function:

```
// --- The basic rotate function
      function runTheTicker()
      {
         if(theStoryState == 1)
         {
            setupNextStory();
         }

         if(theCurrentLength != theStorySummary.length)
         {
            drawStory();
         }
```

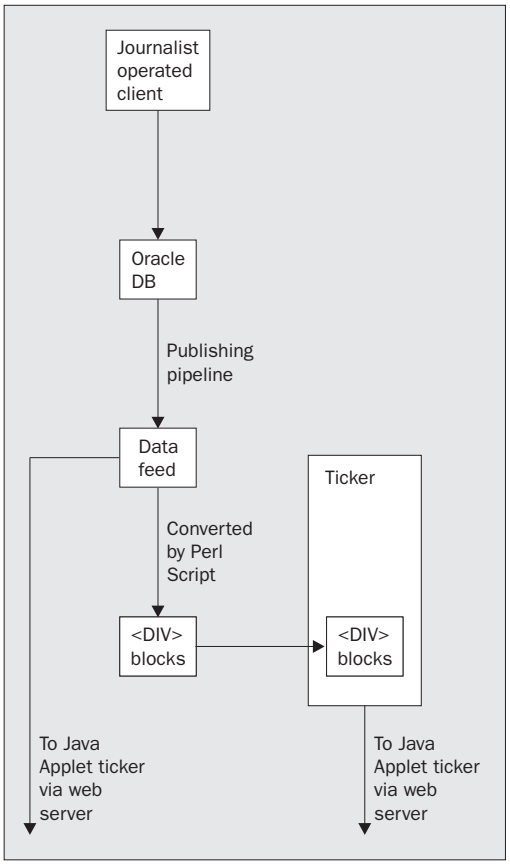
```
            else
            {
               closeOutStory();
            }
         }
```

The `StoryState` indicates whether we need to set up the next story. That involves fetching its headline text and determining the URL value and headline length.

If we are not already at the end of the headline (`currentLength = headline length`), then we draw the story, taking into account that we must append one more character than we did last time. If we are at the end of a headline, we must close out the story. This draws the headline text without the golf ball effect.

Data import

Setting up the story involves the least portable part of this application. Due to the constraints imposed on the ticker design, the textual data must be embedded within the same file as the ticker script so it is loaded as a single document. The workflow processes that involve a team of journalists entering headline data via a client on a database, and that data being published as a data feed imposed this constraint. The format and presentation of that data feed was immutable, because it is used to feed several other processes, but we were able to pass it through a filter and convert it into something we could use. Unfortunately, that something was not JavaScript, but HTML structured with `DIV` blocks. That fragment of HTML is then inserted into a template ticker file with a server-side include. This is illustrated in the following diagram, which shows the critical parts of the process that are used by the ticker:



The story array is traversed one item at a time and having obtained the story text, we construct the HTML to be placed inside the `DIV` block containing the `anchor` which the user will click on to go to the story. We use the `innerHTML` property to change the content of the `anchor` object. Each cycle round the loop, we construct a longer text until the entire string is inserted. At that time, we wait for a few seconds before indexing to the next story. The story is indexed with the global variable called `theCurrentStory` and the ticker text is sub-stringed with the global variable called `theCurrentLength`.

We'll skip the details of that traversal and extraction for a little while, because there are several alternative ways to accomplish it and we'll review them all in due course shortly. We'll also not go into great detail on the sub-string extraction and ticker drawing because we'll also cover that when we come to build our new ticker later on. That part of the ticker functionality at least should be fine. The portability issues are really all to do with extracting the story data from its container.

On the whole, this functionality is not very complex and is based on time-outs, as you might expect. The large string full of non-breaking spaces is necessary to fill the `IFRAME`. There may be a more optimal way to build that string with a `for()` loop, but all of the techniques for iteratively concatenating a single ` ` entity would in fact cause a memory leak due to the fresh instantiation of a string to contain the concatenation on each loop. Perhaps the non-breaking spaces could have been put into the HTML portion of the page and extracted, but there didn't seem to be any benefit and there was certainly no space saving there either.

