
	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01


UNIQUE-YANBAL

Guía de Prácticas y Estándares – Aplicaciones Móviles Web Híbrida

Estado : Borrador
Confidencialidad : Público


YANBAL  UNIQUE

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01


	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

Contenido

1.	Historial de versiones.....	5
2.	Autores.....	5
3.	Lista de distribución	5
4.	Área interesada/proyecto.....	5
5.	Resumen ejecutivo.....	5
6.	Introducción y objetivos	5
7.	Definiciones.....	6
8.	Documentos relacionados	6
9.	Excepciones.....	6
10.	Principios de diseño	7
11.	Estándares y prácticas.....	8
11.1.	Compatibilidad con tiendas.....	8
11.2.	Sistema de “Bug Report”	8
11.3.	Arquitectura aplicaciones tipo web híbrida	8
11.4.	Librería Corporativa	9
11.5.	Minimización	9
11.6.	Base de Datos Móvil.....	9
11.7.	Transaccionalidad.....	9
11.8.	Seguridad.....	10
12.	Pruebas.....	10
13.	Herramientas.....	11
14.	Fuentes	11
15.	Anexo A – Google HTML css Style Guide.....	12
16.	Anexo B – Estándares Programación y Documentación Javascript.	23
	Google JavaScript Style Guide	23
	Important Note	23
	Displaying Hidden Details in this Guide	23
	Background	23
	JavaScript Language Rules.....	23

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

var.....	23
Constants	24
Semicolons	25
Nested functions	26
Function Declarations Within Blocks.....	26
Exceptions	27
Custom exceptions	27
Standards features.....	27
Wrapper objects for primitive types	27
Multi-level prototype hierarchies	28
Method and property definitions	28
delete	29
Closures	29
eval().....	30
with() {}	30
this	31
for-in loop.....	31
Associative Arrays.....	32
Multiline string literals	32
Array and Object literals	32
Modifying prototypes of builtin objects	33
Internet Explorer's Conditional Comments	34
JavaScript Style Rules	34
Naming	34
Custom toString() methods.....	37
Deferred initialization.....	38
Explicit scope	38
Code formatting.....	38
Parentheses	43
Strings.....	43
Visibility (private and protected fields)	44
JavaScript Types.....	46
Comments.....	54
Providing Dependencies With goog.provide	68
Compiling	69
Tips and Tricks	69
Parting Words.....	72
17. Anexo C – Estándares Programación y Angular JS.	73

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

1. Historial de versiones

Responsable	Fecha	Versión/Comentarios
Jorge Cabrera	16/02/2015	Versión inicial - borrador

2. Autores

Nombre	Rol
Martín Valdivia	Revisor/Colaborador – seguridad
Arturo Iglesias	Revisor/Colaborador – CID
Jorge Cabrera	Responsable/Elaborador

3. Lista de distribución

Nombre	Cargo /Rol
Centro de Integración y Desarrollo	
Área de Arquitectura y Nueva Tecnología	
Área de Infraestructura Soporte Técnico y Seguridad	

4. Área interesada/proyecto

El presente es elaborado por el área de Arquitectura y Nuevas Tecnologías (PAN), teniendo como áreas interesadas la organización IT corporativa así como todas las áreas corporativas (y de las empresas del grupo) interesadas en desarrollar aplicaciones móviles o componentes de este tipo.

5. Resumen ejecutivo


El cumplimiento de los presentes estándares y prácticas está relacionado directamente a las siguientes características en las aplicaciones:

1. Tiempo de desarrollo
2. Flexibilidad
3. Portabilidad
4. Re-uso
5. Reducción de las incidencias
6. Rendimiento
7. Costo
8. Soporte

Para su cumplimiento es imprescindible el compromiso de los líderes y responsables de la implementación, debiendo brindar los recursos y mecanismos necesarios para esto.

6. Introducción y objetivos

El objetivo del presente documento es definir los estándares y las prácticas **a ser cumplidas (de forma obligatoria)** en el desarrollo de aplicaciones móviles de web híbrida así como componentes diseñados para ser ejecutados en este tipo de dispositivos.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

7. Definiciones

A continuación se enumeran las definiciones a ser tomadas en cuenta en el presente documento y anexos:

- ✓ Documento de Diseño Técnico: Documento que detalla el diseño técnico de un aplicativo o solución en particular.
- ✓ Documento de Análisis Técnico: Documento que detalla el análisis técnico de un aplicativo o solución en particular.
- ✓ Documento de Arquitectura: Documento que detalla la arquitectura de un aplicativo ó solución en particular.
- ✓ Practica: Ejercicio realizado bajo ciertas reglas; para los fines de este manual se considera obligatoria.
- ✓ Estándar: Modelo, regla o patrón
- ✓ CID : Centro de integración y desarrollo corporativo Yanbal
- ✓ PAN: Área de proyectos , arquitectura y nuevas tecnologías corporativa de Yanbal
- ✓ ISS : Área de infraestructura, soporte y seguridad de sistemas corporativo
- ✓ Aplicaciones Móviles: Aplicaciones diseñadas para ser ejecutadas en teléfonos inteligentes, tabletas y otros dispositivos móviles.
- ✓ Comité de Arquitectura: Órgano rector del área de arquitectura y de la arquitectura corporativa de Yanbal.

