Upon a survey of already existing programming guidelines for JavaScript, we decided on following "Google Style Guide for JavaScript". Below is a copy of the guidelines along with its justifications. This is also available at: https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml

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 *do*s and *don't*s 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 constant *values*.
- Use @const to indicate a constant (non-overwritable) *pointer* (a variable or property).
- Never use the const keyword as it's not supported in Internet Explorer.

Decision:

Constant values

If a value is intended to be *constant* and *immutable*, it should be given a name in CONSTANT_VALUE_CASE.
ALL_CAPS additionally implies @const (that the value is not overwritable).
Primitive types (number, string, boolean) are constant values.
Objects' immutability is more subjective — objects should be considered immutable only if they do not demonstrate observable state change. This is not enforced by the compiler.

Constant pointers (variables and properties)

The @const annotation on a variable or property implies that it is not overwritable. This is enforced by the compiler at build time. This behavior is consistent with the const keyword (which we do not use due to the lack of support in Internet Explorer).
A @const annotation on a method additionally implies that the method cannot not be overridden in subclasses.
A @const annotation on a constructor implies the class cannot be subclassed (akin to final in Java).

Examples

Note that @const does not necessarily imply CONSTANT_VALUES_CASE. However, CONSTANT_VALUES_CASE *does* imply @const.

```
/**
 * Request timeout in milliseconds.
 * @type {number}
 */
goog.example.TIMEOUT_IN_MILLISECONDS = 60;
```

The number of seconds in a minute never changes. It is a constant value.ALL_CAPS also implies @const, so the constant cannot be overwritten.
The open source compiler will allow the symbol to be overwritten because the constant is *not* marked as @const.

```
/**
 * Map of URL to response string.
 * @const
 */
MyClass.fetchedUrlCache_ = new goog.structs.Map();
/**
 * Class that cannot be subclassed.
 * @const
 * @constructor
 */
sloth.MyFinalClass = function() {};
```

In this case, the pointer can never be overwritten, but value is highly mutable and not constant (and thus in camelCase, not ALL_CAPS).

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:

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

(function() {
  // 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.
[ffVersion, ieVersion][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, ieVersion][isIE]().
3. die is always called since the array minus 1 is NaN which is never equal to anything (not even if resultOfOperation() returns 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.

Clarification: Semicolons and functions

Semicolons should be included at the end of function expressions, but not at the end of function declarations. The distinction is best illustrated with an example:

```
var foo = function() {
  return true;
};  // semicolon here.

function foo() {
  return true;
}  // no semicolon here.
```

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:

```
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:

```
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
.

```
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 Libraryor a similar library function.

```
function D() {
  goog.base(this)
}
goog.inherits(D, B);

D.prototype.method = function() {
  ...
};
```

Method and property definitions

```
/** @constructor */
function SomeConstructor()
 {
     this.someProperty = 1;
 }
Foo.prototype.someMethod = function() { ... };
```

While there are several ways to attach methods and properties to an object created via "new", the preferred style for methods is:

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

The preferred style for other properties is to initialize the field in the constructor:

```
/** @constructor */
function Foo() {
  this.bar = value;
}
```

Why?

Current JavaScript engines optimize based on the "shape" of an object,adding a property to an object (including overriding a value set on the prototype) changes the shape and can degrade performance.

delete

Prefer this.foo = null.

```
Foo.prototype.dispose = function() {
  this.property_ = null;
};
```

Instead of:

```
Foo.prototype.dispose = function() {
  delete this.property_;
};
```

In modern JavaScript engines, changing the number of properties on an object is much slower than reassigning the values. The delete keyword should be avoided except when it is necessary to remove a property from an object's iterated list of keys, or to change the result of if (key in obj).

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:

```
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:

```
function foo(element, a, b) {
  element.onclick = bar(a, b);
}

function bar(a, b) {
  return function() { /* uses a and b */ };
}
```

eval()

Only for code loaders and REPL (Read–eval–print loop)

eval() makes for confusing semantics and is dangerous to use if the string being eval()'d contains user input. There's usually a better, clearer, and safer way to write your code, so its use is generally not permitted.