The key to this is the `setTimeout()` method, which is a very useful facility. A similar and related method can be used to call a function periodically, which is the `setInterval()` method. That method would not be appropriate here, because we want to change the duration of the timeout when we have completed the drawing of a story.

There are two ways we call `setTimeout()` to invoke the next loop. The first, is when we are still intending to tick out some more text. In the `drawStory` function, we call `setTimeout()` like this:

```
setTimeout("runTheTicker()", theCharacterTimeout);
```

The other case is when we close out a story to remove the golf ball. It's very similar, but uses a different global variable to yield the longer timeout value:

```
setTimeout("runTheTicker()", theStoryTimeout);
```

Visual Effects and Presentation

To create the effect of an old-fashioned golf ball teletypewriter, the last character of the headline string is alternately replaced by an underscore (_) or a hyphen (-). You can experiment with using other characters to make the effect more obvious, but this seems effective and reasonably subtle. Using an asterisk instead of the hyphen, makes the effect much more noticeable.

Here is how two adjacent characters are painted in with the golf ball effect:

To determine which of the two characters to use, we perform a modulo 2 operation on the length of the output string to yield either a value of 1 or 0. This can then be used to select one or other of the widget characters. The code in the ticker is used inline, but can be wrapped in a function that returns one or other character. We'll use a function like this later on in our new ticker design:

```
function makeWidget()
{
   if((theCurrentLength % 2) == 1)
   {
      return theWidgetOne;
   }
   else
   {
      return theWidgetTwo;
   }
}
```

The string that is stored in the `innerHTML` of the `target` object needs to be constructed with the minimum amount of string construction/destruction possible. This is because garbage collection on some browsers does not reuse memory that is freed up by discarded strings until the page is cleared. The content of the ticker is manufactured when needed and is not stored in a string variable at all:

```
target.innerHTML = theLeadString + theStorySummary.substring(0,theCurrentLength) +
makeWidget() + theSpaceFiller;
```

Styling the Output

The appearance can be controlled with some simple CSS style settings at the top of the file. We just need to create a style for the `anchor` and define the `hover` effect. Here is the CSS style to define just those attributes:

```
<STYLE TYPE='text/css'>
   <!--
      A
      {
         font-family: Verdana, Arial, Helvetica, sans-serif;
         font-size: 11px;
         line-height: 11px;
         text-decoration: none;
         color: #333366;
         font-weight: bold;
      }

      A:hover
      {
         color: #CC3300;
      }
   -->
</STYLE>
```

Issues Regarding Screen Updates

During the development of the ticker algorithm, some important pitfalls were found. These were especially problematic with the IE browser on the Macintosh.

Initially, the headline item was composed simply of the text that was visible. This meant that the clickable text in the `A` tag did not cover the entire frame. It was constantly being replaced to create the animated effect, therefore there was some degree of object construction and destruction going on.

If at the same time the mouse was being rolled over the headline, an event was being generated to indicate `mouseOver`. This edge detection is fine, as long as the objects being monitored are static and non-moving. Destroying the object that the mouse has just entered, even though another is created, causes some problems to the event handling mechanism, which has some kind of memory handle on the object that was originally rolled over.

Since that object no longer exists, the browser rapidly becomes very confused when the `mouseOver` event is handled. IE on the Macintosh simply cannot cope with this and the whole thing crashes catastrophically.

The same object boundary problems with `mouseOver` events were in evidence when the text string "`LATEST:`" was placed in front of the link but outside the `A` tag. This is why a background graphic containing that text is placed in the `BODY` tag. Some leading non-breaking spaces are used to indent the text in the `anchor` but are included inside the `A` tag. Some additional trailing non-breaking spaces are also included in the `A` tag to ensure the clickable object covers the entire area of the frame. Even so, word wrapping comes into play and there is some uncertainty about the right hand edge of the object, though this does not seem to cause as much of a problem.

Memory Leaks and Garbage Collection

In the early trials, a great deal of string concatenation and construction/destruction took place. The visible string was assembled one character at a time on each cycle of the ticker loop. Amazingly, this could consume 50K of memory every few seconds and before long, massive amounts of memory were being used up. This is because JavaScript does not properly clean up its 'Auto Release Pool' until the page is reloaded.

