Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks like this: ☰️. You may toggle all summaries with the big arrow button:

### Table of Contents

<table>
<thead>
<tr>
<th>JavaScript Language Rules</th>
<th>var</th>
<th>Constants</th>
<th>Semicolons</th>
<th>Nested functions</th>
<th>Function Declarations Within Blocks</th>
<th>Exceptions</th>
<th>Custom exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JavaScript Style Rules</th>
<th>Naming</th>
<th>Custom toString() methods</th>
<th>Deferred initialization</th>
<th>Explicit scope</th>
<th>Code formatting</th>
<th>Parentheses</th>
<th>Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tips and Tricks</th>
<th>Wrappers for primitive types</th>
<th>Multi-level prototype hierarchies</th>
<th>Method definitions</th>
<th>Closures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Important Note

**Displaying Hidden Details in this Guide**

This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. Click it now. You should see "Hooray" appear below.

Hooray! Now you know you can expand points to get more details. Alternatively, there's a "toggle all" at the top of this document.

### Background

JavaScript is the main client-side scripting language used by many of Google's open-source projects. This style guide is a list of dos and don'ts for JavaScript programs.

### JavaScript Language Rules

**var**

- Declarations with `var`: Always

  **Decision:**

  When you fail to specify `var`, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable lives (e.g., it could be in the Document or Window just as easily as in the local scope). So always declare with `var`.

**Constants**

- Use `NAMES LIKE THIS` for constants. Use `@const` where appropriate. Never use the `const` keyword.

  **Decision:**

  For simple primitive value constants, the naming convention is enough.

  ```javascript
  /**
   * The number of seconds in a minute.
   * @type {number}
   */
  goog.example.SECONDS_IN_A_MINUTE = 60;
  ```

  For non-primitives, use the `@const` annotation.

  ```javascript
  /**
   * The number of seconds in each of the given units.
   * @type {Object.<number>}
   * @const
   */
  goog.example.SECONDS_TABLE = {
    minute: 60,
    hour: 60 * 60,
    day: 60 * 60 * 24
  }
  ```

  This allows the compiler to enforce constant-ness.

  As for the `const` keyword, Internet Explorer doesn't parse it, so don't use it.

### Semicolons
Always use semicolons.

Relying on implicit insertion can cause subtle, hard to debug problems. Don't do it. You're better than that.

There are a couple places where missing semicolons are particularly dangerous:

```javascript
// 1. MyClass.prototype.myMethod = function() {
  return 42;
}  // No semicolon here.

{}  // Some initialization code wrapped in a function to create a scope for locals.
}();

var x = {
  'i': 1,
  'j': 2
}  // No semicolon here.

// 2. Trying to do one thing on Internet Explorer and another on Firefox.
// I know you'd never write code like this, but throw me a bone.
[normalVersion, ffVersion][isIE]();

var THINGS_TO_EAT = [apples, oysters, sprayOnCheese]  // No semicolon here.

// 3. conditional execution a la bash
-1 === resultOfOperation() || die();
```

So what happens?

1. JavaScript error - first the function returning 42 is called with the second function as a parameter, then the number 42 is "called" resulting in an error.
2. You will most likely get a 'no such property in undefined' error at runtime as it tries to call x[ffVersion][isIE]().
3. die is called unless resultOfOperation() is NaN and THINGS_TO_EAT gets assigned the result of die().

Why?

JavaScript requires statements to end with a semicolon, except when it thinks it can safely infer their existence. In each of these examples, a function declaration or object or array literal is used inside a statement. The closing brackets are not enough to signal the end of the statement. Javascript never ends a statement if the next token is an infix or bracket operator.

This has really surprised people, so make sure your assignments end with semicolons.

**Nested functions**

Yes

Nested functions can be very useful, for example in the creation of continuations and for the task of hiding helper functions. Feel free to use them.

**Function Declarations Within Blocks**

No

Do not do this:

```javascript
if (x) {
  function foo() {}
}
```

While most script engines support Function Declarations within blocks it is not part of ECMAScript (see ECMA-262, clause 13 and 14). Worse implementations are inconsistent with each other and with future EcmaScript proposals. ECMAScript only allows for Function Declarations in the root statement list of a script or function. Instead use a variable initialized with a Function Expression to define a function within a block:

```javascript
if (x) {
  var foo = function() {}
}
```

**Exceptions**

Yes

You basically can't avoid exceptions if you're doing something non-trivial (using an application development framework, etc.). Go for it.

**Custom exceptions**

Yes

Without custom exceptions, returning error information from a function that also returns a value can be tricky, not to mention inelegant. Bad solutions include passing in a reference type to hold error information or always returning Objects with a potential error member. These basically amount to a primitive exception handling hack. Feel free to use custom exceptions when appropriate.

**Standards features**

Always preferred over non-standards features

For maximum portability and compatibility, always prefer standards features over non-standards features (e.g., `string.charAt(3)` over `string[3]`) and element access with DOM functions instead of using an application-specific shorthand.

**Wrapper objects for primitive types**

No

There's no reason to use wrapper objects for primitive types, plus they're dangerous:
var x = new Boolean(false);
if (x) {
    alert('hi'); // Shows 'hi'.
}

Don't do it!

However type casting is fine.

```javascript
var x = Boolean(0);
if (x) {
    alert('hi'); // This will never be alerted.
}
typeof Boolean(0) == 'boolean';
typeof new Boolean(0) == 'object';
```

This is very useful for casting things to number, string and boolean.

**Multi-level prototype hierarchies**

Not preferred

Multi-level prototype hierarchies are how JavaScript implements inheritance. You have a multi-level hierarchy if you have a user-defined class D with another user-defined class B as its prototype. These hierarchies are much harder to get right than they first appear!

For that reason, it is best to use goog.inherits() from the Closure Library or something similar.

```javascript
function D() {
    goog.base(this);
    goog.inherits(D, B);
    D.prototype.method = function() {
        ...
    };
}
```

**Method definitions**

Yes, but be careful.

While there are several methods for attaching methods and properties to a constructor, the preferred style is:

```javascript
Foo.prototype.bar = function() { /* ... */);
```

**Closures**

Yes, but be careful.

The ability to create closures is perhaps the most useful and often overlooked feature of JS. Here is a good description of how closures work.

