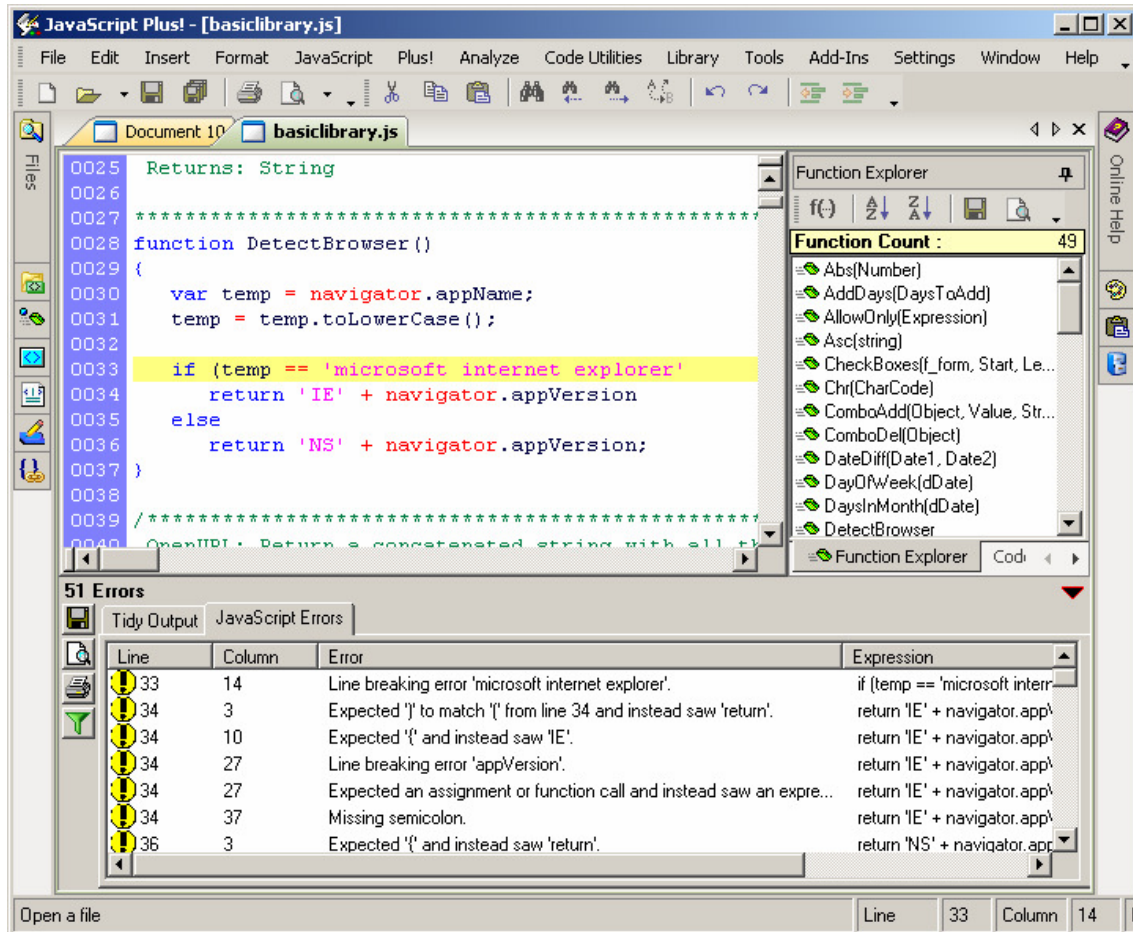


# Table of Contents

<b><i>Analyzing javascript files with JavaScript Plus!</i></b>	<b>2</b>
<b>JavaScript Error Panel</b>	<b>2</b>
<b>Icon Information</b>	<b>3</b>
<b>Configure JavaScript Analyzer</b>	<b>3</b>
<b>Batch Mode</b>	<b>5</b>
<b>Button Help</b>	<b>5</b>
Configure file extensions	5
<b>Technical Information</b>	<b>6</b>
<b><i>What is JSLint ?</i></b>	<b>7</b>
<b>Undefined Variables and Functions</b>	<b>7</b>
<b>Members</b>	<b>8</b>
<b>Semicolon</b>	<b>8</b>
<b>Line Breaking</b>	<b>8</b>
<b>Comma</b>	<b>9</b>
<b>Required Blocks</b>	<b>9</b>
<b>Forbidden Blocks</b>	<b>9</b>
<b>Expression Statements</b>	<b>10</b>
<b>for in</b>	<b>10</b>
<b>switch</b>	<b>10</b>
<b>var</b>	<b>10</b>
<b>with</b>	<b>10</b>
<b>=</b>	<b>11</b>
<b>== and !=</b>	<b>11</b>
<b>Labels</b>	<b>11</b>
<b>Unreachable Code</b>	<b>12</b>
<b>Confusing Pluses and Minuses</b>	<b>12</b>
<b>++ and --</b>	<b>12</b>
<b>Bitwise Operators</b>	<b>12</b>
<b>eval is evil</b>	<b>12</b>