For RPC you can always use JSON and read the result usingJSON.parse() instead of eval().
Let's assume we have a server that returns something like this:

```
{
  "name": "Alice",
  "id": 31502,
  "email": "looking_glass@example.com"
}
```

```
var userInfo = eval(feed);
var email = userInfo['email'];
```

If the feed was modified to include malicious JavaScript code, then if we useeval then that code will be executed.

```
var userInfo = JSON.parse(feed);
var email = userInfo['email'];
```

With JSON.parse, invalid JSON (including all executable JavaScript) will cause an exception to be thrown.

with() {}

No

Using with clouds the semantics of your program. Because the object of thewith can have properties that collide with local variables, it can drastically change the meaning of your program. For example, what does this do?

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

Answer: anything. The local variable x could be clobbered by a property offoo 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

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 call()ed orapply()ed).
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

Only for iterating over keys in an object/map/hash
for-in loops are often incorrectly used to loop over the elements in anArray. This is however very error prone because it does not loop from 0 tolength - 1 but over all the present keys in the object and its prototype chain. Here are a few cases where it fails:

```
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.buhu = '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.

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

Associative Arrays

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 Objectinstead of Array in these cases because the features that you want are actually features of Object and not of Array. Array just happens to extendObject (like any other object in JS and therefore you might as well have used Date, RegExp or String).

Multiline string literals

No

Do not do this:

```
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.';
```

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:

```
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

Yes

Use Array and Object literals instead of Array and Object constructors.
Array constructors are error-prone due to their arguments.

```
// 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.

```
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.

```
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:

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

Modifying prototypes of builtin objects

No

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.

Internet Explorer's Conditional Comments

No

Don't do this:

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

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,CONSTANT_VALUES_LIKE_THIS, foo.namespaceNamesLikeThis.bar, andfilenameslikethis.js.

Properties and methods

- *Private* properties and methods should be named with a trailing underscore.
- *Protected* properties 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 @paramannotations. 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.

```
/**
 * 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 namedsetFoo(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.*.

```
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.

```
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 usingsloth.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 infoo.hats.*, your internal code should not define all its symbols infoo.hats.*, because it will break if the other team defines new symbols.

```
foo.require('foo.hats');

/**
 * WRONG -- Do NOT do this.
 * @constructor
 * @extends {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.

```
foo.provide('googleyhats.BowlerHat');

foo.require('foo.hats');

/**
 * @constructor
 * @extends {foo.hats.RoundHat}
 */
googleyhats.BowlerHat = function() {
  ...
};

goog.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.

```
/**
 * @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 create local aliases of namespaces. Namespaces should only be aliased using goog.scope.

```
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.

```
/** @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:
      ...
  }
};
myapp.main = function() {
  var MyClass = some.long.namespace.MyClass;
  MyClass.staticHelper(null);
};
```

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 customtoString() 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

We follow the C++ 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:

```
if (something) {
  // ...
} else {
  // ...
}
```

Array and Object Initializers

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

```
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, with the braces on their own line, just like blocks.

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

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

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

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

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

Not like this:

```
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:

```
// 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,
    tableModelEventHandlerProxy, 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,
    tableModelEventHandlerProxy,
    artichokeDescriptorAdapterIterator) {
  // ...
};

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

// Parenthesis-aligned, one argument per line.  Emphasizes each
// individual argument.
function bar(veryDescriptiveArgumentNumberOne,
             veryDescriptiveArgumentTwo,
             tableModelEventHandlerProxy,
             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.

```
if (veryLongFunctionNameA(
        veryLongArgumentName) ||
    veryLongFunctionNameB(
    veryLongArgumentName)) {
  veryLongFunctionNameC(veryLongFunctionNameD(
      veryLongFunctioNameE(
          veryLongFunctionNameF)));
}
```

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).

