In addition, if the HTML element being represented by the DOM object has any other attributes, those attributes and their values are also represented as properties in the element's DOM object. For example, if an `img` tag has a `src` attribute set to `"dog.gif"`, that element's DOM object will have a `.src` property whose value is `"dog.gif"`. In your JavaScript code you can examine these properties' values or change them to alter the appearance of the page. Most DOM object property names are identical to names of the corresponding attributes in the HTML, with a few notable exceptions such as the `class` HTML attribute that maps to the `className` DOM property. Table 7.8 lists some other properties you may expect to find in various DOM objects.

Property	Type	Description
`checked`	boolean	Whether a checkbox or radio button is checked
`disabled`	boolean	Whether a form control is disabled
`href`	string	Target URL for a link (`a`)
`src`	string	Source URL for an image (`img`)
`value`	string	Text inside a form control such as an `input` or `textarea`

Table 7.8 Additional properties of some DOM objects

For example, suppose there is an image on the page that you want to change from displaying a picture of a cat stored in `cat.jpg`, represented by the following HTML code:

```
<img id="mypet" src="cat.jpg" alt="my favorite pet" />
```

Suppose you want to the image's source URL to instead show an image of a dog stored in `dog.jpg`. Recall that you can access an HTML element's corresponding DOM object by calling `document.getElementById`. Therefore the following JavaScript code would do the trick:

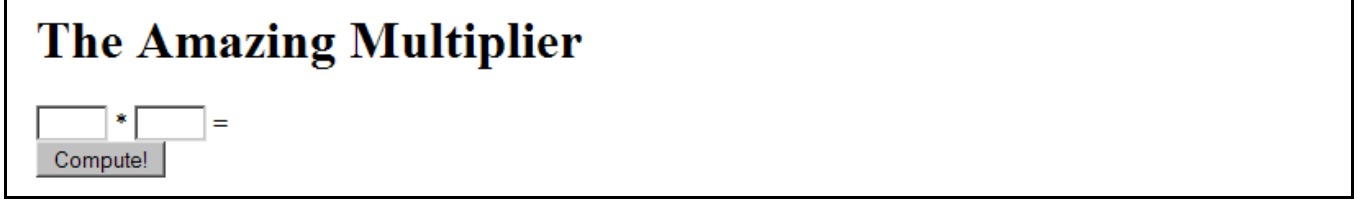
```
var myImage = document.getElementById("mypet");
myImage.src = "dog.jpg";
```

There are a lot more properties inside each DOM object that we will explore in the next chapter, such as properties related to the styling and appearance of the element.

Example Program: Multiplier

Let's write a page that acts as a simple multiplication calculator. The page will contain two input text boxes. When the user types an integer into each box and presses a Compute button, the numbers will be multiplied and the answer will be shown. Example 7.13 shows the initial HTML code.

```
<h1>The Amazing Multiplier</h1>
<div>
  <input type="text" size="3" /> * <input type="text" size="3" /> =<br />
  <button>Compute!</button>
</div>
```



Example 7.13 Multiplier initial HTML code

To bring this page to life with JavaScript, we'll first need to link the page to a JavaScript file. This is done by adding a `script` reference to the page's header:

```
<script src="multiply.js" type="text/javascript"></script>
```

In the previous section about the DOM, we saw that JavaScript code can interact with elements on the page. Conceptually what we'd like to do is fetch the integers inside both text boxes, multiply them together, and display the result on the page. To do this, we'll need to access the DOM objects for both text boxes.

Earlier in this chapter we saw that you can access an HTML element's DOM object using the `document.getElementById` method. This means that in order to grab the integers to multiply, we'll need the two `input` elements to have `id` attributes. We will also need a place to display the answer on the page. Let's create an empty `span` with an `id` of `answer` and put the result into that span. Lastly, we want our JavaScript code to run when the Compute button is clicked, so let's add an `onclick` attribute that calls a JavaScript function named `compute` that we'll write in multiply.js.