8. Documentos relacionados

El diseño de las aplicaciones debe de enmarcarse dentro de lo especificado en los siguientes documentos, lo indicado en el presente no implica bajo ningún criterio el no cumplimiento de lo allí definido.


- ✓ GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION
- ✓ POLITICAS DE SISTEMAS DE INFORMACION
- ✓ PRINCIPIOS DE ARQUITECTURA
- ✓ POLITICAS DE ARQUITECTURA

Dichos documentos pueden ser encontrados en el portal de intranet Yanbalnet ó Yn.

9. Excepciones

Sin perjuicio de lo expuesto y reconociendo que una práctica y/o estándar no es aplicable a todos los proyectos y situaciones, **cualquier excepción al presente deberá de ser presentada, justificada y documentada mediante alguno de los siguientes mecanismos de acuerdo a su condición:**

1. Excepción aprobada por el proyecto y director del área de seguridad (ISS): Cuando la excepción vaya en perjuicio de los principios y políticas definidas por el área de seguridad.
2. Excepción aprobada por el proyecto y director del área PAN: Cuando la excepción vaya en perjuicio de los principios y políticas definidas por el área.
3. Detallada y justificada en el documento de arquitectura (o equivalente): Cuando se trate de una desviación a los presentes estándares y donde este documento lo indique.
4. Detallada y justificada en el diseño técnico (o equivalente): En los puntos que especifica así el presente documento.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

10. Principios de diseño

Los siguientes principios deben ser cumplidos durante el diseño de aplicaciones y componentes de aplicativos móviles:

1. Las aplicaciones móviles serán de naturaleza Web Híbrida, salvo excepción al presente documentada en el documento de Arquitectura y aprobada por el comité de arquitectura.
2. KIS (keep it simple): El código debe de tratar de ser lo más sencillo posible, partiendo las tareas complejas en tareas sencillas y simples, evitando (en la medida de lo posible) código largo, sobre-complicado y difícil de entender.
3. DRY: De las siglas “Don’t Repeat Yourself”, este principio indica que :
 - ✓ Todo Comportamiento (o conocimiento “know how” ¹) debe tener una sola representación en código y debe de ser una sola pieza dentro de sistema.
 - ✓ Las funciones comunes o lógica similar **no deben de ser duplicadas** (Ej: Copy + Paste).

Por tanto, la lógica (comportamiento o “know how”) debe ser encapsulada e incluida en componentes comunes de la aplicación (si son de su ámbito de interés) ó en la librería corporativa (ver punto i “Librería Corporativa”) de tener probabilidades de re-uso entre aplicaciones.

4. YAGNY (“You Aren’t Gonna Need It”): No se deben incluir características o funcionalidades que no sean necesarias para cumplir lo especificado - **‘si no está en el concepto, no debe de ser incluido’**.
5. Open/Closed principle: Este principio postula que las clases deben estar abiertas para ser extendidas, pero cerradas para su modificación.
6. Segregación de Interfaces²: Se debe considerar que: “Muchas interfaces específicas son mejores que una sola de uso genérico”. Esto nos indica que ninguna clase cliente debería de depender de métodos que no utiliza, para esto es necesario dividir las tareas en interfaces más pequeñas, de esta manera no deberán “saber ó conocer” (dependen) de métodos que no son de su interés.
7. Inversión de Dependencia : Se deben observar estos dos postulados :
 - 7.1. Los módulos de alto nivel no deben depender de los módulos de bajo nivel, ambos deben depender de abstracciones.
 - 7.2. *Las abstracciones no deben depender de los detalles, los detalles deben depender de las abstracciones*.
8. Nunca presente información de depuración a sí mismo y/o al usuario final a través de la interfaz de usuario (por ejemplo: mensaje de alertas). Utilice el comando console, para el registro de la información de salida en la depuración.

¹ Kwon How : Conocimiento sobre cómo se realiza una tarea o se consigue algo

² Interface : En este caso se refiere al estereotipo Interface, no a las interfaces entre sistemas

El presente documento es propiedad de Corporación Yanbal International, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

11. Estándares y prácticas

Los estándares y prácticas a ser cumplidas e detallan a continuación:

11.1. Compatibilidad con tiendas

Las aplicaciones deberán cumplir los requisitos de publicación de las siguientes tiendas de aplicaciones móviles:

1. Apple iStore
2. Google Play
3. Windows Store

Para esto se deberá comprobar mediante su publicación (privada) en dichas tiendas, para comprobar el cumplimiento de ellas.

11.2. Sistema de “Bug Report”

Las aplicaciones móviles externas (dirigidas al público en general o fuerza de ventas) deberán de adoptar una herramienta de reporte de errores externa, el estándar actual para esta herramienta es:

1. Raygun (<https://raygun.io/>)

El acceso a esta librería deberá de estar encapsulado en la librería corporativa móvil y sus parámetros deben ser parámetros de la aplicación no editables por el usuario.