<b>void</b>	<b>12</b>
<b>Regular Expressions</b>	<b>12</b>
<b>Constructors and new</b>	<b>13</b>
<b>Unsafe Characters</b>	<b>13</b>
<b>Not Looked For</b>	<b>13</b>
<b>HTML</b>	<b>13</b>
<b>CSS</b>	<b>14</b>
<b>Report</b>	<b>14</b>

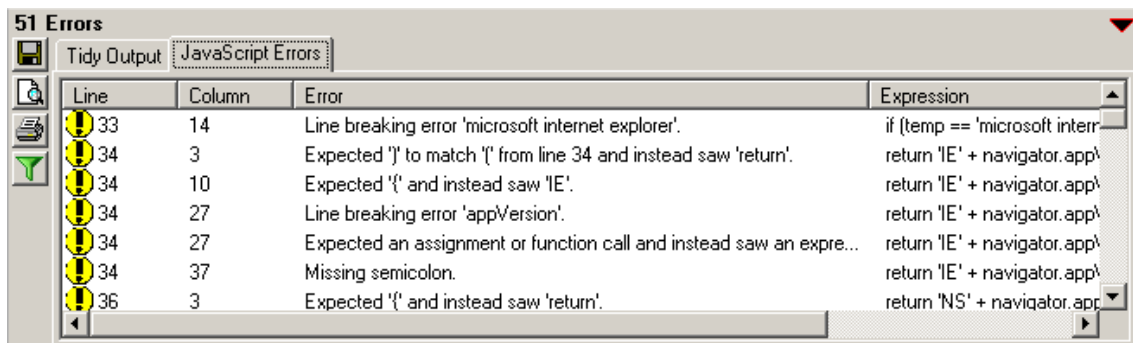
## Analyzing javascript files with JavaScript Plus!

You can search for javascript errors using **JavaScript Plus!**. To search for errors clic on **Analyze** menu and clic on **"Find Errors"**.



The javascript analyzer will try to find javascript errors in the current document. The errors will be displayed in the **JavaScript Error Panel**.

### JavaScript Error Panel








The JavaScript error panel show information about the error detected in the javascript file. The error panel contains the next information :

Item	Description
Line	Line where error was detected.
Column	Column where error was detected.
Error	Information about error detected.
Expression	Information about the expression where analyzer find error.