The final version of the ticker uses the sub-stringing capabilities of JavaScript to slice out an increasingly longer portion of the text string and performs no persistent string concatenation. There is evidence on some platforms that a very small memory leak may occur, which is much better than the original version. It would be nice to eliminate these leaks altogether. By way of illustration, the following approach leaks massively:

```
myString = "";
for(iii=0; iii<10; iii++)
{
   myString = myString + "A";
   document.write(myString);
}
```

By contrast, this approach leaks hardly at all, although it appears to waste space by assigning content to `myString` at the outset:

```
myString = "AAAAAAAAAA";
for(iii=0; iii<10; iii++)
{
   document.write(myString.substring(0,iii));
}
```

Every few minutes, the whole ticker page can be reloaded to avoid any small memory leaks becoming a problem. This was originally done within the JavaScript code based on a count of the number of headlines that had been played out. That has now been removed, and happens courtesy of the main page being reloaded by the browser.

This reloading also has the added benefit that the ticker headline list is likely to be refreshed very soon after the master copy on the web server has been updated. It isn't a perfect synchronization but it's good enough. Usage patterns from the web server logs indicate that people don't stay on the front page for very long before going to read a story and then coming back again. If they do leave the page on display for long periods, then an automated page refresh ensures the ticker is updated periodically. There is a trade off in setting the update period too short. If many hundreds of thousands of users have the front page on display with the ticker, shortening the update time causes a measurable increase in traffic and web server loading.

These memory-leaking problems were observed to be far worse on Windows than on the Macintosh version of IE. This may be due to a variety of reasons dependant on how the underlying memory management is implemented. The Macintosh system software is known to implement a particularly effective garbage collection and memory compacting scheme. Segments of memory within the application heap are moved when necessary to collect all the free space into one contiguous block. While this happens, the memory manager also frees up any purgeable space to keep things neat and tidy.

Our Updated Ticker Design Specification

Let's outline our design criteria for the new ticker implementation:

- It should run on as many platforms as possible.
- Perhaps the code can be simpler than before.
- Passing the data to the ticker should be the most efficient technique and not be limited by the workflow constraints imposed by BBC News Online.
- Can we add any functional enhancements to the ticker?
- Can we find a better event detection mechanism that does not crash browsers with `mouseOver/mouseOut` boundary conditions?
- Can we place some HTML inside the `IFRAME` tag that avoids the need for the `APPLET` tag?
- Pose and answer the question "Does DOM standardization really help us in solving a problem like this?"

In our new implementation, lets also simplify things by removing the background graphic and see if we can streamline the main `run` loop.

Passing Data to the Ticker

Before proceeding with the coding of our new ticker design, we need to choose a means by which we shall import the story headline and URL into the scripting environment.

Passing data to a JavaScript execution context from a back end database is actually quite difficult to do in a portable manner. Probably because of the security implications, it is not possible to request a URL and access its data content as a text string within a variable. It would be useful to have a facility to do something like:

```
myVar = getURL('http://www.abc.com/mydata.txt');
```

This would be great even if there were a limitation, such as the file having to be on the same server that the script came from. It is certainly possible to do it with Java, because the applet we replaced in News Online does that very thing. However, the whole point of this JavaScript implementation is to eliminate the applet, so we can't really use a Java applet as a delegate to fetch the data for us.

Microsoft provides some non-portable solutions to this problem with IE 5, such as the download behavior or an ActiveX control for loading content. Both have the 'same server' restriction but are otherwise fully functional to load a text file into a string variable. However, neither of them work in Netscape or Opera.

There are several portable approaches, each having advantages and disadvantages:

- Hidden fields inside a form
- Textual content in a hidden frame or layer
- Building a document structure from nested `DIV` blocks
- DOM navigation of document structures
- XML data islands
- JavaScript code inserted or included into the document

We can experiment with some examples that illustrate each of these to discover their limitations.