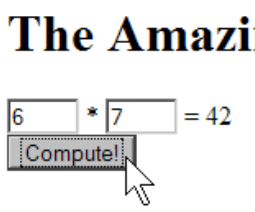
```
<h1>The Amazing Multiplier</h1>
<div>
  <input id="num1" type="text" size="3" /> *
  <input id="num2" type="text" size="3" /> =
  <span id="answer"></span> <br />
  <button onclick="compute();">Compute!</button>
</div>
```

Example 7.14 Multiplier improved HTML code

Now that our HTML page is more JavaScript-friendly, let's write the `compute` function to be called when our button is clicked. We start by retrieving three elements by their ids: the two `input` boxes and the answer `span`. We can grab the text out of an input control by accessing its `value` property, which directly mirrors the `value` attribute in the HTML. Once we've multiplied the two numbers, we can store the result into the answer `span` by setting its `innerHTML` property. Example 7.15 shows the complete code and a screenshot of the appearance after the button is clicked.

```
// Multiplies two numbers typed into input boxes on the page,
// and displays the result in a span on the page.

function compute() {
  var input1 = document.getElementById("num1");   // fetch the 2 numbers
  var input2 = document.getElementById("num2");
  var answer = document.getElementById("answer");
  var result = input1.value * input2.value;       // compute result
  answer.innerHTML = result;
}
```



Example 7.15 JavaScript multiplier code

Many students get the `value` and `innerHTML` properties mixed up. It can be a bit confusing that most elements' DOM objects hold their inner content in `innerHTML`, but that controls like text boxes store their values in `value`. The thing to remember is that `innerHTML` is analogous to text between an element's opening and closing tag. An `input` element has no such inner content because it is a self-closing tag. The difference can be summarized by the following example:

```
<div>this is innerHTML inside a div element</div>
<input type="text" value="this is value inside an input element" />
```

7.2.6 Debugging Common Errors (a.k.a., "Why Doesn't My Program Do Anything?")

By far, the most common report we hear from students learning JavaScript is, "When I press the button, nothing happens! What's wrong?" Unfortunately, the web browser isn't a very friendly environment for software development. By default if anything is wrong with your JavaScript program, you will see nothing. The browser won't show any error message, and nothing will appear on the page to indicate an error. Nothing will happen, as though you didn't write any JavaScript code at all. New JavaScript programmers can become very frustrated by programs that won't respond and give no hint of what is causing the problem.

There are several reasons a JavaScript program might do nothing. In this section we explore a few of the most common ones and give general tips about how to diagnose the problem.

If you specified the wrong file name in the script tag at the top of your page, the browser will be unable to load your JavaScript code, and no error message will be initially shown. Let's consider our previous Multiplier example. Imagine that the programmer has misspelled the name of the JavaScript file multilpy.js (sic) in the HTML header, as shown in Example 7.16.

Common Error

Failing to link to JavaScript file properly

(Fix: Place an `alert` at the top of your .js file as a sanity check, and/or use Firebug)

```
<script src="multilpy.js" type="text/javascript"></script>
```