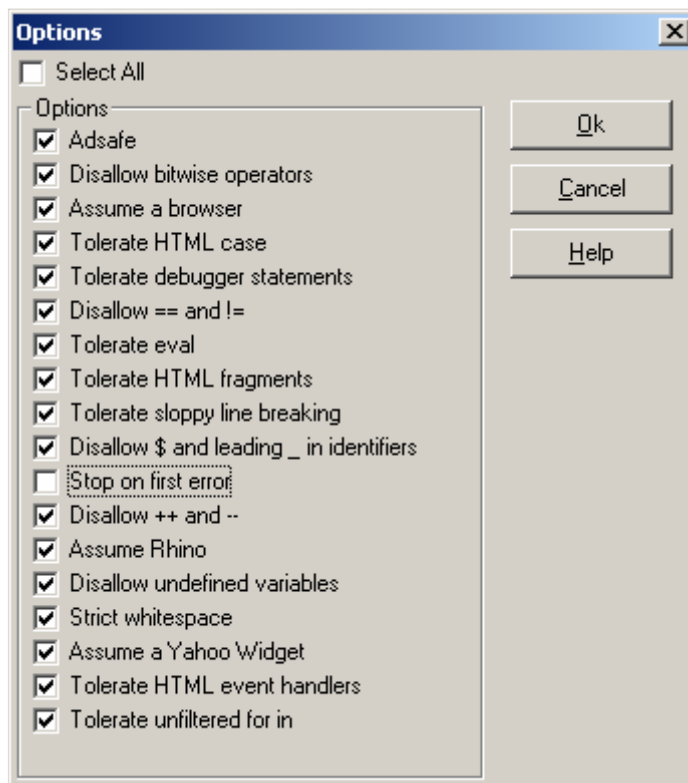
Click an item from the JavaScript Error Panel to activate code line on editor.

### ***Icon Information***

Icon	Help
	Save errors detected to text file.
	Save errors detected to html file.
	Information about error detected.
	Configure options to be analyzed.
	Show/Hide the JavaScript Error Help Panel.

### ***Configure JavaScript Analyzer***

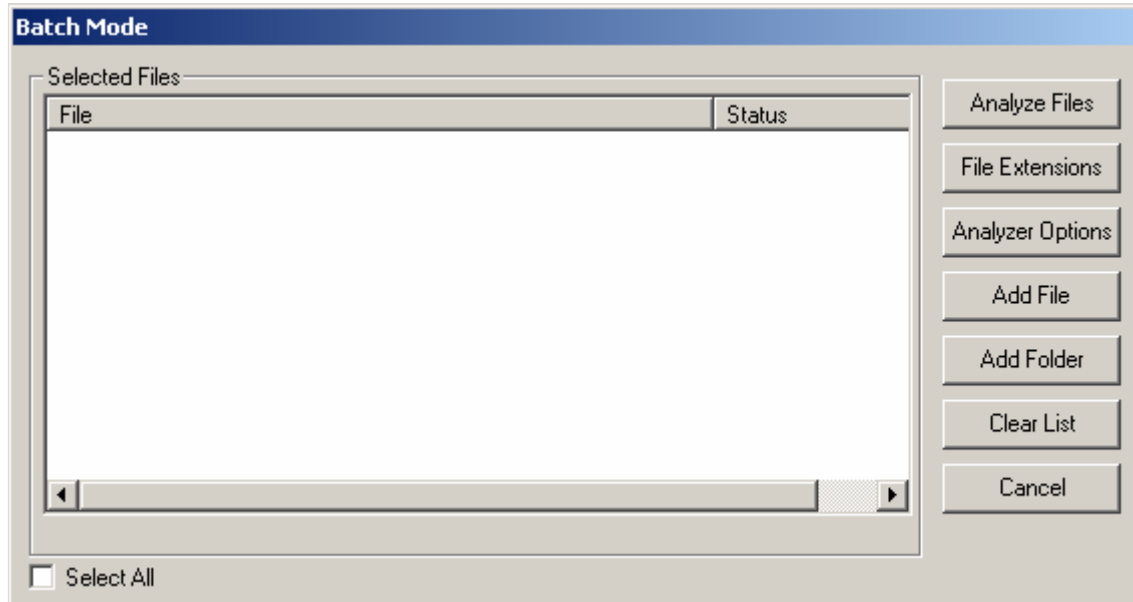
To configure the options to be analyzed by the **JavaScript Analyzer** clic on Analyze menu and clic on **Configure** menu item.