Hidden Fields

The limiting factor with hiding fields inside a form is that there is no structure implied and it is simply a one-dimensional array of strings. Another disadvantage (although you may not consider it a problem), is that it must be embedded within the same page. You can add structure by embedding mark-up within the data, but that needs to be parsed out somehow. You could carefully hide executable JavaScript in these form fields and then use the `eval()` function to call it into your script. Here is an example that extracts some hidden JavaScript and uses that technique to insert the text into a `DIV` block at the top of the page:

```
<HTML>
<SCRIPT LANGUAGE="JavaScript">


function init()
{
   // Locate form element containing the script
   myHiddenScript = document.forms.HiddenData.Content.value;

   // Execute the hidden script data
   eval(myHiddenScript);

   // Locate the target DIV block
   myTarget = document.getElementById("TextBox");


   // Store the values extracted from the script
   myTarget.innerHTML = myData1 + "<HR>" + myData2 + "<HR>" + myData3;
}
</SCRIPT>

<BODY onLoad='init()'>
<DIV ID='TextBox'>
</DIV>
<FORM NAME='HiddenData'>
<INPUT TYPE='HIDDEN' NAME='Content'
VALUE='myData1='Hidden Text String one';
       myData2='Hidden Text String two';
       myData3='Hidden Text String three';'>
</FORM>
</BODY>
</HTML>
```

This is fairly portable. The insertion in `DIV` blocks limits how far back in browser versions you can go, but pulling things out of hidden form fields should work with very old browsers since access to forms data and the `eval()` function have been around for a long time.

Hidden Frames and Layers

This technique passes textual content in a hidden frame or layer. Layers are already risky, because they are deprecated. Hidden frames are a good solution although there is a flaw that we'll consider in a moment. At News Online, the HTML design authorities mandated that we should not use additional frames on the front page. That effectively ruled out that technique, even though it would be a fairly optimal approach. The downside, is that you need to hide the frame somewhere. The same JavaScript `eval()` trick would work though.

There's another more serious shortcoming in this technique, which is to do with the way web browsers load pages and web servers respond to simultaneous requests. The outcome of that, is that you simply cannot be sure when that frame containing the hidden data is loaded. The necessary tricks to set synchronization flags require `onLoad` handlers in the data frame and the ticker, which both call functions in the parent frameset container. That's guaranteed to be loaded, because otherwise the child frames wouldn't have been requested. You can then set semaphores to indicate loading is complete and then access the data conditionally on those semaphores being set. It's all a bit too uncertain and any connection failures and loading errors can cause even more problems.

DOM Navigation

DOM navigation of document structures has been proven to be unworkable. Both IE and Netscape 6 provide the same API to the document content. Unfortunately, they treat whitespace and empty `textNodes` differently, and so the `childNodes` collections that might have been useful for walking down the tree are of no use, because none of the index values correlate between browsers. Indeed, IE on the Macintosh and Windows platforms are even further apart than the Macintosh versions of IE and Netscape 6.

Setting those problems aside, it's quite desirable to use DOM techniques to extract content from within the same document that the `SCRIPT` tag is located.

If only the DOM implementations provided a way to examine the child nodes of an object in a consistent manner, this would be a really useful technique, because the structured `DIV` block approach is widely supported. The biggest problem with DOM is that the interstitial text between each `DIV` tag is treated differently in IE and Netscape. In fact, IE on Macintosh and Windows platforms are significantly different. This means that the `childNodes` collection on each platform puts the child `DIV` blocks in completely different index locations in the collections. You can fix this by manually filtering the `childNodes` collections to discard all the `textNodes`. That way you could walk down the `childNodes` tree.

The DOM situation is under development and the DOM Level 2 traversal and range module provides the necessary filters, node iterators and tree walking capabilities that we currently lack. Without these, the DOM implementations are really so incomplete as to only provide marginal help over and above what we had with DHTML.

Structured DIV Blocks

Building a document structure from nested `DIV` blocks has the same disadvantage of needing to be statically included in the same document, although that can be done with a server-side include. It has the major advantage of providing structure. There are limitations, such that the `ID` values need to be unique and HTML 4 compliant in their naming conventions. That may cause problems with generating ID names algorithmically, so they can be fetched more easily.