11.3. Arquitectura aplicaciones tipo web híbrida

Las aplicaciones móviles del tipo Web Híbrida se basaran en la siguiente arquitectura y considerarán una filosofía de SPA (single page application):

1. SQLite : Persistencia de datos en el dispositivo móvil
2. Apache Cordova /Phonegap: Plataforma de ejecución de las aplicaciones y acceso a los componentes de los dispositivos móviles.
3. Google AngularJS : Gestión de vistas , controladores y llamadas a la base de datos o servicios bajo un esquema MVC.
4. IONIC : Interface de usuario

El desarrollo en esta plataforma deberá de obedecer los siguientes lineamientos de acuerdo a la tecnología indicada:

1. HTML y CSS: Se deberá seguir las reglas planteadas en el Anexo A – Google HTML css Style Guide.
2. JavaScript: El código deberá de cumplir con lo especificado en el AnexoB – Estándares Programación y Documentación Javascript, en cuanto a código y documentación.
3. AngularJS: El uso de AngularJS, debe de enmarcarse dentro de los estándares de programación y documentación Javascript (Anexo B) y además lo especificado en el Anexo C – Estándares Programación y Angular JS

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

De adoptarse una herramienta de desarrollo se podrá utilizar el formato de código por default brindado por la misma, de lo contrario se adoptará el formato indicado en cada anexo de acuerdo a la tecnología.

11.4. Librería Corporativa

Las funciones que sean (o tengan altas probabilidades de ser) utilizadas en más de una aplicación deberán de añadirse a esta librería, siguiendo este mismo criterio los frameworks (y/o librerías) externas que se decidan incorporar ya sean libres o de fabricantes deberán estar encapsuladas dentro de esta librería.

Esta librería debe de ser incluida en los documentos de Arquitectura y Diseño indicando la versión de la misma además debe especificar que métodos y clases deberán de ser modificadas y/o añadidas a la misma (en diseño).

Criterio de Adopción: La decisión de adopción pertenecerá al área de arquitectura detallándose la incorporación en el documento de diseño y además en la documentación correspondiente a la librería.

En el caso de proveedores, este código deberá de ser incluido dentro de las aplicaciones móviles, siendo el proveedor responsable de las mejoras, correcciones o modificaciones (de ser necesarias) para su correcto funcionamiento.

11.5. Minimización

El código fuente javascript debe minimizarse, es decir remover los espacios en blanco antes de su compilación final, con la finalidad de reducir el tamaño de las aplicaciones.

Sin embargo el código fuente como entregable debe mantener los estándares fijados en el presente con respecto a espacios en blanco, indentación y orden.


11.6. Base de Datos Móvil

1. La base de datos a utilizarse en los aplicativos móviles es SQLite, siguiendo los estándares y prácticas definidas en el documento relativo a este punto (Guía de Práctica y Estándares de Base de Datos Relacional), el cual se podrá encontrar en el portal de intranet Yn ó Yanbalnet.

11.7. Transaccionalidad

Las aplicaciones móviles deberán asegurar (dentro su contexto: aplicación) no generar datos inconsistentes dentro de la base de datos del dispositivo móvil ni en los sistemas de **back end**, debiendo considerar para este fin:

1. Utilizar los mecanismos que ofrece la Base de Datos SQLite para mantener la integridad dentro de la base de datos del dispositivo móvil.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

2. En caso la aplicación permita su manejo fuera de línea, deberá de considerar y resolver las posibles inconsistencias y conflictos que se pudieran generar por su uso fuera de línea, debiendo realizar las consultas y validaciones necesarias al reconectarse (o en el momento propicio). A fin de no actualizar o insertar los sistemas del **back end** con información inconsistente ó expirada.
3. Deberá implementar los mecanismos necesarios para soportar errores y fallos en la transmisión de datos con los sistemas de **back end**, debiendo asegurar la consistencia de datos tanto en el dispositivo como en los sistemas antes mencionados ante dichas fallas.

11.8. Seguridad


Las aplicaciones móviles deberán:

1. Autenticar los usuarios contra los sistemas del **back-end** apropiado de acuerdo al caso (Intranet, Génesis, Directorio Activo).
2. Deberán exponer solamente los **contenidos y operaciones** a los cuales el usuario tiene autorización.
3. Se comunicarán con los sistemas de Unique mediante los protocolos seguros establecidos en la **GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION**.
4. Podrán almacenar las contraseñas de manera local siempre y cuando estas estén encriptados de acuerdo al punto 5.
5. Para encriptar datos utilizarán los algoritmos definidos en la **GUIA DE SEGURIDAD DE SISTEMAS DE INFORMACION**; sin embargo el número de iteraciones utilizadas así como el resto de parámetros serán un parámetro de la aplicación (no manipulable por el usuario) y será fijado de acuerdo a los requerimientos de esta, no debiendo ser menores a 2 iteraciones en ningún caso.
6. Deberán encriptar los siguientes tipos de datos (siempre y cuando la aplicación los gestione):
 - a. Datos personales
 - b. Información sensible sobre Yanbal y su negocio
 - c. Parámetros de Conexión a otros sistemas y dispositivos como por ejemplo servicios web.