Example 7.16 Common error: misspelled script file name

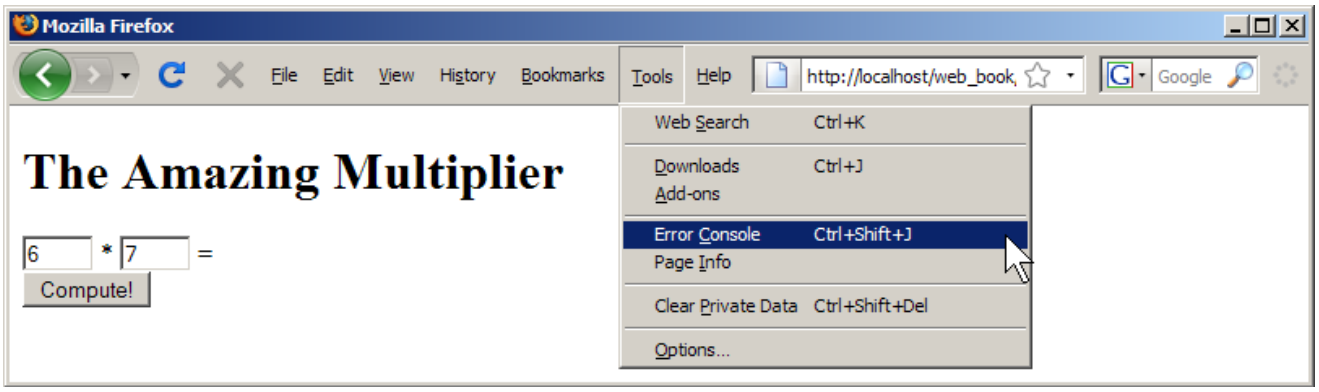


Figure 7.6 Opening the Firefox Error Console

The page shows up in the browser without any problems, but when you click the Compute button, nothing happens. No hint of any error is shown. In some browsers, you can view a log of JavaScript errors to help diagnose what is wrong. In Firefox, you can access an Error Console by clicking Tools, Error Console, as shown in Figure 7.6.

After clicking the Compute button error message shows up in our Error Console stating that the `compute` function is not defined, as shown in Figure 7.7. If you install the Firebug add-on for Firefox (which we *highly recommend* when debugging JavaScript programs), the moment you click the Compute button, you will see a red message in the bottom-right of your browser window indicating an error. Clicking this message shows more information about the error, as shown in Figure 7.8.