So, here is an example data block formatted with DIV tags and showing how to build a simple childNodes filter that produces consistently the same list of childNodes on all browsers:

```
<HTML>
<BODY>
   <DIV ID='Level0' STYLE='display:none'>
   <DIV ID='Level1'>
   <DIV ID='Level2'>
      Content
   </DIV>
   </DIV>
   </DIV>

<SCRIPT LANGUAGE="JavaScript">
   var myArray = new Array();
   var ii;
   var jj;

// Get a reference to the top level <DIV>
myObject = document.getElementById("Level0");

// Filter the childNodes collection
jj = 0;
for(ii=0; ii<myObject.childNodes.length; ii++)
{
   if(myObject.childNodes[ii].id)
   {
      myArray[jj] = myObject.childNodes[ii];
   }
}

// Display filtered childNodes
document.write(myArray.length);
document.write("<BR>");
document.write("<BR>");
for(ii=0; ii<myArray.length; ii++)
{
   document.write(myArray[ii]);
   document.write(" : ");
   document.write(myArray[ii].id);
   document.write("<BR>");
}
</SCRIPT>
</BODY>
</HTML>
```

In the News Online ticker, the DIV blocks are structured so that the content is a collection of several story headline texts and the associated links where we can see a page full of text about the story. The additional meta-data is just an item count, so we know how many stories there are. We could store other meta-data too. We could store some controls for the ticker speed, and how often it should be reloaded.

The entire story content and meta-data tree is also enclosed in a single top-level DIV block. This is helpful, because we can set the style of the block so its display: property makes the DIV block invisible. That's also shown in this simpler example. Without that, the content of the DIV blocks would be visible.

On a site, such as News Online, the news story data for the ticker is published into a separate file and a server-side include is used to insert it into the outgoing response from the web server.

An Alternative Using XML Data Islands

XML data islands are supported by IE 5+ and can be accessed easily from JavaScript but this is not yet available on other browsers in a portable manner.

The XML data island is created with the `XML` tag. The navigation mechanism is different to that normally used. We would usually expect to build a tree of nodes using `ID` values assigned as HTML tag attributes. This XML navigation technique uses tag names for navigation and is a lot more powerful.

Here, is an example of a block of XML in the middle of a HTML document:

```
<XML ID='myBlock'>
   <METADATA>
      <OWNER>Wrox</OWNER>
      <DATATYPE>Example</DATATYPE>
      <ABSTRACT>This is an example block of text.</ABSTRACT>
   </METADATA>
</XML>
```

Individual nodes in that so-called data island can be accessed through this `XMLDocument` property. The object returned by this property responds to the `selectSingleNode()` method. The argument to this is the slash-separated path to the node within the document you are looking for. The slash-separated values are the XML tagnames used to construct the document.

In this example, they all begin with the string "`METADATA`", and since the document only contains one layer inside that, all nodes can be reached with the following strings:

- `METADATA/OWNER`
- `METADATA/DATATYPE`
- `METADATA/ABSTRACT`

Given that our XML block has an `ID` value of "`myBlock`", this line of script code should yield a reference to an object that encapsulates the `ABSTRACT` node:

```
myBlock.XMLDocument.selectSingleNode("METADATA/ABSTRACT")
```

Having accessed the DOM node you want, its content can be examined by looking at its `text` property.

The example code illustrates this concept as it might be assembled together in a simple page:

```
<HTML>
<BODY>
<!-- Create an XML island -->
<XML ID='myBlock'>
  <METADATA>
     <OWNER TYPE='PUBLISHER'>Wrox</OWNER>
     <DATATYPE>Example</DATATYPE>
     <ABSTRACT>This is an example block of text.</ABSTRACT>
  </METADATA>
</XML>

<SCRIPT LANGUAGE="JavaScript">
// Get the XML island block
myXMLBlock = document.getElementById("myBlock");

// Get the DOM document
myXMLDocument = myXMLBlock.XMLDocument;
```

```
// Find the node
myNode = myXMLDocument.selectSingleNode("METADATA/ABSTRACT");

// Display the text in the node
alert(myNode.text);

// Now access node attributes and content
myOwner = myXMLDocument.getElementsByTagName('OWNER')[0];
alert(myOwner.getAttribute('TYPE') + ': ' + myOwner.text);

</SCRIPT>
</BODY>
</HTML>
```

That lets us have some very convenient access to textual data, but this is only available in Windows-based versions of IE 5+. We can still accomplish largely the same effect with `DIV` blocks. We can at least build some structure and give them unique `ID` values.