One thing to keep in mind, however, is that a closure keeps a pointer to its enclosing scope. As a result, attaching a closure to a DOM element can create a circular reference and thus, a memory leak. For example, in the following code:

```javascript
function foo(element, a, b) {
    element.onclick = function() { /* uses a and b */};
}
```

the function closure keeps a reference to `element`, `a`, and `b` even if it never uses `element`. Since `element` also keeps a reference to the closure, we have a cycle that won't be cleaned up by garbage collection. In these situations, the code can be structured as follows:

```javascript
function foo(element, a, b) {
    element.onclick = bar(a, b);
}
function bar(a, b) {
    return function() { /* uses a and b */}
}
```

**eval()**

Only for deserialization (e.g. evaluating RPC responses)

`eval()` makes for confusing semantics and is dangerous to use if the string being `eval()`d contains user input. There's usually a better, more clear safer way to write your code, so its use is generally not permitted. However `eval()` makes deserialization considerably easier than the non-`eval` alternatives, so its use is acceptable for this task (for example, to evaluate RPC responses).

Deserialization is the process of transforming a series of bytes into an in-memory data structure. For example, you might write objects out to a file as:

```javascript
users = [
    {name: 'Eric',
     id: 37824,
     email: 'jellyvore@myway.com'},
    {name: 'xtof',
     id: 31337,
     email: 'b4dd55h4x0r@google.com'},
...,
];
```

Reading these data back into memory is as simple as `eval()`ing the string representation of the file.
Similarly, eval() can simplify decoding RPC return values. For example, you might use an XMLHttpRequest to make an RPC, and in its response the server can return JavaScript:

```javascript
var userOnline = false;
var user = 'nusrat';
var xmlhttp = new XMLHttpRequest();
xmlhttp.open('GET', 'http://chat.google.com/isUserOnline?user' + user, false);
xmlhttp.send('');
// Server returns:
// userOnline = true;
if (xmlhttp.status == 200) {
    eval(xmlhttp.responseText);
}
// userOnline is now true.
```

**with()**

- **V No** Using with clouds the semantics of your program. Because the object of the with can have properties that collide with local variables, it can drastically change the meaning of your program. For example, what does this do?

```javascript
with (foo) {
    var x = 3;
    return x;
}
```

Answer: anything. The local variable x could be clobbered by a property of foo and perhaps it even has a setter, in which case assigning 3 could cause lots of other code to execute. Don’t use with.

**this**

- **V Only in object constructors, methods, and in setting up closures**

The semantics of this can be tricky. At times it refers to the global object (in most places), the scope of the caller (in eval), a node in the DOM tree (when attached using an event handler HTML attribute), a newly created object (in a constructor), or some other object (if function was called or applied).

Because this is so easy to get wrong, limit its use to those places where it is required:
- in constructors
- in methods of objects (including in the creation of closures)

**for-in loop**

- **V Only for iterating over keys in an object/map/hash**

For-in loops are often incorrectly used to loop over the elements in an Array. This is however very error prone because it does not loop from 0 to length - 1 but over all the present keys in the object and its prototype chain. Here are a few cases where it fails:

```javascript
function printArray(arr) {
    for (var key in arr) {
        print(arr[key]);
    }
}
printArray([0,1,2,3]);  // This works.
var a = new Array(10);
printArray(a);  // This is wrong.
a = document.getElementsByTagName('*');
printArray(a);  // This is wrong.
a = [0,1,2,3];
a.bubu = 'wine';
printArray(a);  // This is wrong again.
a = new Array;
a[3] = 3;
printArray(a);  // This is wrong again.
```

Always use normal for loops when using arrays.

```javascript
function printArray(arr) {
    var i = arr.length;
    for (var i = 0; i < l; i++) {
        print(arr[i]);
    }
}
```

**Associative Arrays**

- **V Never use Array as a map/hash/associative array**

Associative Arrays are not allowed... or more precisely you are not allowed to use non number indexes for arrays. If you need a map/hash use Object instead of Array in these cases because the features that you want are actually features of Object and not of Array. Array just happens to extend Object (like any other object in JS and therefore you might as well have used Date, RegExp or String).

**Multiline string literals**

- **V No**

Do not do this:
The whitespace at the beginning of each line can't be safely stripped at compile time; whitespace after the slash will result in tricky errors; and while most script engines support this, it is not part of ECMAScript.

Use string concatenation instead:

```javascript
var myString = 'A rather long string of English text, an error message \ 
    actually that just keeps going and going -- an error \ 
    message to make the Energizer bunny blush (right through \ 
    those Schwarzenegger shades) Where was I? Oh yes, \ 
    you've got an error and all the extraneous whitespace is \ 
    just gravy. Have a nice day.';
```

### Array and Object literals

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Array and Object literals instead of Array and Object constructors.</td>
<td></td>
</tr>
</tbody>
</table>

Array constructors are error-prone due to their arguments.

```javascript
// Length is 3.
var a1 = new Array(x1, x2, x3);
// Length is 2.
var a2 = new Array(x1, x2);
// If x1 is a number and it is a natural number the length will be x1.
// If x1 is a number but not a natural number this will throw an exception.
// Otherwise the array will have one element with x1 as its value.
var a3 = new Array(x1);  // Length is 0.
var a4 = new Array();
```

Because of this, if someone changes the code to pass 1 argument instead of 2 arguments, the array might not have the expected length.

To avoid these kinds of weird cases, always use the more readable array literal.

```javascript
var a = [x1, x2, x3];
var a2 = [x1, x2];
var a3 = [x1];
var a4 = [];
```

Object constructors don't have the same problems, but for readability and consistency object literals should be used.

```javascript
var o = new Object();
var o2 = new Object();
o2.a = 0;
o2.b = 1;
o2.c = 2;
o2['strange key'] = 3;
```

Should be written as:

```javascript
var o = {};
var o2 = {
    a: 0,
    b: 1,
    c: 2,
    'strange key': 3
};
```

### Modifying prototypes of builtin objects

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifying builtins like Object.prototype and Array.prototype are strictly forbidden. Modifying other builtins like Function.prototype is less dangerous but still leads to hard to debug issues in production and should be avoided.</td>
<td></td>
</tr>
</tbody>
</table>

### Internet Explorer's Conditional Comments

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don't do this:</td>
<td></td>
</tr>
</tbody>
</table>

```javascript
var f = function () {
    /*@cc_on if (@_jscript) { return 2* 8*/ 3; /*@ } */ 8*/
};
```

Conditional Comments hinder automated tools as they can vary the JavaScript syntax tree at runtime.

### JavaScript Style Rules

#### Naming
In general, use functionNamesLikeThis, variableNamesLikeThis, ClassNamesLikeThis, EnumNamesLikeThis, methodNamesLikeThis, and SYMBOLIC_CONSTANTS_LIKE_THIS.

Properties and methods

- Private properties, variables, and methods (in files or classes) should be named with a trailing underscore.
- Protected properties, variables, and methods should be named without a trailing underscore (like public ones).