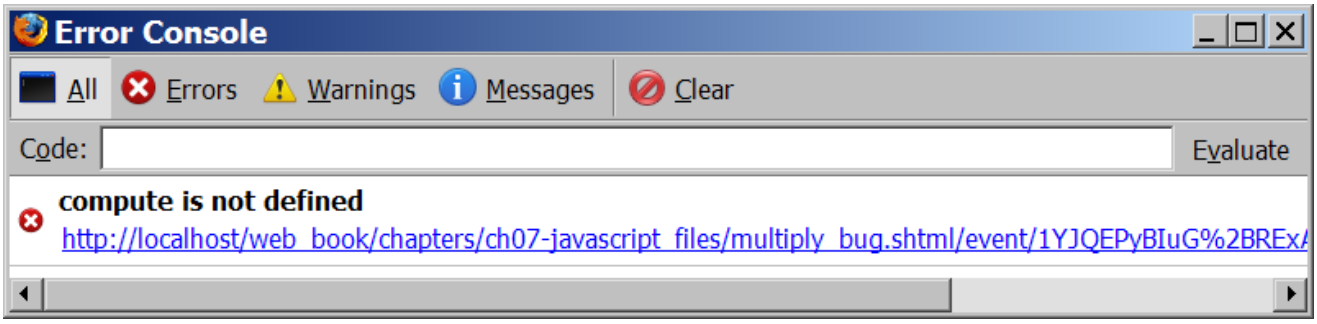


Figure 7.7 Error console displaying an error message

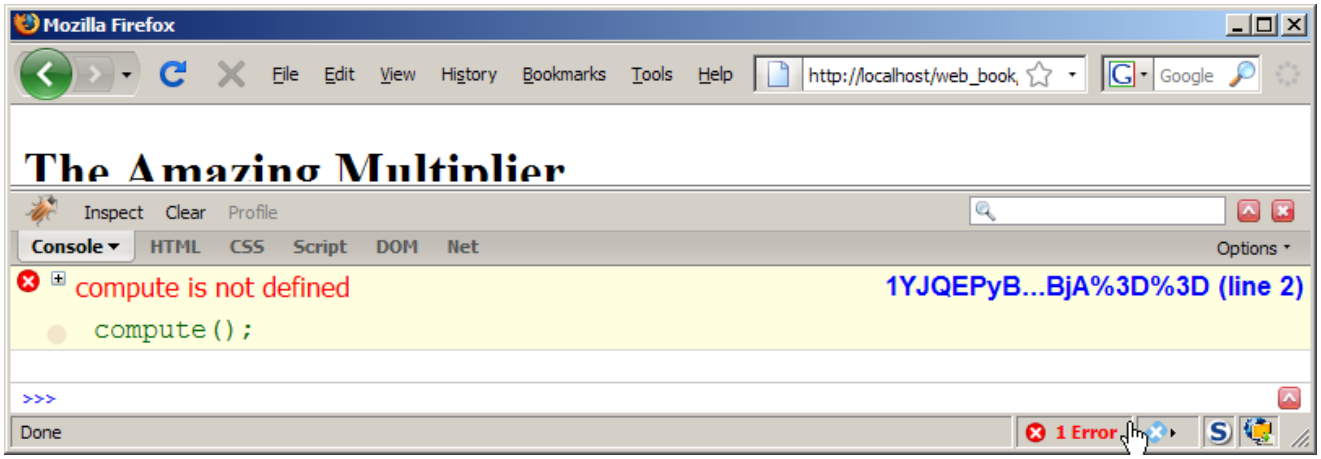


Figure 7.8 Firebug add-on displaying error message

Now that we see this error, we double-check our code and are sure that we defined a function named `compute`. The fact that the browser cannot find this function indicates that it may not be properly finding our JavaScript file. Firebug provides evidence of this if you click its Net tab. This tab shows all of the files the browser has tried to fetch during the loading of the page. As seen in Figure 7.9, the browser got an HTTP 404 error (file not found) when trying to load the script multilpy.js.

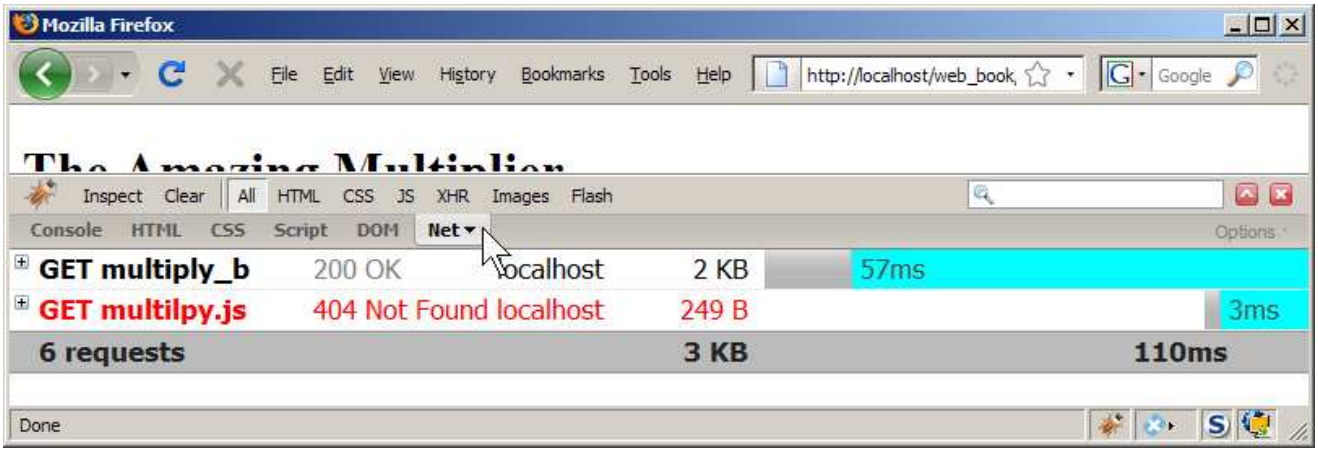
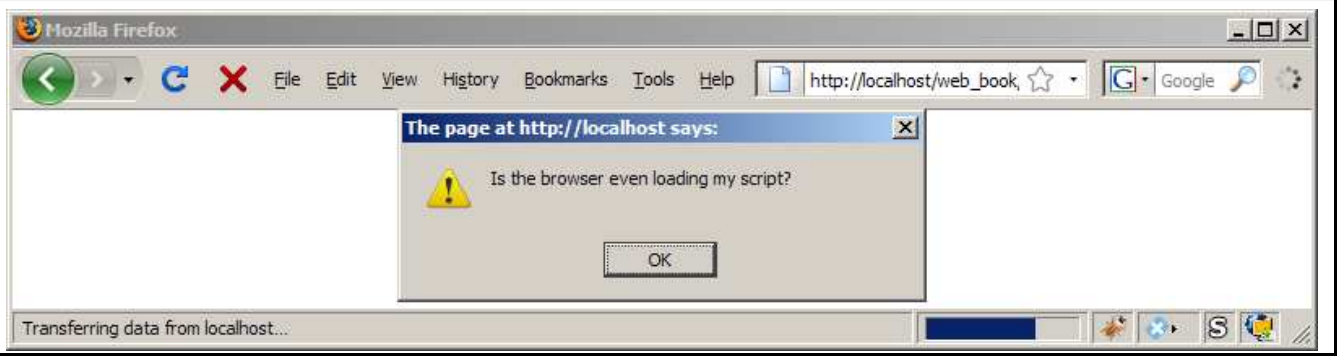