```
prefix.something.reallyLongFunctionName('whatever', function(a1, a2) {
  if (a1.equals(a2)) {
    someOtherLongFunctionName(a1);
  } else {
    andNowForSomethingCompletelyDifferent(a2.parrot);
  }
});

var names = prefix.something.myExcellentMapFunction(
    verboselyNamedCollectionOfItems,
    function(item) {
      return item.name;
    });
```

Aliasing with goog.scope

goog.scope may be used to shorten references to namespaced symbols in programs using the Closure Library.

Only one goog.scope invocation may be added per file. Always place it in the global scope.

The opening goog.scope(function() { invocation must be preceded by exactly one blank line and follow any goog.provide statements,goog.require statements, or top-level comments. The invocation must be closed on the last line in the file. Append // goog.scope to the closing statement of the scope. Separate the comment from the semicolon by two spaces.

Similar to C++ namespaces, do not indent under goog.scope declarations. Instead, continue from the 0 column.

Only alias names that will not be re-assigned to another object (e.g., most constructors, enums, and namespaces). Do not do this (see below for how to alias a constructor):

```
goog.scope(function() {
var Button = goog.ui.Button;

Button = function() { ... };
...
```

Names must be the same as the last property of the global that they are aliasing.

```
goog.provide('my.module.SomeType');

goog.require('goog.dom');
goog.require('goog.ui.Button');

goog.scope(function() {
var Button = goog.ui.Button;
var dom = goog.dom;

// Alias new types after the constructor declaration.
my.module.SomeType = function() { ... };
var SomeType = my.module.SomeType;

// Declare methods on the prototype as usual:
SomeType.prototype.findButton = function() {
  // Button as aliased above.
  this.button = new Button(dom.getElement('my-button'));
};
...
});  // goog.scope
```

Indenting wrapped lines

Except for array literals, object literals, and anonymous functions, all wrapped lines should be indented either left-aligned to a sibling expression above, or four spaces (not two spaces) deeper than a parent expression (where "sibling" and "parent" refer to parenthesis nesting level).

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

thisIsAVeryLongVariableName =
    hereIsAnEvenLongerOtherFunctionNameThatWillNotFitOnPrevLine();

thisIsAVeryLongVariableName = siblingOne + siblingTwo + siblingThree +
    siblingFour + siblingFive + siblingSix + siblingSeven +
    moreSiblingExpressions + allAtTheSameIndentationLevel;

thisIsAVeryLongVariableName = operandOne + operandTwo + operandThree +
    operandFour + operandFive * (
        aNestedChildExpression + shouldBeIndentedMore);

someValue = this.foo(
    shortArg,
    'Some really long string arg - this is a pretty common case, actually.',
    shorty2,
    this.bar());

if (searchableCollection(allYourStuff).contains(theStuffYouWant) &&
    !ambientNotification.isActive() && (client.isAmbientSupported() ||
                                        client.alwaysTryAmbientAnyways())) {
```

```
  ambientNotification.activate();
}
```

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. Otherwise, line breaks and indentation follow the same rules as in other Google style guides. This operator placement was initially agreed upon out of concerns about automatic semicolon insertion. In fact, semicolon insertion cannot happen before a binary operator, but new code should stick to this style for consistency.

```
var x = a ? b : c;  // All on one line if it will fit.

// Indentation +4 is OK.
var y = a ?
    longButSimpleOperandB : longButSimpleOperandC;

// Indenting to the line position of the first operand is also OK.
var z = a ?
        moreComplicatedB :
        moreComplicatedC;
```

This includes the dot operator.

```
var x = foo.bar().
    doSomething().
    doSomethingElse();
```

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.

`@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.

```
// File 1.
// AA_PrivateClass_ and AA_init_ are accessible because they are global
// and in the same file.

/**
 * @private
 * @constructor
 */
AA_PrivateClass_ = function() {
};

/** @private */
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 overridden by a subclass.

```
// File 1.

/** @constructor */
AA_PublicClass = function() {
  /** @private */
  this.privateProp_ = 2;

  /** @protected */
  this.protectedProp = 4;
};

/** @private */
AA_PublicClass.staticPrivateProp_ = 1;

/** @protected */
AA_PublicClass.staticProtectedProp = 31;

/** @private */
AA_PublicClass.prototype.privateMethod_ = function() {};

/** @protected */
AA_PublicClass.prototype.protectedMethod = function() {};

// 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 based on the EcmaScript 4 spec.

The JavaScript Type Language

The ES4 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 ES4 proposal has evolved, this language has changed. The compiler still supports old syntaxes for types, but those syntaxes are deprecated.

Syntax Name	Syntax	Description	Deprecated Syntaxes
Primitive Type	There are 5 primitive types in JavaScript: `{null}`,`{undefined}`, `{boolean}`,`{number}`, and `{string}`.	Simply the name of a type.	