12. Pruebas

De acuerdo al alcance definido para la aplicación, su arquitectura y funcionalidades se deberán considerar:

1. El área de arquitectura definirá el conjunto de equipos sobre los cuales se probará y certificará la aplicación, mismos sobre los cuales se realizarán la

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

totalidad de las pruebas de desarrollo y certificación considerando todos los roles e idiomas de ser el caso.

2. Se podrá definir un conjunto adicional de equipos para pruebas de **validación** con la finalidad de realizar pruebas adicionales sobre una o varias funcionalidades y escenarios acotados como por ejemplo (instalación e inicio), con la finalidad de comprobar parcialmente su funcionamiento.
3. De la misma manera se deberán de considerar las pruebas no funcionales listadas en el **Anexo D - CHECKLIST DE CASOS DE PRUEBA NO FUNCIONALES PARA APLICACIONES MOVILES**, especificando el conjunto de equipos donde deben ser ejecutadas.


13. Herramientas

Las siguientes herramientas pueden ser utilizadas para el desarrollo y mantenimiento de los aplicativos, debiendo acordarse las versiones de las mismas al inicio del proyecto:

- ✓ Android JDK
- ✓ Eclipse ADT (Android Developer Tools)
- ✓ Sublime Text Editor
- ✓ IntelliJ IDEA + PhoneGAP/Cordova Plugin
- ✓ Notepad ++
- ✓ xCode
- ✓ NodeJS
- ✓ IONIC
- ✓ PhoneGap
- ✓ NodeJS
- ✓ Apache ANT
- ✓ W3C HTML validator
- ✓ W3C CSS validator
- ✓ Firefox Developer Edition

14. Fuentes

1. Instalación y Configuración sobre Windows 7 u 8
(<http://learn.ionicframework.com/videos/windows-android/>)

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

15. Anexo A – Google HTML css Style Guide.

Traducido de: http://google-styleguide.googlecode.com/svn/trunk/htmlcssguide.xml?_sm_au_=iVVkWMRWjQbRjrTN el 15/01/2015

1. Generales

1. UTF – 8

Asegure de que el editor que está utilizando soporte este encoding sin macas, además especifiquelo en los archivos HTML la cabecera utilizando:

```
<meta charset="utf-8">
```

No lo especifique en los archivos de estilo, ya que se asumirá este encoding.

2. Protocolo:

Omitir el protocolo (http: https:) al momento de llamar a un recurso local de URLs que apuntan a las imágenes y otros archivos multimedia, hojas de estilo y scripts a menos que los respectivos archivos no están disponibles en ambos protocolos.

Incorrecto	Correcto
<code><script src="http://www.google.com/js/gweb/analytics/autotrack.js"></script></code>	<code><script src="//www.google.com/js/gweb/analytics/autotrack.js"></script></code>

3. Capitalización :

Todo el código debe estar en minúsculas, esto se aplica a los nombres de elementos HTML, atributos, valores de atributos, selectores CSS, las propiedades y los valores de propiedad (con la excepción de las cadenas).

Incorrecto	Correcto
<code></code>	<code></code>

4. Comentarios :

Explique el código cuando sea útil y necesario indicando su función y que realiza con respecto a la funcionalidad.

No es necesario ni recomendable documentar todo el código HTML y CSS y quedará al criterio del programador.

5. Comentarios de Acción (TODO)

Utilice solo la etiqueta TODO para este tipo de comentarios.

Incorrecto	Correcto
------------	----------

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

<code>{# TODO(john.doe): revisit centering #} <center>Test</center></code>	<code><!-- TODO: remove optional tags --> Apples Oranges </code>
--	--

6. Trailing Whitespace

No utilizar los caracteres de salto de carro ó trailing WhiteSpace, ya que son innecesarios y pueden complicar las divisiones.

Incorrecto	Correcto
<code><p>What?_</code>	<code><p>Yes please.</code>

2. HTML

7. Sintaxis HTML

Utilice código HTML con sintaxis valida, excepto en donde hacerlo no permita cumplir con la performance requerida dado el tamaño del archivo.

Incorrecto	Correcto
<code>All recommendations<title>Test</title> <article>This is only a test.</code>	<code>All recommendations<!DOCTYPE html> <meta charset="utf-8"> <title>Test</title> <article>This is only a test.</article></code>

- Utilice una herramienta como **W3C HTML validator**.

8. Semántica

Utilice los elementos para el fin que fueron creados (tags) , por ejemplo los elementos <headigs> para encabezados, <p> para párrafos y <a> para anclas, por ejemplo :

Incorrecto	Correcto
<code><div onclick="goToRecommendations();">All recommendations</div></code>	<code>All recommendations</code>

Esta característica es importante para asegurar la accesibilidad y re-uso de código.