Figure 7.9 Firebug Net tab

Another good sanity check to make sure the browser is loading your file is to insert an `alert` statement at the very top of your file, outside of any function. If the browser loads your .js file, it will immediately show this `alert` message box on the screen. If you don't see the `alert` box, you know that the browser didn't load the script file. Example 7.17 shows an example of this technique and a screenshot of the appearance that would result if the box popped up. The box would not pop up with our buggy code, but once we discover the error, fix it, and reload the page, the `alert` appears.

```
alert("Is the browser even loading my script?");

function compute() {
  var input1 = document.getElementById("num1");   // fetch the 2 numbers
  var input2 = document.getElementById("num2");
  var answer = document.getElementById("answer");
  var result = input1.value * input2.value;       // compute result
  answer.innerHTML = result;
}
```



Example 7.17 Using an `alert` to test whether a script is loaded

The `alert` trick isn't limited to the top of the file. You can imagine a similar bug where the HTML file misspells the name of the `compute` function we're trying to call, writing it as `copmuet` or similar. In a case like this, we'll again get a message saying that the function is undefined. If you put an `alert` statement at the start of the `compute` function, you'd notice that it was not being called, which should give a strong hint that the page is not correctly naming the function it is trying to call.

Let's examine another error many students make. Recall that our Multiplier page has two input elements with `id`s of `num1` and `num2` into which the user types the two numbers to multiply.

```
  <input id="num1" type="text" size="3" /> *
  <input id="num2" type="text" size="3" /> =
```

Common Error

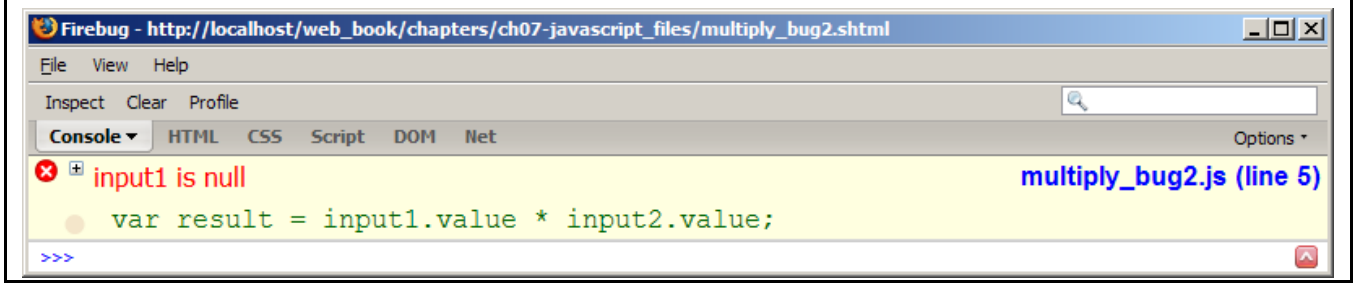
Trying to access an undefined object or property

(Fix: Use Firebug to find the line with an error, and double-check relevant expressions.)

Suppose we have a bug in our JavaScript code where we accidentally try to fetch the DOM objects for these elements using the wrong `id`s (in this case, `number1` and `number2` instead of `num1` and `num2`). Again, when we go to use the web page and click the Compute button, the symptom we get is that nothing happens. Luckily, Firebug again shows the helpful error message to indicate that our page has an error.