Instance Type	`{Object}`
	An instance of Object or null.
	`{Function}`
	An instance of Function or null.
	`{EventTarget}`
	An instance of a constructor that implements the EventTarget interface, or null.
		An instance of a constructor or interface function.
		Constructor functions are functions defined with the`@constructor`JSDoc tag. Interface functions are functions defined with the `@interface`JSDoc tag. By default, instance types will accept null. This is the only type syntax that makes the type nullable. Other type syntaxes in this table will not accept null.	

Enum Type	{goog.events.EventType} One of the properties of the object literal initializer ofgoog.events.EventType.	An enum must be initialized as an object literal, or as an alias of another enum, annotated with the @enum JSDoc tag. The properties of this literal are the instances of the enum. The syntax of the enum is definedbelow. Note that this is one of the few things in our type system that were not in the ES4 spec.	
Type Applicatio n	{Array.<string>} An array of strings. {Object.<string, number>} An object in which the keys are strings and the values are numbers.	Parameterizes a type, by applying a set of type arguments to that type. The idea is analogous to generics in Java.	
Type Union	{(number|boolean)} A number or a boolean.	Indicates that a value might have type A OR type B. The parentheses may be omitted at the top-level expression, but the parentheses should be included in sub-expressions to avoid ambiguity. {number|boolean} {function(): (number|boolean)}	{(number, boolean)}, {(number|| boolean)}
Nullable type	{?number} A number or null.	Shorthand for the union of the null type with any other type. This is just syntactic sugar.	{number? }
Non-nulla ble type	{!Object} An Object, but never the nullvalue.	Filters null out of nullable types. Most often used with instance types, which are nullable by default.	{Object!}
Record Type	{{myNum: number, myObject}} An anonymous type with the given type members.	Indicates that the value has the specified members with the specified types. In this case,myNum with a typenumber andmyObject with any type. Notice that the braces are part of the type syntax. For example, to denote an Array of objects that have a lengthproperty, you might write Array.<{length}>.	
Function Type	{function(string, boolean)} A function that takes two arguments (a string and a boolean), and has an unknown return value.	Specifies a function.	
Function Return Type	{function(): number} A function that takes no arguments and returns a number.	Specifies a function return type.	
Functiont his Type	{function(this:goog.ui.Menu, string)} A function that takes one argument (a string), and executes in the context of a goog.ui.Menu.	Specifies the context type of a function type.	
Functionn ew Type	{function(new:goog.ui.Menu, string)} A constructor that takes one argument (a string), and creates a new instance of goog.ui.Menu when called with the 'new' keyword.	Specifies the constructed type of a constructor.	

Variable arguments	{function(string, ...[number]): number} A function that takes one argument (a string), and then a variable number of arguments that must be numbers.	Specifies variable arguments to a function.	
Variable arguments (in @paramannotations)	@param {...number} var_args A variable number of arguments to an annotated function.	Specifies that the annotated function accepts a variable number of arguments.	
Functionoptional arguments	{function(?string=, number=)} A function that takes one optional, nullable string and one optional number as arguments. The =syntax is only for function type declarations.	Specifies optional arguments to a function.	
Functionoptional arguments(in @paramannotations)	@param {number=} opt_argument An optional parameter of typenumber.	Specifies that the annotated function accepts an optional argument.	
The ALL type	{*}	Indicates that the variable can take on any type.	
The UNKNOWN type	{?}	Indicates that the variable can take on any type, and the compiler should not type-check any uses of it.	

Types in JavaScript

Type Example	Value Examples	Description
number	1 1.0 -5 1e5 Math.PI	
Number	new Number(true)	Number object
string	'Hello' "World" String(42)	String value
String	new String('Hello') new String(42)	String object
boolean	true false Boolean(0)	Boolean value
Boolean	new Boolean(true)	Boolean object
RegExp	new RegExp('hello') /world/g	
Date	new Date new Date()	
null	null	
undefined	undefined	

void	function f() {
	  return;
	}
		No return value

Array	['foo', 0.3, null] []	Untyped Array
Array.<number>	[11, 22, 33]	An Array of numbers
Array.<Array.<string>>	[['one', 'two', 'three'], ['foo', 'bar']]	Array of Arrays of strings
Object	{} {foo: 'abc', bar: 123, baz: null}	
Object.<string>	{'foo': 'bar'}	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 converted to strings, soobj['1'] == obj[1]. So the key will always be a string in for...in loops. But the compiler will verify the type of the key when indexing into the object.