9. Separación de Intereses (Separation of concerns)

Mantenga la estructura (markup) estrictamente separada de la presentación (styling) y mantenga su interacción al mínimo. Es decir los documentos y templates solo deben

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

contener HTML y servir a propósitos estructurales, todo lo correspondiente a presentación en hojas de estilo y todo lo relacionado a comportamiento en scripts.

Es más caro mantener documentos HTML así como templates cuando incluyen el estilo y los scripts.

Incorrecto	Correcto
<pre><!DOCTYPE html> <title>HTML sucks</title> <link rel="stylesheet" href="base.css" media="screen"> <link rel="stylesheet" href="grid.css" media="screen"> <link rel="stylesheet" href="print.css" media="print"> <h1 style="font-size: 1em;">HTML sucks</h1> <p>I've read about this on a few sites but now I'm sure: <u>HTML is stupid!!1</u> <center>I can't believe there's no way to control the styling of my website without doing everything all over again!</center></pre>	<pre><!DOCTYPE html> <title>My first CSS-only redesign</title> <link rel="stylesheet" href="default.css"> <h1>My first CSS-only redesign</h1> <p>I've read about this on a few sites but today I'm actually doing it: separating concerns and avoiding anything in the HTML of my website that is presentational. <p>It's awesome!</pre>

Excepción:

En caso utilizar angularjs, se puede omitir esta regla, la cual solo aplicaría a directivas tipo A (de atributo) sean propias de angular o creadas por el desarrollador, un ejemplo es aplicar el ng-click sobre un li de un ul, o utilizar un ng-show para mostrar u ocultar un elemento.

10. No utilices Dashes


No utilice “dashes” como **—**, **”**, o **☺** , asumiendo que todos los editores utilizados son compatibles con UTF-8 , salvo los caracteres especiales que forman parte de la sintaxis HTML como < y &, así como los caracteres de control como el de no salto de página.

Incorrecto	Correcto
The currency symbol for the Euro is “&eur;”.	The currency symbol for the Euro is “€”.

11. Omite las etiquetas opcionales

Con el fin de generar archivos más pequeños elimine todas las etiquetas opcionales indicadas en la especificación HTML:

<https://html.spec.whatwg.org/multipage/syntax.html#syntax-tag-omission>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

Incorrecto	Correcto
<pre><!DOCTYPE html> <html> <head> <title>Spending money, spending bytes</title> </head> <body> <p>Sic.</p> </body> </html></pre>	<pre><!DOCTYPE html> <title>Saving money, saving bytes</title> <p>Qed.</pre>

12. Omite el atributo type

Omite el atributo type para las hojas de estilo y scripts, este atributo no es necesario incluso en navegadores antiguos.

Incorrecto	Correcto
<pre><link rel="stylesheet" href="//www.google.com/css/maia.css" type="text/css"></pre>	<pre><link rel="stylesheet" href="//www.google.com/css/maia.css"></pre>
<pre><script src="//www.google.com/js/gweb/analytics/autotrack.js" type="text/javascript"></script></pre>	<pre><script src="//www.google.com/js/ gweb/analytics/autotrack.j s"></script></pre>

13. Espacios en blanco

No utilice espacios en blancos cuando comience dentro de un tag HTML: No es necesario incluir espacios en blanco, al momento de carga del documento, serán ignorados y limpiados por el navegador.

Incorrecto	Correcto
<pre><p> Hello, World </p></pre>	<pre><p>Hello, World</p></pre>

14. Especifique el doctype

Utilice siempre el tag document type: La mayoría de los navegadores reconocen siempre el primer tag como el uso correcto para las páginas HTML, siempre y cuando vaya a utilizar a cargar una página HTML y no un web component.

Incorrecto	Correcto
<pre><html lang="en"> <head> <meta charset="utf-8"> <meta http-equiv="X-UA-Compatible" content="IE=edge"></pre>	<pre><!doctype html> <html lang="en" ng-app="yanbal"> <head> <meta charset="utf-8"> <meta http-equiv="X-UA-Compatible" content="IE=edge"></pre>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

15. Bloques

Utilice una nueva línea por cada bloque HTML: Una nueva línea por cada bloque permite tener una lectura más rápida del documento y una organización más escalable.

Incorrecto	Correcto
<code><p>Hola a todos</p></code>	<code><p> Hola a todos </p></code>

16. Uso de Class e id

Utilice correctamente los atributos class e id: El atributo class se debe utilizar para mencionar que uno o varios elementos van a compartir un estilo o información en común.

El atributo id se debe utilizar para elementos que van a ser nombrados una sola vez, como un componente web.