Example 7.18 shows the faulty code and the error that shows up in Firebug. The error says that the variable `input1` is `null`. The value `null` is a special empty value returned by some functions to indicate an empty value or an error condition. The `document.getElementById` method returns `null` when there is no element on the page with the `id` specified.

```
function compute() {
  var input1 = document.getElementById("number1");
  var input2 = document.getElementById("number2");
  var answer = document.getElementById("answer");
  var result = input1.value * input2.value;
  answer.innerHTML = result;
}
```



Example 7.18 Incorrect JavaScript multiplier code (wrong ids)

A very helpful facility in Firebug is its Console, where you can type in arbitrary JavaScript statements or expressions. The Console will execute the expression or statement and immediately show its result. We encourage our students to try typing in short pieces of their code into this console to see what the results are, in the hope of finding an error. In this case, since the error stated that `input1` was `null`, perhaps we'll decide to double-check the lines of code that fetched `input1`'s value. If you type `document.getElementById("number1")` into the console, it shows `null` as the result. Hopefully at this moment we double-check the HTML and realize our mistake, then try it again in Firebug with `num1` instead of `number1`, and the `input` tag is shown as originally intended. Figure 7.10 shows this usage of the Console.

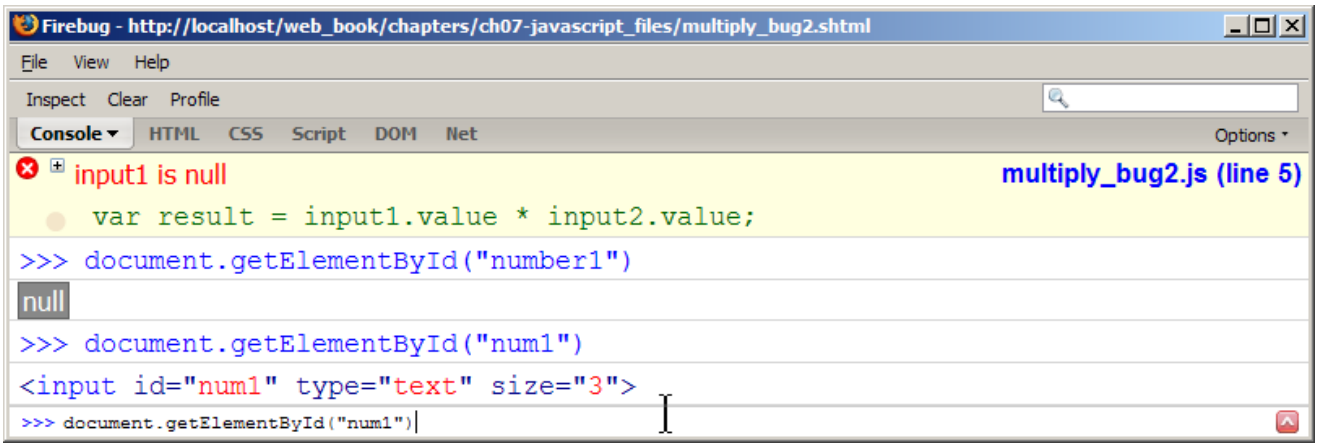


Figure 7.10 Using Firebug's Console to debug DOM errors

Another very useful utility for finding JavaScript errors is JSLint, by Douglas Crockford of Yahoo. JSLint is a web page that attempts to make up for the fact that JavaScript has no compiler by analyzing your JavaScript code and reporting possible errors in it. Its author Douglas Crockford is considered one of the preeminent authorities on JavaScript programming. If your code doesn't work and you're not sure why, you may want to paste it into JSLint at http://www.jslint.com/ and see if any helpful error messages present themselves. Figure 7.11 shows a screenshot of JSLint in action.