Inserted and Included JavaScript Code

JavaScript code inserted into the document using an include mechanism, is an ideal solution for getting the data feed into the ticker. At News Online, we found that the data coming from the existing feed could not be changed because other systems depended on it. The format of data was such that valid and syntactically correct JavaScript could not be generated reliably due to the placement of certain quote characters. That won't be a problem for us here as we develop a ticker design from scratch and this is likely to be the approach we shall adopt for our optimally designed ticker.

Here is an example of how we can accomplish this. It is amazingly simple, which also makes it a very attractive option.

Save this fragment of script in a file called `data.js`:

```
var myData1 = "String one";
var myData2 = "String two";
var myData3 = "String three";
```

Now we can include that file using a `<SCRIPT SRC="">` tag:

```
<HTML>
<SCRIPT SRC="data.js"></SCRIPT>
   <SCRIPT LANGUAGE="JavaScript">
   function init()
   {
      // Locate the target DIV block
      myTarget = document.getElementById("TextBox");

      // Store the values extracted from the script
      myTarget.innerHTML = myData1 + "<HR>" + myData2 + "<HR>" + myData3;
   }
   </SCRIPT>
<BODY onLoad='init()'>
<DIV ID='TextBox'>
</DIV>
</BODY>
</HTML>
```

The really neat thing about this approach, is that it factors the ticker code and the data into separate files, and the data in the included file can be rendered from a publishing system quite independently of the ticker software.

Implementing Our New Ticker Design

We need a way to include our ticker into a page. The `IFRAME` technique was good, and although it doesn't work on some older Netscape browsers, we can work around that. We can put in some kind of link that takes the user to a static page containing the headlines. Assuming we are driving this all from a content management system, creating the page containing the headlines should be fairly easy.

So, first of all, here is the code that you need to place into your page to include the ticker:

```
<HTML>
<BODY>
<IFRAME SRC='ticker.html' WIDTH='315' HEIGHT='60'
      SCROLLING='no' FRAMEBORDER='0'>
      <A HREF='headlines.html'>Click here for headlines</A>
</IFRAME>
</BODY>
</HTML>
```

The content of the `ticker.html` file is a combination of HTML, JavaScript, and CSS style control. Here is the main HTML structure with the CSS and JavaScript blocks condensed so you can see the overall layout:

```
<HTML>
<HEAD>
   <STYLE TYPE='text/css'>
      … CSS Styles here …
   </STYLE>
   <SCRIPT SRC="data.js">
   </SCRIPT>
   <SCRIPT LANGUAGE="JavaScript">
      … Ticker execution script here …
   </SCRIPT>
</HEAD>
<BODY onLoad='startTicker();'>
<A ID='Anchor' HREF='/' target=_top></A>
</BODY>
</HTML>
```

In the `HEAD` section, there is a `STYLE` block and two independent `SCRIPT` blocks.

The CSS block is for styling control. The two separate `SCRIPT` blocks are provided to abstract the executable ticker code from the data. The data block is included from a separate file so your content management system can publish new ticker data without having to republish the ticker code. We talked about this a little while ago when we discussed different ways of passing data to the ticker.

The `BODY` of the document is very simple; it's just the `anchor` tag. We set as a criterion the simplification of the design if it was possible. This is a much simpler `BODY` than the original News Online ticker.

The CSS styling is necessary to give us control over the hovering appearance, and to give us a way to eliminate the underscore that the browser automatically places on a link. This reveals a small bug in the Netscape 6 CSS object model, which requires that implemented CSS style properties must be specified, otherwise the rules are not created. This results in an empty `cssRules` collection and crashes the browser:

```
A
{
   display: none;
}
```

The following instantiates a rule and the browser works fine:

```
A
{
   text-decoration: none;
}
```

Here is the content of the `STYLE` block in the `HEAD` section of our new ticker:

```
<STYLE TYPE='text/css'>
A
{
   text-decoration: none;
}

A:hover
{
   text-decoration: none;
}
</STYLE>
```

Note that we only define the `text-decoration:` property. All the others will be defined from script, having located the style object that each of these rules instantiates. You could define more default values here and set up fewer properties in the script.