Function	function(x, y) {
	  return x * y;
	}
		Function object

function(number, number): number	function(x, y) {
	  return x * y;
	}
		function value

SomeClass	/** @constructor */ function SomeClass() {} new SomeClass();	
SomeInterface	/** @interface */ function SomeInterface() {} SomeInterface.prototype.draw = function() {};	
project.MyClass	/** @constructor */ project.MyClass = function () {} new project.MyClass()	

project.MyEnum	/** @enum {string} */
	project.MyEnum = {
	  /** The color blue. */
	  BLUE: '#0000dd',
	  /** The color red. */
	  RED: '#dd0000'
	};
		Enumeration
		JSDoc comments on enum values are optional.

Element	document.createElement('div')	Elements in the DOM.
Node	document.body.firstChild	Nodes in the DOM.

HTMLInputElement	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.

```
/** @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.

Instances of classes and interfaces are nullable by default. For example, the following declaration

```
/**
 * 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. IfmyValue_ 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 MyClassis 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 thisargument 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 undefined 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 JSDoccomments with the appropriate tags and types. Textual descriptions for properties, methods, method parameters and method return values should be included unless obvious from the property, method, or parameter name.
Inline comments should be of the // variety.
Complete sentences are recommended but not required. Complete sentences should use appropriate capitalization and 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. You do not have to indent the @desc 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>, , <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 last timestamp
Instead, do this:

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

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

Top/File-Level Comments

A copyright notice and author information are optional. File overviews are generally recommended whenever a file consists of more than a single class definition. The top level comment is designed to orient readers unfamiliar with the code to what is in this file. If present, it should provide a description of the file's contents and any dependencies or compatibility information. As an example:

```
/**
 * @fileoverview Description of file, its uses and information
 * about its dependencies.
 */
```

Class Comments

Classes must be documented with a description and a type tag that identifies the constructor.

```
/**
 * 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

Parameter and return types should be documented. The method description may be omitted if it is obvious from the parameter or return type descriptions. Method descriptions should start with a sentence written in the third person declarative voice.

```
/**
 * 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) {
  // ...
}
```

Property Comments