Option	Description
<b>Ad Safe</b>	true if ADsafe.org rules widget pattern should be enforced
<b>Disallow bitwise operations</b>	true if bitwise operators should not be allowed
<b>Assume a browser</b>	true if the standard browser globals should be predefined
<b>Tolerate HTML Case</b>	true if upper case HTML should be allowed
<b>Tolerate Debugger Statements</b>	true if debugger statements should be allowed
<b>Disallow == and !=</b>	true if === should be required
<b>Tolerate eval</b>	true if eval should be allowed
<b>Tolerate html fragments</b>	true if HTML fragments should be allowed
<b>Tolerate sloppy line breaking</b>	true if statement breaks should not be checked
<b>Disallow \$ and leading _ in identifiers</b>	true if names should be checked for initial underbars
<b>Stop on first error</b>	true if the scan should stop on first error
<b>Disallow ++ and --</b>	true if ++ and -- should not be allowed
<b>Assume Rhino</b>	true if the Rhino environment globals should be predefined
<b>Disallow undefined variables</b>	true if undefined global variables are errors
<b>Strict whitespace</b>	true if strict whitespace rules apply
<b>Assume a Yahoo Widget</b>	true if the Yahoo Widgets globals should be predefined
<b>Tolerate HTML events handlers</b>	true if HTML event handlers should be allowed
<b>Tolerate unfiltered for in</b>	true if unfiltered for in statements should be allowed

## Batch Mode

JavaScript Plus! can analyze javascript files in batch mode. The errors are reported to html file detailed by file. Click on **Analyze menu** and click on **Batch Mode** menu item.

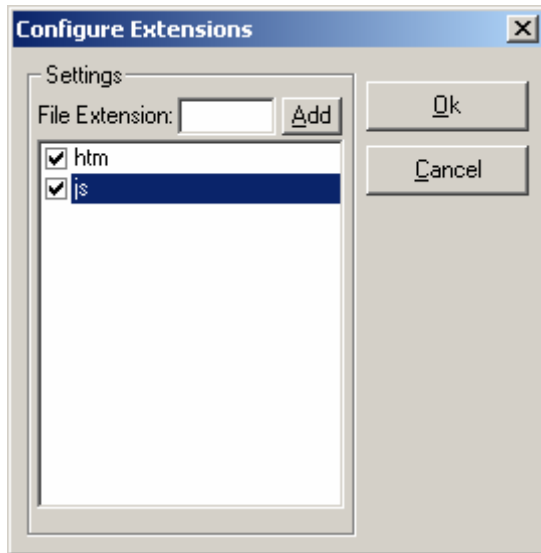


## Button Help

Button	Help
<b>Analyze Files</b>	Analyze the selected files.
<b>File Extensions</b>	Configure the file extensions to be analyzed.
<b>Analyzer Options</b>	Configure the options to be used by the javascript analyzer.
<b>Add File</b>	Add a file to be analyzed.
<b>Add Folder</b>	Add files from selected folder to be analyzed. The files to be added must match the file extension allowed under file extension.
<b>Clear List</b>	Clear the file list.
<b>Cancel</b>	Close window.

## Configure file extensions

To configure **file extensions** to be analyzed. Default extensions are .js and .html.



To add a file extension input the extension and clic to **Add button**.

Clic **Ok** to save changes.

## ***Technical Information***

JavaScript Plus! uses **JSLint** from Douglas Crockford <http://www.jshint.com/lint.html> to analyze javascript files. To load and parse javascript files JavaScript Plus! needs **Microsoft Internet Explorer** 6.0 or higher to analyze files.

The analyzer is a javascript file located under jsplus\jshint folder. **Never delete/remove/modify the jshint.js.**

## What is JSLint ?

JSLint is a JavaScript program that looks for problems in JavaScript programs.

When [C](#) was a [young](#) programming language, there were several common programming errors that were not caught by the primitive compilers, so an accessory program called [lint](#) was developed which would scan a source file, looking for problems.

As the language matured, the definition of the language was strengthened to eliminate some insecurities, and compilers got better at issuing warnings. lint is no longer needed.

[JavaScript](#) is a young-for-its-age language. It was originally intended to do small tasks in webpages, tasks for which Java was too heavy and clumsy. But JavaScript is a very capable language, and it is now being used in larger projects. Many of the features that were intended to make the language easy to use are troublesome for larger projects. A lint for JavaScript is needed: JSLint, a JavaScript syntax checker and validator.