For more information on private and protected, read the section on visibility.

Method and function parameter

Optional function arguments start with opt_.

Functions that take a variable number of arguments should have the last argument named var_args. You may not refer to var_args in the code; use the arguments array.

Optional and variable arguments can also be specified in @param annotations. Although either convention is acceptable to the compiler, using both together is preferred.

_getters and Setters

EcmaScript 5 getters and setters for properties are discouraged. However, if they are used, then getters must not change observable state.

```javascript
/**
 * WRONG -- Do NOT do this.
 */
var foo = { get next() { return this.nextId++; } };
```

Accessor functions

Getters and setters methods for properties are not required. However, if they are used, then getters must be named getFoo() and setters must be named setFoo(value). (For boolean getters, isFoo() is also acceptable, and often sounds more natural.)

Namespaces

JavaScript has no inherent packaging or namespacing support.

Global name conflicts are difficult to debug, and can cause intractable problems when two projects try to integrate. In order to make it possible to share common JavaScript code, we've adopted conventions to prevent collisions.

Use namespaces for global code

ALWAYS prefix identifiers in the global scope with a unique pseudo namespace related to the project or library. If you are working on "Project Sloth", a reasonable pseudo namespace would be sloth.*.

```javascript
var sloth = {);
    sloth.sleep = function() {
        ...
    };
```

Many JavaScript libraries, including the Closure Library and Dojo toolkit give you high-level functions for declaring your namespaces. Be consistent about how you declare your namespaces.

```javascript
goog.provide('sloth');
    sloth.sleep = function() {
        ...
    };
```

Respect namespace ownership

When choosing a child-namespace, make sure that the owners of the parent namespace know what you are doing. If you start a project that creates hats for sloths, make sure that the Sloth team knows that you're using sloth.hats.

Use different namespaces for external code and internal code

"External code" is code that comes from outside your codebase, and is compiled independently. Internal and external names should be kept strictly separate. If you're using an external library that makes things available in foo.hats.*, your internal code should not define all its symbols in foo.hats.*, because it will break if the other team defines new symbols.

```javascript
foo.require('foo.hats');
/**
 *  WRONG -- Do NOT do this.
 * @constructor
 * @extend {foo.hats.RoundHat}
 */
foo.hats.BowlerHat = function() {
    ...
};
```

If you need to define new APIs on an external namespace, then you should explicitly export the public API functions, and only those functions. Your internal code should call the internal APIs by their internal names, for consistency and so that the compiler can optimize them better.

```javascript
foo.provide('googleyhats.BowlerHat');
foo.require('foo.hats');
/**
 * @constructor
 * @extend {foo.hats.RoundHat}
 */
googleyhats.BowlerHat = function() {
    ...
};
google.exportSymbol('foo.hats.BowlerHat', googleyhats.BowlerHat);
```

Alias long type names to improve readability
Use local aliases for fully-qualified types if doing so improves readability. The name of a local alias should match the last part of the type.

```javascript
/** *
* @constructor
* some.long.namespace.MyClass = function() {
* }
/**
* @param {some.long.namespace.MyClass} a
*/
some.long.namespace.MyClass.staticHelper = function(a) {
  ...
};

myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  var staticHelper = some.long.namespace.MyClass.staticHelper;
  staticHelper(new MyClass());
};
```

Do not alias namespaces.

```javascript
myapp.main = function() {
  var namespace = some.long.namespace;
  namespace.MyClass.staticHelper(new namespace.MyClass());
};
```

Avoid accessing properties of an aliased type, unless it is an enum.

```javascript
/** *
* @enum {string} *
*/
some.long.namespace.Fruit = {
  APPLE: 'a',
  BANANA: 'b'
};

myapp.main = function() {
  var Fruit = some.long.namespace.Fruit;
  switch (Fruit) {
    case Fruit.APPLE:
      ...
    case Fruit.BANANA:
      ...
  }
};
```

Never create aliases in the global scope. Use them only in function blocks.

### Filenames

Filenames should be all lowercase in order to avoid confusion on case-sensitive platforms. Filenames should end in `.js`, and should contain no punctuation except for `-` or `_` (prefer `-` to `_`).

### Custom toString() methods

Must always succeed without side effects.

You can control how your objects string-ify themselves by defining a custom `toString()` method. This is fine, but you need to ensure that your method (1) always succeeds and (2) does not have side-effects. If your method doesn't meet these criteria, it's very easy to run into serious problems. For example, if `toString()` calls a method that does an `assert`, `assert` might try to output the name of the object in which it failed, which of course requires calling `toString()`.

### Deferred initialization

OK

It isn't always possible to initialize variables at the point of declaration, so deferred initialization is fine.

### Explicit scope

Always

Always use explicit scope - doing so increases portability and clarity. For example, don't rely on `window` being in the scope chain. You might want to use your function in another application for which `window` is not the content window.

### Code formatting

Expand for more information.

We follow the [C++ formatting rules](https://en.wikipedia.org/wiki/C%2B%2B_formatting_rules) in spirit, with the following additional clarifications.

### Curly Braces

Because of implicit semicolon insertion, always start your curly braces on the same line as whatever they're opening. For example:

```javascript
if (something) {
  // ...
} else {
  // ...
}
```
Array and Object Initializers

Single-line array and object initializers are allowed when they fit on a line:

```javascript
var arr = [1, 2, 3]; // No space after { or before }.
var obj = {a: 1, b: 2, c: 3}; // No space after { or before }.
```

Multiline array initializers and object initializers are indented 2 spaces, just like blocks.

```javascript
// Object initializer.
var inset = {
    top: 10,
    right: 20,
    bottom: 15,
    left: 12
};

// Array initializer.
this.rows = [
    "'Larriibartfast' <fjordmaster@magrathea.com'>,
    "'Zaphod Beeblebrox' <theprez@universe.gov'>,
    "'Ford Prefect' <ford@thequide.com'>,
    "'Arthur Dent' <has.no.tea@gmail.com'>,
    "'Marvin the Paranoid Android' <marvin@googlemail.com'>,
    'the.mice@magrathea.com'
];

// Used in a method call.
goog.dom.createDom(goog.dom.TagName.DIV, {
    id: 'foo',
    className: 'some-csa-class',
    style: 'display:none'
}, 'Hello, world!');
```

Long identifiers or values present problems for aligned initialization lists, so always prefer non-aligned initialization. For example:

```javascript
CORRECT_Object.prototype = {
    a: 0,
    b: 1,
    lengthyName: 2
};

Not like this:

```javascript
WRONG_Object.prototype = {
    a : 0,
    b : 1,
    lengthyName : 2
};
```

Function Arguments

When possible, all function arguments should be listed on the same line. If doing so would exceed the 80-column limit, the arguments must be line-wrapped in a readable way. To save space, you may wrap as close to 80 as possible, or put each argument on its own line to enhance readability. The indentation may be either four spaces, or aligned to the parenthesis. Below are the most common patterns for argument wrapping:

```javascript
// Four-space, wrap at 80. Works with very long function names, survives
// renaming without reindenting, low on space.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableViewModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
};

// Four-space, one argument per line. Works with long function names,
// survives renaming, and emphasizes each argument.
goog.foo.bar.doThingThatIsVeryDifficultToExplain = function(
    veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableViewModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
};

// Parenthesis-aligned indentation, wrap at 80. Visually groups arguments,
// low on space.
function foo(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableViewModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
}

// Parenthesis-aligned, one argument per line. Visually groups and
// emphasizes each individual argument.
function bar(veryDescriptiveArgumentNumberOne, veryDescriptiveArgumentTwo,
    tableViewModelEventHandlerProxy, artichokeDescriptorAdapterIterator) {
    // ...
}
```

When the function call is itself indented, you're free to start the 4-space indent relative to the beginning of the original statement or relative to the beginning of the current function call. The following are all acceptable indentation styles:

```javascript
if (veryLongFunctionNameA(
    veryLongArgumentName) ||
    veryLongFunctionNameB(
    veryLongArgumentName)) {
    veryLongFunctionNameC(veryLongFunctionNameD(}
Passing Anonymous Functions

When declaring an anonymous function in the list of arguments for a function call, the body of the function is indented two spaces from the left edge of the statement, or two spaces from the left edge of the function keyword. This is to make the body of the anonymous function easier to read (i.e. not be all squished up into the right half of the screen).

```
veryLongFunctionName(
    veryLongFunctionNameF));
```

More Indentation

In fact, except for array and object initializers, and passing anonymous functions, all wrapped lines should be indented either left-aligned to the expression above, or indented four spaces, not indented two spaces.

```
someWonderfulHtml = ' ' +
    getEveryMoreHtml(someReallyInterestingValues, moreValues, 
    evenMoreParams, 'a duck', true, 72, 
    slightlyMoreMonkeys(0xffffffff) + 
    ' ';
```

Blank lines

Use newlines to group logically related pieces of code. For example:

```
doSomethingTo(x);
doSOMethingElseTo(x);
andThen(x);
nowDoSomethingWith(y);
andnowWith(z);
```

Binary and Ternary Operators

Always put the operator on the preceding line, so that you don't have to think about implicit semi-colon insertion issues. Otherwise, line breaks and indentation follow the same rules as in other Google style guides.

```
var x = a ? b : c; // All on one line if it will fit.
// Indentation +4 is OK.
var y = a 
    longButSimpleOperand1 : longButSimpleOperandC;
// Indent to the line position of the first operand is also OK.
var z = a 
    moreComplicatedB : 
    moreComplicatedC;
```

Parentheses

- **Only where required**
  - Use sparingly and in general only where required by the syntax and semantics.
  - Never use parentheses for unary operators such as `delete`, `typeof` and `void` or after keywords such as `return`, `throw` as well as others (`case`, `in` or `new`).

Strings

- **Prefer ' over "**
  - For consistency single-quotes (') are preferred to double-quotes ("). This is helpful when creating strings that include HTML.

```
var msg = 'This is some HTML';
```
Visibility (private and protected fields)

Encouraged, use JSDoc annotations `@private` and `@protected`.

We recommend the use of the JSDoc annotations `@private` and `@protected` to indicate visibility levels for classes, functions, and properties.

The `--jscomp_warning=visibility` compiler flag turns on compiler warnings for visibility violations. See [Closure Compiler Warnings](http://closure-library.googlecode.com/). `@private` global variables and functions are only accessible to code in the same file.

Constructors marked `@private` may only be instantiated by code in the same file and by their static and instance members. `@private` constructors may also be accessed anywhere in the same file for their public static properties and by the `instanceof` operator.

Global variables, functions, and constructors should never be annotated `@protected`.

```javascript
// File 1.
//@private
@constructor
AA_PrivateClass_ = function() {
}

//@protected
function AA_init_() {
    return new AA_PrivateClass_();
}
AA_init_();

@private properties are accessible to all code in the same file, plus all static methods and instance methods of that class that "owns" the property, if the property belongs to a class. They cannot be accessed or overridden from a subclass in a different file.

@protected properties are accessible to all code in the same file, plus any static methods and instance methods of any subclass of a class that "owns" the property.

Note that these semantics differ from those of C++ and Java, in that they grant private and protected access to all code in the same file, not just in the same class or class hierarchy. Also, unlike in C++, private properties cannot be overriden by a subclass.

```javascript
// File 1.
//@constructor
AA_PublicClass = function() {
}

//@private
AA_PublicClass.staticPrivateProp_ = 1;
//@private
AA_PublicClass.prototype.privateProp_ = 2;
//@protected
AA_PublicClass.staticProtectedProp = 3;
//@protected
AA_PublicClass.prototype.protectedProp = 4;

// File 2.
//@return {number} The number of ducks we've arranged in a row.
AA_PublicClass.prototype.method = function() {
    // Legal accesses of these two properties.
    return this.privateProp_ + AA_PublicClass.staticPrivateProp_;
}

// File 3.
//@constructor
//@extends {AA_PublicClass}
AA_SubClass = function() {
    // Legal access of a protected static property.
    AA_PublicClass.staticProtectedProp = this.method();
    goog.inherits(AA_SubClass, AA_PublicClass);
    // @return {number} The number of ducks we've arranged in a row.
    AA_SubClass.prototype.method = function() {
        // Legal access of a protected instance property.
        return this.protectedProp;
    }
}

Notice that in JavaScript, there is no distinction between a type (like `AA_PrivateClass`) and the constructor for that type. There is no way to express both that a type is public and its constructor is private (because the constructor could easily be aliased in a way that would defeat the privacy check).

**JavaScript Types**

Encouraged and enforced by the compiler.

When documenting a type in JSDoc, be as specific and accurate as possible. The types we support are JS2 style types and JS1.x types.
The JavaScript Type Language

The JS2 proposal contained a language for specifying JavaScript types. We use this language in JsDoc to express the types of function parameters and return values.

As the JS2 proposal has evolved, this language has changed. The compiler still supports old syntaxes for types, but those syntaxes are deprecated.