Incorrecto	Correcto
<code><p id="idLista">Lista #1</p> <p id="idLista">Lista #2</p></code>	<code><p id="idLista"> Lista # 1 Lista # 2 </p></code>

Los id solo se deberían utilizar para aquellos elementos del DOM que vamos a interactuar a través de JS, para aplicación de estilos particulares se debería recurrir a utilizar selectores encadenados ejemplos:

```
p span {
  color: black;
}
.clase1 .clase2 div { lo que sea };
```

17. Uso de comillas dobles

Utilice comillas dobles (no comillas simples): Las comillas dobles deben ser usadas al momento de establecer un valor en un atributo de un tag HTML, ya que las comillas simples no son reconocidas por algunos navegadores a pesar que no generen error al momento de declararlos.

Incorrecto	Correcto
<code><div class="mensaje">Esto es un parrafo</div></code>	<code><div class="mensaje">Esto es un parrafo</div></code>

18. Declaración de Scripts

Declare sus archivos script siempre al final de la página: Al declarar archivos script al comienzo de la página incrementa el tiempo que toma la carga y visualización de la interfaz de usuario.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

Además se debería separar las áreas de los tipos de script que intervienen en el aplicativo, como componentes de terceros y componentes propios de la aplicación.

Incorrecto	Correcto
<pre> <!doctype html> <html lang="en"> <head> <script src="jquery.js"></script> </head> <body> </body> </html> </pre>	<pre> <!doctype html> <html lang="en"> <head> </head> <body> <script src="jquery.js"></script> </body> </html> </pre>

19. Anidación HTML

No anide demasiados tag HTML dentro de otro: Al tener demasiados tag dentro de otros genera que el navegador tenga una demora más al momento de leer todo el árbol HTML. El máximo de tag anidados, contando desde el tag inicial, es de 10. El tag inicial se considera al creador de un objeto dentro de la estructura HTML.

Incorrecto	Correcto
<pre> <p> <p> ... <p> <p> <p> Hello World </p> </p> </p> ... </p> </p> </pre>	<pre> <p>Hello World</p> </pre>

3. Formato HTML


En el caso de utilizar una herramienta, utilice el formato por default de la misma, caso contrario utilice las siguientes reglas:

- Utilice una nueva línea por cada bloque, lista, o elemento de una tabla e indentelo, como por ejemplo :

```

<blockquote>
  <p><em>Space</em>, the final frontier.</p>
</blockquote>
<ul>
  <li>Moe
  <li>Larry