JSLint takes a JavaScript source and scans it. If it finds a problem, it returns a message describing the problem and an approximate location within the source. The problem is not necessarily a syntax error, although it often is. JSLint looks at some style conventions as well as structural problems. It does not prove that your program is correct. It just provides another set of eyes to help spot problems.

JSLint defines a professional subset of JavaScript, a stricter language than that defined by [Edition 3 of the ECMAScript Language Specification](#). The subset is related to recommendations found in [Code Conventions for the JavaScript Programming Language](#).

JavaScript is a sloppy language, but inside it there is an elegant, better language. JSLint helps you to program in that better language and to avoid most of the slop.

JSLint can operate on JavaScript source, HTML source, or [JSON](#) text.

## Undefined Variables and Functions

JavaScript's [biggest problem](#) is its dependence on global variables, particularly implied global variables. If a variable is not explicitly declared (usually with the var statement), then JavaScript assumes that the variable was global. This can mask misspelled names and other problems.

JSLint expects that all variables and functions are declared before they are used or invoked. This allows it to detect implied global variables. It is also good practice because it makes programs easier to read.

Sometimes a file is dependent on global variables and functions that are defined elsewhere. You can identify these to JSLint by including a comment in your file that lists the global functions and objects that your program depends on, but that are not defined in your program or script file.

A global declaration can look like this:

```
/*global getElementByAttribute, breakCycles, hanoi */
```

A global declaration starts with `/*global`. Notice that there is no space before the `g`. You can have as many `/*global` comments as you like. They must appear before the use of the variables they specify.

Some globals can be predefined for you. Select the *Assume a browser* (browser) option (see Options below) to predefine the standard global properties that are supplied by web browsers, such as `window` and `document` and `alert`. Select the *Assume Rhino* (rhino) option to predefine the global properties provided by the Rhino environment. Select the *Assume a Yahoo Widget* (widget) option to predefine the global properties provided by the Yahoo! Widgets environment.

## Members

Since JavaScript is a loosely-typed, dynamic-object language, it is not possible to determine at compile time if property names are spelled correctly. JSLint provides some assistance with this.

At the bottom of its report, JSLint displays a `/*members*/` comment. It contains all of the names and string literals that were used with dot notation, subscript notation, and object literals to name the members of objects. You can look through the list for misspellings. Member names that were only used once are shown in italics. This is to make misspellings easier to spot.

You can copy the `/*members*/` comment into your script file. JSLint will check the spelling of all property names against the list. That way, you can have JSLint look for misspellings for you.

## Semicolon

JavaScript uses a C-like syntax which requires the use of semicolons to delimit statements. JavaScript attempts to make semicolons optional with a semicolon insertion mechanism. This is dangerous.

Like C, JavaScript has `++` and `--` and `()` operators which can be prefixes or suffixes. The disambiguation is done by the semicolon.

In JavaScript, a linefeed can be whitespace or it can act as a semicolon. This replaces one ambiguity with another.

JSLint expects that every statement be followed by `;` except for `for`, `function`, `if`, `switch`, `try`, and `while`. JSLint does not expect to see unnecessary semicolons or the empty statement.

## Line Breaking

As a further defense against the semicolon insertion mechanism, JSLint expects long statements to be broken only after one of these punctuation characters or operators:

```
, . ; : { } ( [ = < > ? ! + - * / % ~ ^ | &
== != <= >= += -= *= /= %= ^= |= &= << >> || &&
=== !== <<= >>= >>> >>>=
```

JSLint does not expect to see a long statement broken after an identifier, a string, a number, closer, or a suffix operator:



) ] ++ --

JSLint allows you to turn on the *Tolerate sloppy line breaking* (laxbreak) option.

Semicolon insertion can mask copy/paste errors. If you always break lines after operators, then JSLint can do better at finding them.

## Comma

The comma operator can lead to excessively tricky expressions. It can also mask some programming errors.

JSLint expects to see the comma used as a separator, but not as an operator (except in the initialization and incrementation parts of the for statement). It does not expect to see elided elements in array literals. Extra commas should not be used. A comma should not appear after the last element of an array literal or object literal because it can be misinterpreted by some browsers.