<table>
<thead>
<tr>
<th>Operator Name</th>
<th>Syntax</th>
<th>Description</th>
<th>Deprecated Syntaxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Name</td>
<td>(boolean), (Window), (goog.ui.Menu)</td>
<td>Simply the name of a type.</td>
<td></td>
</tr>
<tr>
<td>Type Application</td>
<td>(Array.&lt;string&gt;)</td>
<td>An array of strings.</td>
<td></td>
</tr>
<tr>
<td>Type Application</td>
<td>(Object.&lt;string, number&gt;)</td>
<td>An object in which the keys are strings and the values are numbers.</td>
<td></td>
</tr>
<tr>
<td>Type Union</td>
<td>(number</td>
<td>boolean)</td>
<td>A number or a boolean.</td>
</tr>
<tr>
<td>Record Type</td>
<td>{myNum: number, myObject}</td>
<td>An anonymous type with the given type members.</td>
<td></td>
</tr>
<tr>
<td>Nullable type</td>
<td>?number</td>
<td>A number or NULL.</td>
<td></td>
</tr>
<tr>
<td>Non-nullable type</td>
<td>!Object</td>
<td>An Object, but never the null value.</td>
<td></td>
</tr>
<tr>
<td>Function Type</td>
<td>function(string, boolean)</td>
<td>A function that takes two arguments (a string and a boolean), and has an unknown return value.</td>
<td></td>
</tr>
<tr>
<td>Function Return Type</td>
<td>function(); number</td>
<td>A function that takes no arguments and returns a number.</td>
<td></td>
</tr>
<tr>
<td>Function this Type</td>
<td>function(this:goog.ui.Menu, string)</td>
<td>A function that takes one argument (a string), and executes in the context of a goog.ui.Menu.</td>
<td></td>
</tr>
<tr>
<td>Function new Type</td>
<td>function(new:goog.ui.Menu, string)</td>
<td>A constructor that takes one argument (a string), and creates a new instance of goog.ui.Menu when called with the 'new' keyword.</td>
<td></td>
</tr>
<tr>
<td>Variable arguments</td>
<td>function(string, ...[number]: number)</td>
<td>A function that takes one argument (a string), and then a variable number of arguments that must be numbers.</td>
<td></td>
</tr>
<tr>
<td>Variable arguments (in @param annotations)</td>
<td>@param {...number} var_args</td>
<td>A variable number of arguments to an annotated function.</td>
<td></td>
</tr>
<tr>
<td>Function optional arguments</td>
<td>function(?string, number?)</td>
<td>A function that takes one optional, nullable string and one optional number as arguments. The ? syntax is only for function type declarations.</td>
<td></td>
</tr>
<tr>
<td>Function optional arguments (in @param annotations)</td>
<td>@param {number=} opt_argument</td>
<td>An optional parameter of type number.</td>
<td></td>
</tr>
<tr>
<td>The ALL type</td>
<td>(*)</td>
<td>Indicates that the variable can take on any type.</td>
<td></td>
</tr>
<tr>
<td>The UNKNOWN type</td>
<td>(?)</td>
<td>Indicates that the variable can take on any type, and the compiler should not type-check any uses of it.</td>
<td></td>
</tr>
</tbody>
</table>

Types in JavaScript

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Value Examples</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>1, 1.0, -5, NaN, Math.PI</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>new Number(true)</td>
<td>Number object</td>
</tr>
<tr>
<td>string</td>
<td>'Hello', 'World', String(42)</td>
<td>String value</td>
</tr>
<tr>
<td>String</td>
<td>new String('Hello'), new String(42)</td>
<td>String object</td>
</tr>
</tbody>
</table>
### Boolean (0)

- **true**
- **false**

**Boolean value**

### New Boolean(true)

**Boolean object**

### New RegExp('hello') /world/g

**RegExp**

### New Date

**Date**

### New Date()

- **null**
- **undefined**

- **void**

  - function f() {
    return;
  }

  **No return value**

### New Boolean()

**Boolean object**

### New RegExp()

**RegExp**

### New Date()

**Date**

- **null**
- **undefined**

### Array

- ['foo', 0.3, null]
- [ ]

- An Array of numbers

- Array of Arrays of strings

### Object

- {}
- {foo: 'abc', bar: 123, baz: null}

- An Object in which the values are strings.

### Object.<number,
string>

- var obj = {};
- obj[1] = 'bar';

**An Object in which the keys are numbers and the values are strings.**

**Note that in JavaScript, the keys are always implicitly coerced to strings, so obj['1'] == obj[1]. So the key will always be a string in for...in loops. But the compiler will verify the type if the key when indexing into the object.**

### Function

- function(x, y) {
  return x * y;
}

**Function object**

- function(number, number): number

**Function value**

### MyClass

```javascript
/** *
 * @constructor *
 */
function MyClass() {
  new project.MyClass();
}
```

**Enumeration**

### Element
document.createElement('div')

**Elements in the DOM.**

### Node
document.body.firstChild

**Nodes in the DOM.**

### HtmlInputElement

```javascript
htmlDocument.getElementsByTagName('input')[0]
```

**A specific type of DOM element.**

### Type Casts

In cases where type-checking doesn’t accurately infer the type of an expression, it is possible to add a type cast comment by adding a type annotation comment and enclosing the expression in parentheses. The parentheses are required, and may surround the type annotation comment as well.

```javascript
/** @type {number} */ (x)
/** @type {number} */ (x)
```

### Nullable vs. Optional Parameters and Properties

Because JavaScript is a loosely-typed language, it is very important to understand the subtle differences between optional, nullable, and undefined function parameters and class properties.

Object types (also known as reference types) are nullable by default. **NOTE:** Function types are not nullable by default. An object is defined as anything except a string, number, boolean, undefined, or null. For example, the following declaration

```javascript
/** *
 * Some class, initialized with a value.
 * @param {Object} value Some value.
 * @constructor
 */
function MyClass(value) {
```
*/
* Some value,
* @type {Object}
* @private
*/
this.myValue_ = value;
}

tells the compiler that the myValue_ property holds either an Object or null. If myValue_ must never be null, it should be declared like this:

/**
 * Some class, initialized with a non-null value.
 * @param {Object} value Some value.
 * @constructor
 */
function MyClass(value) {
    /**
     * Some value.
     * @type {Object}
     * @private
     */
    this.myValue_ = value;
}

This way, if the compiler can determine that somewhere in the code MyClass is initialized with a null value, it will issue a warning.

Optional parameters to functions may be undefined at runtime, so if they are assigned to class properties, those properties must be declared accordingly:

/**
 * Some class, initialized with an optional value.
 * @param {Object=} opt_value Some value (optional).
 * @constructor
 */
function MyClass(opt_value) {
    /**
     * Some value.
     * @type {Object|undefined}
     * @private
     */
    this.myValue_ = opt_value;
}

This tells the compiler that myValue_ may hold an Object, null, or remain undefined.