The first `SCRIPT` block contains the ticker data:

```
<SCRIPT SRC="data.js">
</SCRIPT>
```

Note that it includes an external file. This provides a good way to integrate things with your content management system. It also separates content from functionality. In this implementation, we can also control rather more of the stylistic appearance than we could with the original BBC News Online ticker. Here is the content of the `data.js` file. The story texts and URLs have been shortened to save space here:

```
// Control parameters
var theCharacterTimeout = 50;
var theStoryTimeout     = 5000;
var theWidgetOne        =  "_";
var theWidgetTwo        =  "-";
var theWidgetNone       =  "";
var theLeadString       = "LATEST: ";

// Styling parameters
var theBackgroundColor  = "white";
var theForegroundColor  = "#333366";
var theFontFamily       = "Verdana, Arial, Helvetica, sans-serif";
var theFontSize         = "11px";
var theLineHeight       = "11px";
var theFontWeight       = "bold";
var theTextDecoration   = "none";
var theHoverColor       = "#CC3300";

// Content parameters
var theSummaries = new Array();
var theSiteLinks = new Array();

var theItemCount = 4;

theSummaries[0] = "This is the stext for story 1";
theSiteLinks[0] = story1.htm";
```

```
theSummaries[1] = "Here is story 2.";
theSiteLinks[1] = "/newsid_1405000/1405821.htm";

theSummaries[2] = "Story three has its headline here";
theSiteLinks[2] = "./Story3.html";

theSummaries[3] = "This is headline number four";
theSiteLinks[3] = "/stories/Four.htm";
```

The variable names should be fairly obvious. The control variables are similar to the ones that existed before, except that we can access them externally. There is an additional widget value that is placed at the end of the text, only when the story headline is complete. The leading string is also defined here instead of requiring a background image.

The CSS style property names are reflected in the variables that contain their values. These are assigned to the appropriate properties belonging to the style objects for the `A` rule, the `A:hover` rule and the document body object. This will become apparent shortly when we walk through the code.

The second `SCRIPT` block contains the functional code that makes the ticker work. This is much simpler than the original design we looked at. We'll work through the various components in there, starting with the more basic ones first.

It is necessary to construct a series of non-breaking spaces separated by breaking spaces so that they can be placed on the end of the anchor text so as to completely fill the `IFRAME`. This helps the UI to be a little more responsive and because of memory leak avoidance, this was done as a constant in the BBC ticker. Here it is created with a small function. There might be a memory leak due to the string concatenations, but because this is called only once it should not cause a serious problem. Here is the function that builds the space filler:

```
function buildSpaceFiller(aCount)
{
   var myResult = "";

   for(var ii=0; ii<aCount; ii++)
   {
      myResult = myResult + "  ";
   }

   return myResult;
}
```

We need to generate a widget character to create an animated golf ball effect, while the text 'teletypes' out. This function does the same job as the earlier example with one difference. Here we also return a special widget if we are at the end of the line which allows us to place a trailing symbol on the output if we want to. That could be done with a small `IMG` item. Here is the widget generator:

```
function whatWidget()
{
   if(theCurrentLength == theStorySummary.length)
   {
      return theWidgetNone;
   }

   if((theCurrentLength % 2) == 1)
   {
      return theWidgetOne;
   }
   else
   {
```

```
        return theWidgetTwo;
    }
}
```

There are just two functions left to describe. One starts the ticker after initializing all the style settings, while the other is the ticker `run` loop that calls itself with the `setTimeout()` function. Let's look at the start up function first:

```
function startTicker()
{
    // Define run time values
    theCurrentStory     = -1;
    theCurrentLength    = 0;
    theSpaceFiller      = buildSpaceFiller(200);


    // Locate base objects
    theAnchorObject     = document.getElementById("Anchor");

    // Locate style sheet objects
    theStyleSheet = document.styleSheets[0];


    // Fix the missing cssRules property for MSIE
    if(!theStyleSheet.cssRules)
    {
        theStyleSheet.cssRules = theStyleSheet.rules;
    }

    // Locate the style objects we want to modify
    theBodyStyle   = document.body.style;
    theAnchorStyle = theStyleSheet.cssRules[0].style;
    theHoverStyle  = theStyleSheet.cssRules[1].style;

    // Apply data driven style changes
    theBodyStyle.backgroundColor = theBackgroundColor;

    theAnchorStyle.color = theForegroundColor;
    theAnchorStyle.fontFamily = theFontFamily;
    theAnchorStyle.fontSize = theFontSize;
    theAnchorStyle.lineHeight = theLineHeight;
    theAnchorStyle.fontWeight = theFontWeight;
    theAnchorStyle.textDecoration = theTextDecoration;

    theHoverStyle.color = theHoverColor;

    // Fire up the ticker
    runTheTicker();
}
```