## Required Blocks

JSLint expects that if and for statements will be made with blocks {that is, with statements enclosed in braces}.

JavaScript allows an if to be written like this:

```
if (condition)  
    statement;
```

That form is known to contribute to mistakes in projects where many programmers are working on the same code. That is why JSLint expects the use of a block:

```
if (condition) {  
    statements;  
}
```

Experience shows that this form is more resilient.

## Forbidden Blocks

In many languages, a block introduces a scope. Variables introduced in a block are not visible outside of the block.

In JavaScript, blocks do not introduce a scope. There is only function-scope. A variable introduced anywhere in a function is visible everywhere in the function. JavaScript's blocks confuse experienced programmers and lead to errors because the familiar syntax makes a false promise.

JSLint expects blocks with function, if, switch, while, for, do, and try statements and nowhere else. An exception is made for an unblocked if statement on an else or for in.

## ***Expression Statements***

An expression statement is expected to be an assignment or a function/method call or delete. All other expression statements are considered to be errors.

### ***for in***

The for in statement allows for looping through the names of all of the properties of an object. Unfortunately, it also loops through all of the members which were inherited through the prototype chain. This has the bad side effect of serving up method functions when the interest is in data members.

The body of every for in statement should be wrapped in an if statement that does filtering. It can select for a particular type or range of values, or it can exclude functions, or it can exclude properties from the prototype. For example,

```
for (name in object) {  
    if (object.hasOwnProperty(name)) {  
        ....  
    }  
}
```

### ***switch***

A [common error](#) in switch statements is to forget to place a break statement after each case, resulting in unintended fall-through. JSLint expects that the statement before the next case or default is one of these: break, return, or throw.

### ***var***

JavaScript allows var definitions to occur anywhere within a function. JSLint is more strict.

JSLint expects that a var will be declared only once, and that it will be declared before it is used.

JSLint expects that a function will be declared before it is used.

JSLint expects that parameters will not also be declared as vars.

JSLint does not expect the arguments array to be declared as a var.

JSLint does not expect that a var will be defined in a block. This is because JavaScript blocks do not have block scope. This can have unexpected consequences. Define all variables at the top of the function.

### ***with***

The with statement was intended to provide a shorthand in accessing members in deeply nested objects. Unfortunately, it behaves [very badly](#) when setting new members. Never use the with statement. Use a var instead.

JSLint does not expect to see a with statement.

**=**

JSLint does not expect to see an assignment statement in the condition part of an if or for or while or do statement. This is because it is more likely that

```
if (a = b) {  
    ...  
}
```

was intended to be

```
if (a == b) {  
    ...  
}
```

If you really intend an assignment, wrap it in another set of parens:

```
if ((a = b)) {  
    ...  
}
```

**== and !=**

The == and != operators do type coercion before comparing. This is bad because it causes '0' == 0 to be true. This can mask type errors.

When comparing to any of the following values, use the === or !== operators, which do not do type coercion.

0 " undefined null false true

If you want the type coercion, then use the short form. Instead of

```
(foo != 0)
```

just say

```
(foo)
```

and instead of

```
(foo == 0)
```

say

```
(!foo)
```

The === and !== operators are preferred. There is a *Disallow == and !=* (eqeqeq) option which requires the use of === and !== in all cases.

## Labels

JavaScript allows any statement to have a label, and labels have a separate name space. JSLint is more strict.

JSLint expects labels only on statements that interact with break: switch, while, do, and for. JSLint expects that labels will be distinct from vars and parameters.

## ***Unreachable Code***

JSLint expects that a return, break, continue, or throw statement will be followed by a } or case or default.

## ***Confusing Pluses and Minuses***

JSLint expects that + will not be followed by + or ++, and that - will not be followed by - or --. A misplaced space can turn + + into ++, an error that is difficult to see. Use parens to avoid confusion..

### ***++ and --***

The ++ (increment) and -- (decrement) operators have been known to contribute to bad code by encouraging excessive trickiness. They are second only to faulty architecture in enabling to viruses and other security menaces. There is an option that prohibits the use of these operators.