Note that the optional parameter opt_value is declared to be of type {Object=}, not {Object|undefined}. This is because optional parameters may, by definition, be undefined. While there is no harm in explicitly declaring an optional parameter as possibly undefined, it is both unnecessary and makes the code harder to read.

Finally, note that being nullable and being optional are orthogonal properties. The following four declarations are all different:

/**
 * Takes four arguments, two of which are nullable, and two of which are
 * optional.
 * @param {Object=} nonNull Mandatory (must not be undefined), must not be null.
 * @param {Object=} mayBeNull Mandatory (must not be undefined), may be null.
 * @param {Object=} opt_nonNull Optional (may be undefined), but if present,
 *    must not be null!
 * @param {Object=} opt_mayBeNull Optional (may be undefined), may be null.
 */
function strangeButTrue(nonNull, mayBeNull, opt_nonNull, opt_mayBeNull) {
    // ...
}

Typedefs

Sometimes types can get complicated. A function that accepts content for an Element might look like:

/**
 * @param {string} tagName
 * @param {{string|Element|Text|Array.<Element>|Array.<Text>}} contents
 * @return {{Element}}
*/
goog.createElement = function(tagName, contents) {
    ...
};

You can define commonly used type expressions with a @typedef tag. For example,

/** @typedef {{string|Element|Text|Array.<Element>|Array.<Text>}} */
goog.ElementContent;

/**
 * @param {string} tagName
 * @param {goog.ElementContent} contents
 * @return {{Element}}
*/
goog.createElement = function(tagName, contents) {
    ...
};

Template types

The compiler has limited support for template types. It can only infer the type of this inside an anonymous function literal from the type of the this argument and whether the this argument is missing.

/**
 * @param {function(this:T, ...)} fn
*/
* @param {T} thisObj
* @param {...*} var_args
* @template T
* 
* goog.bind = function(fn, thisObj, var_args) {
*   ...
* }
* // Possibly generates a missing property warning.
* goog.bind(function() { this.someProperty; }, new SomeClass());
* // Generates an undeclared this warning.
* goog.bind(function() { this.someProperty; });

Comments

Use JSDoc

We follow the C++ style for comments in spirit.

All files, classes, methods and properties should be documented with JSDoc comments.

Inline comments should be of the // variety.

Avoid sentence fragments. Start sentences with a properly capitalized word, and end them with punctuation.

Comment Syntax

The JSDoc syntax is based on JavaDoc. Many tools extract metadata from JSDoc comments to perform code validation and optimizations. These comments must be well-formed.

/**
 * A JSDoc comment should begin with a slash and 2 asterisks.
 * Inline tags should be enclosed in braces like {@code this}.
 * @desc Block tags should always start on their own line.
 */

JSDoc Indentation

If you have to line break a block tag, you should treat this as breaking a code statement and indent it four spaces.

/**
 * Illustrates line wrapping for long param/return descriptions.
 * @param {string} foo This is a param with a description too long to fit in
 *   one line.
 * @return {number} This returns something that has a description too long to
 *   fit in one line.
 */
* project.MyClass.prototype.method = function(foo) {
*   return 5;
* };

You should not indent the @fileoverview command.

Even though it is not preferred, it is also acceptable to line up the description.

/**
 * This is NOT the preferred indentation method.
 * @param {string} foo This is a param with a description too long to fit in
 *   one line.
 * @return {number} This returns something that has a description too long to
 *   fit in one line.
 */
* project.MyClass.prototype.method = function(foo) {
*   return 5;
* };

HTML in JSDoc

Like JavaDoc, JSDoc supports many HTML tags, like <code>, <pre>, <tt>, <strong>, <ul>, <ol>, <li>, <a>, and others.

This means that plaintext formatting is not respected. So, don’t rely on whitespace to format JSDoc:

/**
 * Computes weight based on three factors: items sent items received last timestamp
 */

It’ll come out like this:

Computes weight based on three factors: items sent items received items received

Instead, do this:

/**
 * Computes weight based on three factors:
 * <ul>
 *   <li>items sent</li>
 *   <li>items received</li>
 *   <li>last timestamp</li>
 * </ul>
 */

The JavaDoc style guide is a useful resource on how to write well-formed doc comments.

Top/ File-Level Comments

The top level comment is designed to orient readers unfamiliar with the code to what is in this file. It should provide a description of the file’s contents,
its author(s), and any dependencies or compatibility information. As an example:

```javascript
// Copyright 2009 Google Inc. All Rights Reserved.
/**
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 * @author user@google.com (Firstname Lastname)
 */

Class Comments

Classes must be documented with a description, and appropriate type tags.

```javascript
/**
 * Class making something fun and easy.
 * @param {string} arg1 An argument that makes this more interesting.
 * @param {Array.<number>} arg2 List of numbers to be processed.
 * @constructor
 * @extends {goog.Disposable}
 */
project.MyClass = function(arg1, arg2) {
  // ...
};
goog.inherits(project.MyClass, goog.Disposable);
```

Method and Function Comments

A description must be provided along with parameters. Method descriptions should start with a sentence written in the third person declarative voice.

```javascript
/**
 * Operates on an instance of MyClass and returns something.
 * @param {project.MyClass} obj Instance of MyClass which leads to a long
 * comment that needs to be wrapped to two lines.
 * @return {boolean} Whether something occurred.
 */
function PR_someMethod(obj) {
  // ...
}
```

For simple getters that take no parameters and have no side effects, the description can be omitted.

Property Comments