```
/** @constructor */
project.MyClass = function() {
  /**
   * Maximum number of things per pane.
   * @type {number}
   */
  this.someProperty = 4;
}
```

JSDoc Tag Reference

Tag	Template & Examples	Description
@author	@author username@google.com (first last) *For example:*	Document the author of a file or the owner of a test, generally only used in

	/** * @fileoverview Utilities for handling textareas. * @author kuth@google.com (Uthur Pendragon) */	the@fileoverview comment.

@code	{@code ...}
	For example:
	/**
	* Moves to the next position in the selection.
	* Throws {@code goog.iter.StopIteration} when it
	* passes the end of the range.
	* @return {Node} The node at the next position.
	*/
	goog.dom.RangeIterator.prototype.next = function()
	{
	// ...
	};
		Indicates that a term in a JSDoc description is code so it may be correctly formatted in generated documentation.

@const	@const
	@const {type}
	For example:
	/** @const */ var MY_BEER = 'stout';
	
	/**
	* My namespace's favorite kind of beer.
	* @const {string}
	*/
	mynamespace.MY_BEER = 'stout';
	
	/** @const */ MyClass.MY_BEER = 'stout';
	
	/**
	* Initializes the request.
	* @const
	*/
	mynamespace.Request.prototype.initialize =
	function() {
	// This method cannot be overridden in a
	subclass.
	};
		Marks a variable (or property) as read-only and suitable for inlining.
		A @const variable is an immutable pointer to a value. If a variable or property marked as @const is overwritten, JSCompiler will give warnings.
		The type declaration of a constant value can be omitted if it can be clearly inferred. An additional comment about the variable is optional.
		When @const is applied to a method, it implies the method is not only not overwritable, but also that the method is *finalized* — not overridable in subclasses.
		For more on @const, see the Constants section.

@constructor	@constructor
	For example:
	/**
	* A rectangle.
	* @constructor
	*/
	function GM_Rect() {
	...
	}
		Used in a class's documentation to indicate the constructor.

@define	@define {Type} description *For example:* /** @define {boolean} */ var TR_FLAGS_ENABLE_DEBUG = true; /** * @define {boolean} Whether we know at compile-time that * the browser is IE.	Indicates a constant that can be overridden by the compiler at compile-time. In the example, the compiler flag --define='goog.userAgent.ASSUME_IE=true' could be specified in the BUILD file to indicate that the constantgoog.userAgent.ASSUME_IE should be replaced with true.

	*/ goog.userAgent.ASSUME_IE = false;	

@deprecated	@deprecated Description
	For example:
	/**
	* Determines whether a node is a field.
	* @return {boolean} True if the contents of
	* the element are editable, but the element
	* itself is not.
	* @deprecated Use isField().
	*/
	BN_EditUtil.isTopEditableField = function(node) {
	// ...
	};
		Used to tell that a function, method or property should not be used any more. Always provide instructions on what callers should use instead.

@dict	@dict Description
	For example:
	/**
	* @constructor
	* @dict
	*/
	function Foo(x) {
	this['x'] = x;
	}
	var obj = new Foo(123);
	var num = obj.x; // warning
	
	(/** @dict */ { x: 1 }).x = 123; // warning
		When a constructor (Foo in the example) is annotated with @dict, you can only use the bracket notation to access the properties of Foo objects. The annotation can also be used directly on object literals.

@enum	@enum {Type}
	For example:
	/**
	* Enum for tri-state values.
	* @enum {number}
	*/
	project.TriState = {
	TRUE: 1,
	FALSE: -1,
	MAYBE: 0
	};	

@export	@export
	For example:
	/** @export */
	foo.MyPublicClass.prototype.myPublicMethod =
	function() {
	// ...
	};
		Given the code on the left, when the compiler is run with the --generate_exportsflag, it will generate the code:
		goog.exportSymbol('foo.MyPublicClass.prototype.myPublicMethod',
		
		foo.MyPublicClass.prototype.myPublicMethod);
		which will export the symbols to uncompiled code. Code that uses the @exportannotation must either
		1. include //javascript/closure/base.js, or
		2. define both goog.exportSymbol and goog.exportProperty with the same method signature in their own codebase.

@expose	@expose *For example:* /** @expose */ MyClass.prototype.exposedProperty = 3;	Declares an exposed property. Exposed properties will not be removed, or renamed, or collapsed, or optimized in any way by the compiler. No properties with the same name will be able to be optimized either. @expose should never be used in library code, because it will prevent that property from ever getting removed.

@extends	@extends Type
	@extends {Type}
	For example:
	/**
	* Immutable empty node list.
	* @constructor
	* @extends goog.ds.BasicNodeList
	*/
	goog.ds.EmptyNodeList = function() {
	...
	};
		Used with @constructor to indicate that a class inherits from another class. Curly braces around the type are optional.

@externs	@externs *For example:* /** * @fileoverview This is an externs file. * @externs */ var document;	Declares an externs file.
@fileoverview	@fileoverview Description *For example:* /** * @fileoverview Utilities for doing things that require this very long * but not indented comment. * @author kuth@google.com (Uthur Pendragon) */	Makes the comment block provide file level information.

@implements	@implements Type
	@implements {Type}
	For example:
	/**
	* A shape.
	* @interface
	*/
	function Shape() {};
	Shape.prototype.draw = function() {};
	
	/**
	* @constructor
	* @implements {Shape}
	*/
	function Square() {};
	Square.prototype.draw = function() {
	...
	};