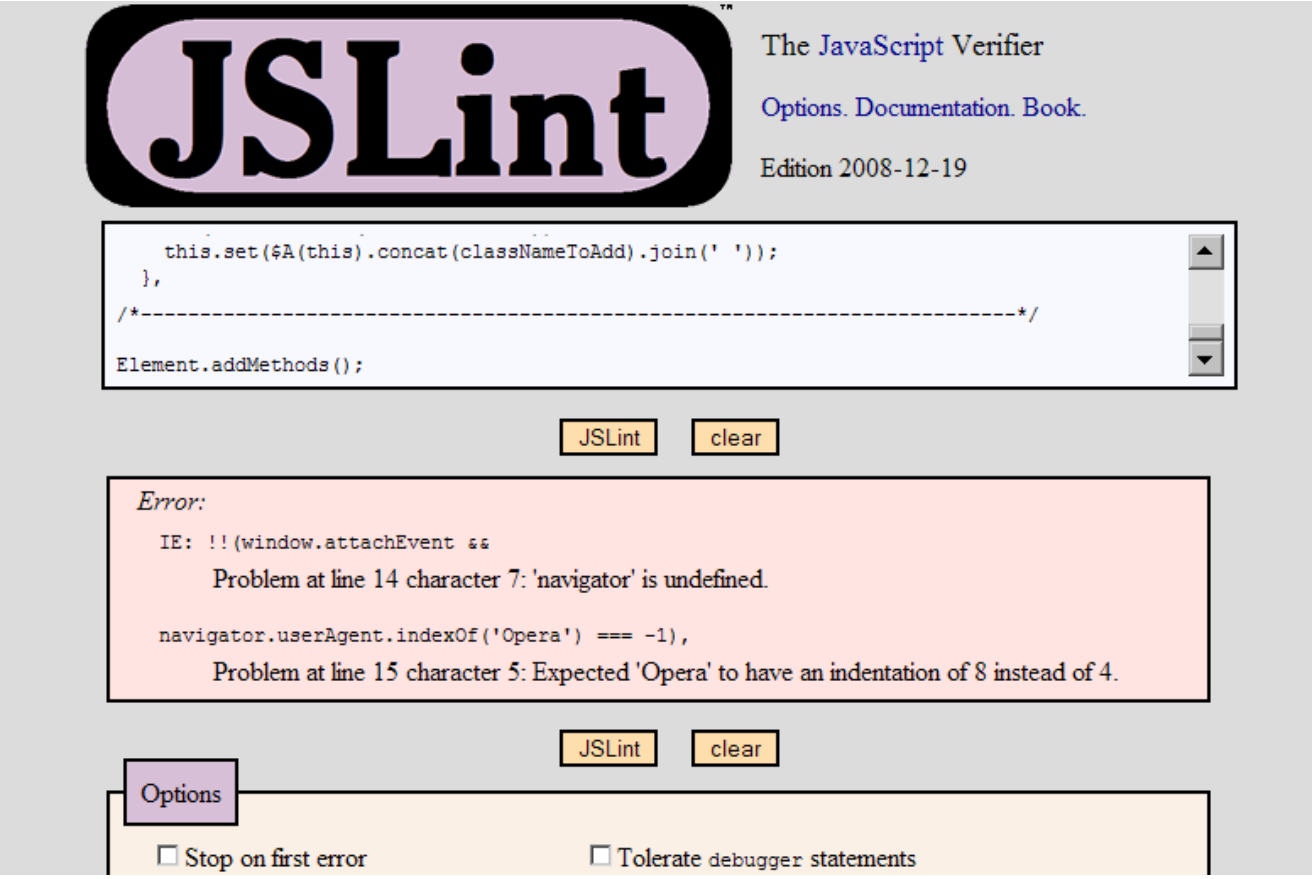


Figure 7.11 JSLint syntax verifier

7.2.7 Strings

JavaScript strings represent sequences of characters. Unlike C or Java, there is no `char` type to represent individual characters; even a single character is represented as a string.

JavaScript string literals can be specified with either single quotes such as `'Hola'` or double quotes such as `"Salut"`. There is no difference between the two except the fact that if you want an apostrophe inside a single-quoted string, you must escape it as `\'`, while in a double-quoted string if you want a quotation mark you must escape it as `\"`. We generally use double-quoted strings in our examples.

Other special characters within a JavaScript string literal, such as `'` and `"`, can be escaped with a backslash just like in Java and C. Table 7.9 lists the common escape sequences.

Name	Escape Sequence
single quote	`\'`
double quote	`\"`
backslash	`\\`
new line	`\n`
horizontal tab	`\t`

Table 7.9 Escape sequences