```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```

    <li>Curly
  </ul>
  <table>
    <thead>
      <tr>
        <th scope="col">Income
        <th scope="col">Taxes
      </tr>
    <tbody>
      <tr>
        <td>$ 5.00
        <td>$ 4.50
      </tr>
    </tbody>
  </table>

```

4. Hoja de estilos (CSS)

20. Sintaxis CSS

Utilice código CSS válido siempre, salvo cuando esté utilizando componentes propietarios y/o corrigiendo errores y la sintaxis válida no sea reconocida correctamente.

21. Nomenclatura de id y Clases

Utilice nombres de selectores con identificación y significativos que reflejen el propósito de elemento, no utilice nombres de un estilo HTML crípticos que contengan números y caracteres que no tengan relación con la funcionalidad que van a realizar, prefiera además los nombres genéricos.

Incorrecto	Correcto
.buton-green {} .abc {} #indexABC {}	.button {} .index {} #video {}


Además utilice nombres cortos, pero lo suficientemente largos como para ser entendibles:

Incorrecto	Correcto
#navigation {} .atr {}	#nav {} .author {}

22. Shorthand Properties (Propiedades Abreviadas)

Utilice las definiciones cortas o abreviadas que brinda CSS (<http://www.w3.org/TR/CSS21/about.html#shorthand>) como por ejemplo font para declarar las propiedades ('font-style', 'font-variant', 'font-weight', 'font-size', 'line-height', y 'font-family') aun cuando solo se está declarando un solo valor.

Incorrecto	Correcto
-------------------	-----------------

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

border-top-style: none; font-family: palatino, georgia, serif; font-size: 100%; line-height: 1.6; padding-bottom: 2em; padding-left: 1em; padding-right: 1em; padding-top: 0;	border-top: 0; font: 100%/1.6 palatino, georgia, serif; padding: 0 1em 2em;
--	---

23. Nomenclatura de Selectores de tipo (Type Selectors)

Utilice selectores cortos: Los navegadores leen los “selectores” de izquierda a derecha y empiezan a recorrer el árbol HTML, cuanto más largo es el nombre del selector, más demora en realizar la acción.

Incorrecto	Correcto
header nav ul li a {} button strong span {} button strong span .callout{}	.primary-link {} Button span {} Button .callout {}

24. Class como identificador

Evite usar class con nombre de un identificador, con la excepción de reutilizar un ‘class’ en diferentes etiquetas HTML, no se deben combinar etiquetas y clases en una regla de estilo.

Incorrecto	Correcto
ul.example {} div.error {}	.example {} .error {} h1.title{} h2.title{}

25. 0 en Unidades


No utilice unidades si va a usar el cero: No se necesita especificar el valor de unidad si es que se va a usar el cero.

Incorrecto	Correcto
border-top: 0px; margin-bottom: 0em;	border: 0; margin: 0;

26. Omitir 0 para decimales

No utilice cero si se va a usar un valor decimal con cero como parte entera: El navegador reconoce automáticamente que se trata de un cero, basta con escribir el punto y el valor decimal.

Incorrecto	Correcto
font-size: 0.8em; line-height: 0.6em;	font-size: .8em; line-height: .6em;

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

27. Notación Hexadecimal

Utilice notación hexadecimal de 3 caracteres en lo posible: Algunos colores permiten poner hexadecimales de 3 caracteres y no es necesario ya poner los otros caracteres restantes.

Incorrecto	Correcto
color: #ffffff; background-color: #eebbcc;	color: #fff; background-color: #ebc;

28. Prefijos

Utilice prefijos en los selectores para módulos: Los prefijos permiten tener una mejor claridad para determinar que un grupo de selectores van a cumplir una funcionalidad en concreto, esto también permite tener un mejor control y que no exista duplicidad de código en los estilos.

Incorrecto	Correcto
/* estilos para un top bar */ nav .ul {} nav .ul li {} nav .search {} .search input {}	/* estilos para un top bar */ .topbar {} .topbar-search {} .topbar-input {}

5. Formato CSS

En caso utilice una herramienta puede utilizar las reglas de formato default que esta establezca, de no ser así siga las siguientes reglas:


29. Orden Alfabético

Organice las propiedades de los selectores alfabéticamente: La organización de las propiedades de forma alfabética permite llevar un orden en los selectores.

Incorrecto	Correcto
color: black; text-align: center; background: fuchsia; border-radius: 4px; text-indent: 2em; border: 1px solid; -moz-border-radius: 4px; -webkit-border-radius: 4px;	background: fuchsia; border: 1px solid; -moz-border-radius: 4px; -webkit-border-radius: 4px; border-radius: 4px; color: black; text-align: center; text-indent: 2em;

30. Sangría

Usar sangría por cada bloque: Utilizar sangría cuando se va a crear bloques dentro de otros.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

Incorrecto	Correcto
<pre>@media screen{ html { background: #fff; color: #444; } }</pre>	<pre>@media screen{ html { background: #fff; color: #444; } }</pre>

31. Comillas Simples

No utilice comillas dobles: Para la declaración del valor dentro de la propiedad, use siempre comillas simples como buena práctica. En caso de url, no use ninguna comilla.

Incorrecto	Correcto
<pre>@import url("//www.google.com/css/maia.css"); html { font-family: "open sans"; }</pre>	<pre>@import url(//www.google.com/css/maia.css); html { font-family: 'open sans'; }</pre>

32. Comentarios

Utilice comentarios para seccionar código: Seccionar el código para tener una mejor vista y coherencia. Crear un índice para saber qué selectores corresponden a quien.


Incorrecto	Correcto
<pre>.header {} .footer {} .content {} .alert {} .modal {}</pre>	<pre>/* Estilos para la funcionalidad x 1.- Header 2.- Footer 3.- Content 4.- Componentes */ /* 1.- Header */ .header {} /* 2.- Footer */ .footer {} /* 3.- Content */ .content {} /* 4.- Componentes */ .alert {} .modal {}</pre>

33. Estructura de archivos y directorios

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

Para la organización de archivos css se debe de agrupar por funcionalidad o características

Incorrecto	Correcto
index.css main.css fonts.css jquery.ui.css	Vendor ... bootstrap.css ... jquery.ui.css Componentes ... modal.css ... buttons.css ... alerts.css ... tables.css Modulos ... home.css ... contact.css ... clientes.css

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

16. Anexo B – Estándares Programación y Documentación Javascript.

Extraído de: <https://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>

Google JavaScript Style Guide

Revision 2.93

Aaron Whyte

Bob Jervis

Dan Pupius

Erik Arvidsson

Fritz Schneider

Robby Walker

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

▽ Toggle all summaries

Table of Contents

[JavaScript Language Rules](#) [var](#) [Constants](#) [Semicolons](#) [Nested functions](#) [Function Declarations Within Blocks](#) [Exceptions](#) [Custom exceptions](#) [Standards features](#) [Wrapper objects for primitive types](#) [Multi-level prototype hierarchies](#) [Method and property definitions](#) [delete](#) [Closures](#) [eval\(\)](#) [with\(\)](#) [{} this](#) [for-in loop](#) [Associative Arrays](#) [Multiline string literals](#) [Array and Object literals](#) [Modifying prototypes of builtin objects](#) [Internet Explorer's Conditional Comments](#)

[JavaScript Style Rules](#) [Naming](#) [Custom toString\(\) methods](#) [Deferred initialization](#) [Explicit scope](#) [Code formatting](#) [Parentheses](#) [Strings](#) [Visibility \(private and protected fields\)](#) [JavaScript Types](#) [Comments](#) [Providing Dependencies With goog.provide](#) [Compiling](#) [Tips and Tricks](#)

Important Note

Displaying Hidden Details in this Guide

[link](#) ▽

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

[link](#) ▽

Declarations with `var`: Always

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

- 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;
```

El presente documento es propiedad de *Corporación Yanbal International*, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

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]();
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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

[link](#) ▾

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

[link](#) ▾

No

Do not do this:

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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

[link](#) ▾

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

[link](#) ▾

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

[link](#) ▾

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

[link](#) ▾

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'.
}
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

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 a similar library function.