		Used with @constructor to indicate that a class implements an interface. Curly braces around the type are optional.

@inheritDoc	@inheritDoc *For example:*	**Deprecated. Use @override instead.** Indicates that a method or property of a

	/** @inheritDoc */
	project.SubClass.prototype.toString() {
	  // ...
	};
	
		subclass intentionally hides a method or property of the superclass, and has exactly the same documentation. Notice that@inheritDoc implies @override

@interface	@interface *For example:* /** * A shape. * @interface */ function Shape() {}; Shape.prototype.draw = function() {}; /** * A polygon. * @interface * @extends {Shape} */ function Polygon() {}; Polygon.prototype.getSides = function() {};	Used to indicate that the function defines an interface.

@lends	@lends objectName
	@lends {objectName}
	For example:
	goog.object.extend(
	  Button.prototype,
	  /** @lends {Button.prototype} */ {
	  isButton: function() { return true; }
	  });
		Indicates that the keys of an object literal should be treated as properties of some other object. This annotation should only appear on object literals.
		Notice that the name in braces is not a type name like in other annotations. It's an object name. It names the object on which the properties are "lent". For example,@type {Foo} means "an instance of Foo", but @lends {Foo} means "the constructor Foo".
		The JSDoc Toolkit docs have more information on this annotation.

@license or@preserve	@license Description *For example:* /** * @preserve Copyright 2009 SomeThirdParty. * Here is the full license text and copyright * notice for this file. Note that the notice can span several * lines and is only terminated by the closing star and slash: */	Anything marked by @license or @preserve will be retained by the compiler and output at the top of the compiled code for that file. This annotation allows important notices (such as legal licenses or copyright text) to survive compilation unchanged. Line breaks are preserved.
@noalias	@noalias *For example:* /** @noalias */ function Range() {}	Used in an externs file to indicate to the compiler that the variable or function should not be aliased as part of the alias externals pass of the compiler.
@nocompile	@nocompile *For example:* /** @nocompile */ // JavaScript code	Used at the top of a file to tell the compiler to parse this file but not compile it. Code that is not meant for compilation and should be omitted from compilation tests (such as bootstrap code) uses this annotation. Use sparingly.
@nosideeffects	@nosideeffects *For example:*	This annotation can be used as part of function and constructor declarations to

	/** @nosideeffects */
	function noSideEffectsFn1() {
	  // ...
	}
	
	/** @nosideeffects */
	var noSideEffectsFn2 = function() {
	  // ...
	};
	
	/** @nosideeffects */
	a.prototype.noSideEffectsFn3 = function() {
	  // ...
	};
		indicate that calls to the declared function have no side-effects. This annotation allows the compiler to remove calls to these functions if the return value is not used.

@override	@override
	For example:
	/**
	 * @return {string} Human-readable representation of project.SubClass.
	 * @override
	 */
	project.SubClass.prototype.toString = function() {
	  // ...
	};
		Indicates that a method or property of a subclass intentionally hides a method or property of the superclass. If no other documentation is included, the method or property also inherits documentation from its superclass.

@param	@param {Type} varname Description
	For example:
	/**
	 * Queries a Baz for items.
	 * @param {number} groupNum Subgroup id to query.
	 * @param {string|number|null} term An itemName,
	 * or itemId, or null to search everything.
	 */
	goog.Baz.prototype.query = function(groupNum, term) {
	  // ...
	};
		Used with method, function and constructor calls to document the arguments of a function.
		Type names must be enclosed in curly braces. If the type is omitted, the compiler will not type-check the parameter.

@private	@private @private {type} *For example:* /** * Handlers that are listening to this logger. * @private {!Array.<Function>} */ this.handlers_ = [];	Used in conjunction with a trailing underscore on the method or property name to indicate that the member is private and final.
@protected	@protected @protected {type} *For example:* /** * Sets the component's root element to the given element. * @param {Element} element Root element for the component. * @protected */ goog.ui.Component.prototype.setElementInternal =	Used to indicate that the member or property is protected. Should be used in conjunction with names with no trailing underscore.

	function(element) {
	  // ...
	};	

@public	@public @public {type} *For example:* /** * Whether to cancel the event in internal capture/bubble processing. * @public {boolean} * @suppress {visiblity} Referencing this outside this package is strongly * discouraged. */ goog.events.Event.prototype.propagationStopped_ = false;	Used to indicate that the member or property is public. Variables and properties are public by default, so this annotation is rarely necessary. Should only be used in legacy code that cannot be easily changed to override the visibility of members that were named as private variables.

@return	@return {Type} Description
	For example:
	/**
	 * @return {string} The hex ID of the last item.
	 */
	goog.Baz.prototype.getLastId = function() {
	  // ...
	  return id;
	};