## ***Bitwise Operators***

JavaScript does not have an integer type, but it does have bitwise operators. The bitwise operators convert their operands from floating point to integers and back, so they are not near as efficient as in C or other languages. They are rarely useful in browser applications. The similarity to the logical operators can mask some programming errors. There is an option that prohibits the use of these operators.

### ***eval is evil***

The eval function (and its relatives, Function, setTimeout, and setInterval) provide access to the JavaScript compiler. This is sometimes necessary, but in most cases it indicates the presence of extremely bad coding. The eval function is the most misused feature of JavaScript.

### ***void***

In most C-like languages, void is a type. In JavaScript, void is a prefix operator that always returns undefined. JSLint does not expect to see void because it is confusing and not very useful.

## ***Regular Expressions***

Regular expressions are written in a terse and cryptic notation. JSLint looks for problems that may cause portability problems. It also attempts to resolve visual ambiguities by recommending explicit escapement.

JavaScript's syntax for regular expression literals overloads the / character. To avoid ambiguity, JSLint expects that the character preceding a regular expression literal is a ( or = or : or , character.

## **Constructors and *new***

Constructors are functions that are designed to be used with the new prefix. The new prefix creates a new object based on the function's prototype, and binds that object to the function's implied this parameter. If you neglect to use the new prefix, no new object will be made and this will be bound to the global object. This is a [serious mistake](#).

JSLint enforces the convention that constructor functions be given names with initial uppercase. JSLint does not expect to see a function invocation with an initial uppercase name unless it has the new prefix. JSLint does not expect to see the new prefix used with functions whose names do not start with initial uppercase. This can be controlled with an option.

JSLint does not expect to see the wrapper forms new Number, new String, new Boolean.

JSLint does not expect to see new Object (use {} instead).

JSLint does not expect to see new Array (use [] instead).

## **Unsafe Characters**

There are characters that are handled inconsistently in the various implementations, and so must be escaped when placed in strings.

```
\u0000-\u001f
\u007f-\u009f
\u00ad
\u0600-\u0604
\u070f
\u17b4
\u17b5
\u200c-\u200f
\u2028-\u202f
\u2060-\u206f
\ueff
\u2060-\u206f
```

## **Not Looked For**

JSLint does not do flow analysis to determine that variables are assigned values before used. This is because variables are given a value (undefined) which is a reasonable default for many applications.

JSLint does not do any kind of global analysis. It does not attempt to determine that functions used with new are really constructors (except by enforcing capitalization conventions), or that method names are spelled correctly.

## **HTML**

JSLint is able to handle HTML text. It can inspect the JavaScript content contained within <script>...</script> tags. It also inspects the HTML content, looking for problems that are known to interfere with JavaScript:

- All tag names must be in lower case.
- All tags that can take a close tag (such as </p>) must have a close tag.

- All tags are correctly nested.
- The entity &lt; must be used for literal '<'.

JSLint is less anal than the sycophantic conformity demanded by XHTML, but more strict than the popular browsers.

JSLint also checks for the occurrence of '</' in string literals. You should always write '<\/' instead. The extra backslash is ignored by the JavaScript compiler but not by the HTML parser. Tricks like this should not be necessary, and yet they are.

There is an option that allows use of upper case tagnames. There is also an option that allows the use of inline HTML event handlers.

## CSS

JSLint can inspect CSS files. It expects the first line of a CSS file to be

```
@charset "UTF-8";
```

This feature is experimental. Please report any problems or limitations. There is an option that will tolerate some of the non-standard-but-customary workarounds.

## Report

If JSLint is able to complete its scan, it generates a function report. It lists for each function:

- The line number on which it starts.
- Its name. In the case of anonymous functions, JSLint will "guess" the name.
- The parameters.
- *Closure*: The variables and parameters that are declared in the function that are used by its inner functions.
- *Variables*: The variables that are declared in the function that are used only by the function.
- *Unused*: The variables that are declared in the function that are not used. This may be an indication of an error.
- *Outer*: Variables used by this function that are declared in another function.
- *Global*: Global variables that are used by this function.
- *Label*: Statement labels that are used by this function.