```javascript
/**
 * Maximum number of things per pane.
 * @type {number}
 */
project.MyClass.prototype.someProperty = 4;
```

JSDoc Tag Reference

<table>
<thead>
<tr>
<th>Tag</th>
<th>Template &amp; Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>@author</td>
<td>#author <a href="mailto:username@google.com">username@google.com</a> (first last)</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>/**</td>
</tr>
<tr>
<td></td>
<td>* @fileoverview Utilities for handling textareas.</td>
</tr>
<tr>
<td></td>
<td>* @author <a href="mailto:kuth@google.com">kuth@google.com</a> (Uthur Pendragon)</td>
</tr>
<tr>
<td></td>
<td>*/</td>
</tr>
<tr>
<td></td>
<td>Document the author of a file or the owner of a test, @fileoverview comment.</td>
</tr>
<tr>
<td>@code</td>
<td>{ @code ...}</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>/**</td>
</tr>
<tr>
<td></td>
<td>* Moves to the next position in the selection.</td>
</tr>
<tr>
<td></td>
<td>* Throws (@code goog.iter.StopIteration) when it</td>
</tr>
<tr>
<td></td>
<td>* passes the end of the range.</td>
</tr>
<tr>
<td></td>
<td>* @return {Node} The node at the next position.</td>
</tr>
<tr>
<td></td>
<td>*/</td>
</tr>
<tr>
<td></td>
<td>goog.dom.RangeIterator.prototype.next = function() {</td>
</tr>
<tr>
<td></td>
<td>// ...</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>Indicates that a term in a JSDoc descr generated documentation.</td>
</tr>
<tr>
<td>@const</td>
<td>@const</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>/**</td>
</tr>
<tr>
<td></td>
<td>* const */ var MY_BEER = 'stout';</td>
</tr>
<tr>
<td></td>
<td>/**</td>
</tr>
<tr>
<td></td>
<td>* My namespace's favorite kind of beer.</td>
</tr>
<tr>
<td></td>
<td>* @const</td>
</tr>
<tr>
<td></td>
<td>* @type {string}</td>
</tr>
<tr>
<td></td>
<td>*/</td>
</tr>
<tr>
<td></td>
<td>mynamespace.MY_BEER = 'stout';</td>
</tr>
<tr>
<td></td>
<td>/**</td>
</tr>
<tr>
<td></td>
<td>* const */ MyClass.MY_BEER = 'stout';</td>
</tr>
<tr>
<td></td>
<td>*/</td>
</tr>
<tr>
<td></td>
<td>Marks a variable as read-only and suitable for inlining. Constants should also be ALL_CAPS, the naming convention. Although @final is listed at jsdoc.org and is supported as a keyword. Note that changes to properties of const objects are not currently supported in the compiler (inconsistent with C++ const keyword). The type declaration can be omitted if it can be clearly inferred. If present, it must be on its own line. An empty line before the @const tag is optional.</td>
</tr>
<tr>
<td>@constructor</td>
<td>@constructor</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
</tr>
<tr>
<td></td>
<td>/**</td>
</tr>
<tr>
<td></td>
<td>* A rectangle.</td>
</tr>
<tr>
<td></td>
<td>*/</td>
</tr>
<tr>
<td></td>
<td>Used in a class's documentation to indicate the constructor.</td>
</tr>
<tr>
<td><strong>@define</strong> Type</td>
<td>description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Indicates a constant that can be overridden by the compiler at example, the compiler flag <code>--define=goog.userAgent_ASSUME_IE=true</code> specified in the BUILD file to indicate if it should be replaced with true.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>@deprecated</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to tell that a function, method or property should not be provided instructions on what callers should use instead.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>@enum</strong> Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>For example:</td>
</tr>
<tr>
<td>Given the code on the left, when the compiler is run with it will generate the code:</td>
</tr>
</tbody>
</table>

```javascript
goog.exportSymbol('foo.MyPublicClass.prototype.myPublicMethod', foo.MyPublicClass.prototype.myPublicMethod);
```

<table>
<thead>
<tr>
<th><strong>@extends</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>For example:</td>
</tr>
<tr>
<td>Used with @constructor to indicate that a class inherits from around the type are optional.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>@externs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Declares an externs file.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>@fileoverview</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Makes the comment block provide file information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>@implements</strong> Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>For example:</td>
</tr>
</tbody>
</table>

### Example Code
```javascript
function GM_Rect() {
  ...
}
```

```javascript
@define (Type) description
For example:

```javascript
/** @define (boolean) */
var TR_FLAGS_ENABLE_DEBUG = true;
/** @define (boolean) */
goog.userAgent.ASSUME_IE = false;
```
<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@implements</td>
<td>Used with @constructor to indicate that a class implements an interface. Around the type are optional.</td>
</tr>
<tr>
<td>@inheritdoc</td>
<td>Deprecated. Use @override instead. Indicates that a method or property of a subclass was inherited from the superclass, and has exactly the same documentation. Notice that no other documentation is included, the method or property documentation from its superclass.</td>
</tr>
<tr>
<td>@interface</td>
<td>Used to indicate that the function defines an interface. For example:</td>
</tr>
<tr>
<td>@lends</td>
<td>Indicates that the keys of an object literal should be object properties. The name in braces is not a type name like <code>@extends {Foo}</code> but <code>@lends {ObjectName}</code>. Notice that the name in braces is not a type name like <code>@extends {Shape}</code> but <code>@lends {ObjectName}</code>. It names the object on which the properties are &quot;lent&quot;. This annotation should only appear on object literals. The JSDoc Toolkit docs have more information on this.</td>
</tr>
<tr>
<td>@license</td>
<td>Anything marked by @license or @preserve will be retained by the top of the compiled code for that file. This annotation allows important notices (such as legal licenses or copyright text) to survive compilation.</td>
</tr>
<tr>
<td>@noalias</td>
<td>Used in an externs file to indicate that a name aliased as part of the alias externals must not be aliased as part of the alias externals. For example:</td>
</tr>
<tr>
<td>@nosideeffects</td>
<td>This annotation can be used as part of calls to the declared function have no side-effects. Remove calls to these functions if the return value is not used. For example:</td>
</tr>
<tr>
<td>@override</td>
<td>Indicates that a method or property of a subclass was implemented by the superclass. If no other documentation is included, the method or property documentation from its superclass. For example:</td>
</tr>
<tr>
<td>Annotation</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| `@param` | Declare a parameter for a method or constructor. | `@param {Type} varname Description`<br>For example:<br>`/**
 * Queries a Baz for items.
 * @param {number} groupNum Subgroup id to query.
 * @param {string|number|null} term An itemName, or searchTerm, or null to search everything.
 * @returns {string} The hex ID of the last item.
 * @example goog.Baz.prototype.query = function(groupNum, term) {
 *   // ...
 *   return id;
 * }` |
| `@private` | Declare a method or property that should not be used outside of the class. | `@private`<br>For example:<br>`/**
 * Handlers that are listening to this logger.
 * @type Array.<Function>
 * @private
 * this.handlers_ = [];` |
| `@protected` | Declare a method or property that is intended to be used by subclasses. | `@protected`<br>For example:<br>`/**
 * Sets the component’s root element to the given element. Considered protected and final.
 * @param {Element} element Root element for the component.
 * @protected
 * goog.ui.Component.prototype.setElementInternal = function(element) {
 *   // ...
 * }` |
| `@return` | Specify the type of the return value of a method or constructor. | `@return {Type} Description`<br>For example:<br>`/**
 * @return {string} The hex ID of the last item.
 * @example goog.Baz.prototype.getLastId = function() {
 *   return id;
 * }` |
| `@see` | Reference a lookup to another class function or method. | `@see Link`<br>For example:<br>`/**
 * Adds a single item, recklessly.
 * @see #addSafely
 * @see goog.collect
 * @see goog.RecklessAdder #add ...
 * @example` |
| `@supported` | Indicate the supported browsers for a feature. | `@supported Description`<br>For example:<br>`/**
 * @fileoverview Event Manager
 * @Provides an abstracted interface to the browsers’ event systems.
 * @supported So far tested in IE6 and FF1.5
 *` |
| `@suppress` | Suppress warnings from tools. | `@suppress (warning1|warning2)`<br>For example:<br>`/**
 * @suppress (deprecation)
 */
 * function f() {
 *   deprecatedVersionOff();
 * }` |
| `@template` | Declare a template parameter. | `@template`<br>For example:<br>`/**
 * @param {function(this:T, ...)} fn
 * @param {T} thisObj
 * @param {...*} var_args
 * @template T
 *` |
### Inner Classes and Enums