		Used with method and function calls to document the return type. When writing descriptions for boolean parameters, prefer "Whether the component is visible" to "True if the component is visible, false otherwise". If there is no return value, do not use an @return tag.
		Type names must be enclosed in curly braces. If the type is omitted, the compiler will not type-check the return value.

@see	@see Link *For example:* /** * Adds a single item, recklessly. * @see #addSafely * @see goog.Collect * @see goog.RecklessAdder#add ...	Reference a lookup to another class function or method.

@struct	@struct Description
	For example:
	/**
	 * @constructor
	 * @struct
	 */
	function Foo(x) {
	  this.x = x;
	}
	var obj = new Foo(123);
	var num = obj['x']; // warning
	obj.y = "asdf"; // warning
	
	Foo.prototype = /** @struct */ {
	  method1: function() {}
	};
	Foo.prototype.method2 = function() {}; // warning
		When a constructor (Foo in the example) is annotated with @struct, you can only use the dot notation to access the properties of Foo objects. Also, you cannot add new properties to Foo objects after they have been created. The annotation can also be used directly on object literals.

@supported	@supported Description *For example:* /** * @fileoverview Event Manager	Used in a fileoverview to indicate what browsers are supported by the file.

	* Provides an abstracted interface to the * browsers' event systems. * @supported So far tested in IE6 and FF1.5 */	

@suppress	@suppress {warning1|warning2} @suppress {warning1,warning2}
	For example:
	/**
	* @suppress {deprecated}
	*/
	function f() {
	deprecatedVersionOfF();
	}
		Suppresses warnings from tools. Warning categories are separated by | or ,.

@template	@template
	For example:
	/**
	* @param {function(this:T, ...)} fn
	* @param {T} thisObj
	* @param {...*} var_args
	* @template T
	*/
	goog.bind = function(fn, thisObj, var_args) {
	...
	};
		This annotation can be used to declare a template typename.

@this	@this Type
	@this {Type}
	For example:
	pinto.chat.RosterWidget.extern('getRosterElement',
	/**
	* Returns the roster widget element.
	* @this pinto.chat.RosterWidget
	* @return {Element}
	*/
	function() {
	return this.getWrappedComponent_().getElement();
	});
		The type of the object in whose context a particular method is called. Required when the this keyword is referenced from a function that is not a prototype method.

@type	@type Type @type {Type} *For example:* /** * The message hex ID. * @type {string} */ var hexId = hexId;	Identifies the type of a variable, property, or expression. Curly braces are not required around most types, but some projects mandate them for all types, for consistency.
@typedef	@typedef *For example:* /** @typedef {(string|number)} */ goog.NumberLike; /** @param {goog.NumberLike} x A number or a string. */ goog.readNumber = function(x) { ...	This annotation can be used to declare an alias of a more complex type.

	}	

You may also see other types of JSDoc annotations in third-party code. These annotations appear in the JSDoc Toolkit Tag Reference but are currently discouraged in Google code. You should consider them "reserved" names for future use. These include:

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

Providing Dependencies With goog.provide

Only provide top-level symbols.

All members defined on a class should be in the same file. So, only top-level classes should be provided in a file that contains multiple members defined on the same class (e.g. enums, inner classes, etc).

Do this:
```
goog.provide('namespace.MyClass');
```

Not this:
```
goog.provide('namespace.MyClass');
goog.provide('namespace.MyClass.Enum');
goog.provide('namespace.MyClass.InnerClass');
goog.provide('namespace.MyClass.TypeDef');
goog.provide('namespace.MyClass.CONSTANT');
goog.provide('namespace.MyClass.staticMethod');
```

Members on namespaces may also be provided:
```
goog.provide('foo.bar');
goog.provide('foo.bar.method');
goog.provide('foo.bar.CONSTANT');
```

Compiling

Required
Use of JS compilers such as the Closure Compiler is required for all customer-facing code.

Tips and Tricks

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:

```
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):

```
while (x) {
```

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

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

But this is shorter and nicer:

```
if (y) {
```

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

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

Conditional (Ternary) Operator (?:)

Instead of this:

```
if (val) {
  return foo();
} else {
  return bar();
}
```

you can write this:

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

The ternary conditional is also useful when generating HTML:

```
var html = '<input type="checkbox"' +
    (isChecked ? ' checked' : '') +
    (isEnabled ? '' : ' 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:

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

you can write this:

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

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

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

you could do this:

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

or this:

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

However, this is going a little too far:

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

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).

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

It is better to do this instead:

```
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.

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