First, we call the `spaceFiller` and define some initial default values. Then we locate the `anchor` object using the `getElemerntById()` method. We then locate the `styleSheets` collection for the document and fix it up to correct the missing `cssRules` property on IE browsers.

Following that fix, we can then locate the `style` objects for the two CSS rules in the `STYLE` tags, one for all `anchor` elements, and the other for the pseudo element that applies style to anchors when they are being hovered over. We also locate the `style` object for the document body.

Having located all of these `style` objects, we now assign values to various properties, using the variables that were defined in the included content file (`data.js`).

Finally, we call the main ticker `run` loop to start things rolling.

Here is the code for that main animation function:

```
function runTheTicker()
{
   var myTimeout;

   // Go for the next story data block
   if(theCurrentLength == 0)
   {
      theCurrentStory++;
      theCurrentStory      = theCurrentStory % theItemCount;
      theStorySummary      = theSummaries[theCurrentStory];
      theTargetLink        = theSiteLinks[theCurrentStory];
      theAnchorObject.href = theTargetLink;
   }

   // Stuff the current ticker text into the anchor
   theAnchorObject.innerHTML = theLeadString +
theStorySummary.substring(0,theCurrentLength) + whatWidget() + theSpaceFiller;

   // Modify the length for the substring and define the timer
   if(theCurrentLength != theStorySummary.length)
   {
      theCurrentLength++;
      myTimeout = theCharacterTimeout;
   }
   else
   {
      theCurrentLength = 0;
      myTimeout = theStoryTimeout;
   }

   // Call up the next cycle of the ticker
   setTimeout("runTheTicker()", myTimeout);

}
```

This is much simpler than the earlier version. Several unnecessary variables are eliminated and all state preservation is accomplished using `theCurrentLength` value. Several duplicate fragments of code have been eliminated by moving the widget generator into a function as well as by pulling common code outside of the conditional test for whether we are at the end or in the middle of a line of ticker playout.

As we have totally eliminated all the DOM `childNodes` problems, this now works fine on IE and Netscape 6, although there are still some necessary functionalities missing from the Opera browser's JavaScript implementation that prevents the ticker from running.

Summary

We have examined a working case history of a ticker being used in a very busy site. We've learned that the design criteria that seemed reasonable in 1999, do not hold up over time and this gives us some cause for concern. We might make similar judgement calls now and still encounter similar problems with browsers because we did not anticipate their behavior. Hopefully, the continuing standards creation and evolution will address those issues and in time they will be less severe. A standards-based approach is probably the safest course.

We have looked at several alternative ways to solve our problems, and have designed a far better ticker. Around the time this book is published, that new ticker (or something similar to it), will be in daily use on the BBC News Online web site.

So, returning to our new design criteria, let's see if we have satisfied our original goals:

- It now runs on more platform/browser combinations than it did before.
- The code is most definitely simpler and is about half the length it was before.
- We now have a really optimum data passing mechanism and it could still be integrated into the workflow of a content management process.
- We have added significant style control functionality.
- We didn't find a better event control mechanism, but the behavior appears to be stable and doesn't crash.
- We have replaced the need for the `APPLET` by allowing the user to call an external page of headlines. We could replace that with all of the ticker headlines in place, but that might change the geometry of the page.
- DOM standardization does not really help us in solving a problem like this, because DOM is not nearly complete enough to be useful. We really need traversal and range to be implemented, or we have to write filter scripts to eliminate unwanted text nodes from the child node collections.