* @augments
* @argument
* @borrows
* @class
* @constant
* @constructs
* @default
* @event
* @example
* @field
* @function
* @ignore
* @inner
* @link
* @memberOf
* @name
* @namespace
* @property
* @public
* @requires
* @returns
* @since
* @static
* @version

**Inner Classes and Enums**

<table>
<thead>
<tr>
<th>@augments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Should be defined in the same file as the top level class.</td>
</tr>
</tbody>
</table>

### Compiling

<table>
<thead>
<tr>
<th>@augments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encouraged</td>
</tr>
</tbody>
</table>

Use of JS compilers such as the [Closure Compiler](https://closure-library.googlecode.com) is encouraged.

### Tips and Tricks

<table>
<thead>
<tr>
<th>@augments</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript tidbits</td>
</tr>
</tbody>
</table>

**True and False Boolean Expressions**

The following are all false in boolean expressions:

- `null`
- `undefined`
- `''` the empty string
- `0` the number

But be careful, because these are all true:
- `'0'` the string
- `[]` the empty array
- `{}` the empty object

This means that instead of this:

```javascript
while (x != null) {
}
```

you can write this shorter code (as long as you don’t expect x to be 0, or the empty string, or false):

```javascript
while (x) {
}
```

And if you want to check a string to see if it is null or empty, you could do this:

```javascript
if (y != null && y != '') {
}
```

But this is shorter and nicer:

```javascript
if (y) {
}
```

Caution: There are many unintuitive things about boolean expressions. Here are some of them:

- `Boolean('0') == true`
- `'0'` is true
- `0 != null`
- `0 == []`
- `0 == false`
- `Boolean(null) == false`
- `null != true`
- `null != false`
- `Boolean(undef) == false`
- `undefined != true`
- `undefined != false`
- `Boolean([]) == true`
- `[] != true`
- `[] == false`
- `Boolean({}) == true`
- `{}` != true
- `{}` != false

**Conditional (Ternary) Operator (?)**

Instead of this:

```javascript
if (val != 0) {
   return foo();
} else {
   return bar();
}
```

you can write this:

```javascript
return val ? foo() : bar();
```

The ternary conditional is also useful when generating HTML:

```javascript
var html = '<input type="checkbox" ' +
    (isChecked ? ' checked=' : '') +
    (isDisabled ? ' disabled=' : '') +
    ' name="foo">';
```

**&& and ||**

These binary boolean operators are short-circuited, and evaluate to the last evaluated term.

"&&" has been called the ‘default’ operator, because instead of writing this:

```javascript
/* @param {=} opt_win */
function foo(opt_win) {
  var win;
  if (opt_win) {
    win = opt_win;
  } else {
    win = window;
  }
  // ...
}
```

you can write this:

```javascript
/* @param {=} opt_win */
function foo(opt_win) {
  var win = opt_win || window;
  // ...
}
```

"||" is also useful for shortening code. For instance, instead of this:

```javascript
if (node) {
  if (node.kids) {
    if (node.kids[index]) {
      foo(node.kids[index]);
    }
  }
}
```
you could do this:

```javascript
if (node && node.kids && node.kids[index]) {
  foo(node.kids[index]);
}
```

or this:

```javascript
var kid = node && node.kids && node.kids[index];
if (kid) {
  foo(kid);
}
```

However, this is going a little too far:

```javascript
node && node.kids && node.kids[index] && foo(node.kids[index]);
```

Use join() to Build Strings

It is common to see this:

```javascript
function listHtml(items) {
  var html = '<div class="foo">';
  for (var i = 0; i < items.length; ++i) {
    if (i > 0) {
      html += ', ';
    }
    html += itemHtml(items[i]);
  }
  html += '</div>';
  return html;
}
```

but this is slow in Internet Explorer, so it is better to do this:

```javascript
function listHtml(items) {
  var html = [];
  for (var i = 0; i < items.length; ++i) {
    html[i] = itemHtml(items[i]);
  }
  return '<div class="foo">' + html.join(',') + '</div>';}
```

You can also use an array as a stringbuilder, and convert it into a string with `myArray.join('')`. Note that since assigning values to an array is faster than using `push()` you should use assignment where possible.

Iterating over Node Lists

Node lists are often implemented as node iterators with a filter. This means that getting a property like length is O(n), and iterating over the list by re-checking the length will be O(n^2).

```javascript
var paragraphs = document.getElementsByTagName('p');
for (var i = 0; i < paragraphs.length; i++) {
  doSomething(paragraphs[i]);
}
```

It is better to do this instead:

```javascript
var paragraphs = document.getElementsByTagName('p');
for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {
  doSomething(paragraph);
}
```

This works well for all collections and arrays as long as the array does not contain things that are treated as boolean false.

In cases where you are iterating over the childNodes you can also use the `firstChild` and `nextSibling` properties.

```javascript
var parentNode = document.getElementById('foo');
for (var child = parentNode.firstChild; child; child = child.nextSibling) {
  doSomething(child);
}
```

**Parting Words**

**BE CONSISTENT.**

If you’re editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around all their arithmetic operators, you should too. If their comments have little boxes of hash marks around them, make your comments have little boxes of hash marks around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you’re saying rather than on how you’re saying it. We present global style rules here so people know the vocabulary, but local style is also important. If code you add to a file looks drastically different from the existing code around it, it throws readers out of their rhythm when they go to read it. Avoid this.

Revision 2.28
Aaron Whyte
Bob Jervis
Dan Pupius
Erik Andresson
Fritz Schneider
Robby Walker