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

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

Method and property definitions

[link](#) ▾

```
/** @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;
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

}

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

[link](#) ▾

```
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

[link](#) ▾

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);
}
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

eval()

[link](#) ▾

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 using `JSON.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 use `eval` 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() {}

[link](#) ▾

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?

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

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

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 or `apply()`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

[link](#) ▾

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:

```
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.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

Associative Arrays

[link](#) ▾

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

[link](#) ▾

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

[link](#) ▾

Yes

Use Array and Object literals instead of Array and Object constructors.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

No

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

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

[link](#) ▾

In general, use `functionNamesLikeThis`, `variableNamesLikeThis`, `classNamesLikeThis`, `enumNamesLikeThis`, `methodNamesLikeThis`, `CONSTANT_VALUES_LIKE_THIS`, `foo.namespaceNamesLikeThis.bar`, and `filenameslikethis.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 `@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.

El presente documento es propiedad de *Corporación Yanbal International*, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
/**
 * 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.*`.

```
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 using `sloth.hats`.

Use different namespaces for external code and internal code

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

"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.

```
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) {
  ...
};
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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.

Filename

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

[link](#) ▾

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

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

[link](#) ▾

OK

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

Explicit scope

[link](#) ▾

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

[link](#) ▾

Expand for more information.

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.
```

El presente documento es propiedad de *Corporación Yanbal International*, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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(
```

El presente documento es propiedad de *Corporación Yanbal* International, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```

        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(
        veryLongFunctionNameE(
            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

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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');
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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(0xffff)) +
                    '';

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();
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
}
```

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

[link](#) ▾

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

[link](#) ▾

Prefer ' over "

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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)

[link](#) ▾

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.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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;
};
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
};
```

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

[link](#) ▾

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}	An enum must be	

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	One of the properties of the object literal initializer of <code>goog.events.EventType</code> .	initialized as an object literal, or as an alias of another enum, annotated with the <code>@enum JSDoc</code> tag. The properties of this literal are the instances of the enum. The syntax of the enum is defined below . Note that this is one of the few things in our type system that were not in the ES4 spec.	
Type Application	<code>{Array.<string>}</code> An array of strings. <code>{Object.<string, number>}</code> 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	<code>{ (number boolean) }</code> 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. <code>{number boolean}</code> <code>{function(): (number boolean)}</code>	<code>{ (number, boolean) },</code> <code>{ (number boolean) }</code>
Nullable type	<code>{?number}</code> A number or null.	Shorthand for the union of the null type with any other type. This is just syntactic sugar.	<code>{number?}</code>
Non-nullable type	<code>{!Object}</code> An Object, but never the null value.	Filters null out of nullable types. Most often used with instance types, which are nullable by default.	<code>{Object!}</code>
Record Type	<code>{{myNum: number, myObject}}</code> An anonymous type with the given type members.	Indicates that the value has the specified members with the specified types. In this case,	

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

		<p>myNum with a type number and myObject with any type.</p> <p>Notice that the braces are part of the type syntax. For example, to denote an Array of objects that have a length property, you might write <code>Array.<{length}></code>.</p>	
Function Type	<pre>{function(string, boolean) }</pre> <p>A function that takes two arguments (a string and a boolean), and has an unknown return value.</p>	Specifies a function.	
Function Return Type	<pre>{function(): number}</pre> <p>A function that takes no arguments and returns a number.</p>	Specifies a function return type.	
Function this Type	<pre>{function(this:goog.ui.Menu, string)}</pre> <p>A function that takes one argument (a string), and executes in the context of a goog.ui.Menu.</p>	Specifies the context type of a function type.	
Function new Type	<pre>{function(new:goog.ui.Menu, string)}</pre> <p>A constructor that takes one argument (a string), and creates a new instance of goog.ui.Menu when called with the 'new' keyword.</p>	Specifies the constructed type of a constructor.	
Variable arguments	<pre>{function(string, ...[number]): number}</pre> <p>A function that takes one argument (a string), and then a variable number of arguments that must be numbers.</p>	Specifies variable arguments to a function.	
Variable arguments (in @param annotations)	<pre>@param {...number} var_args</pre> <p>A variable number of arguments to an annotated function.</p>	Specifies that the annotated function accepts a variable number of arguments.	
Function optional arguments	<pre>{function(?string=, number=)}</pre> <p>A function that takes one optional,</p>	Specifies optional arguments to a function.	

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	nullable string and one optional number as arguments. The = syntax is only for function type declarations.		
Function optional arguments (in @param annotations)	@param {number=} opt_argument An optional parameter of type number.	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	

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

void	<pre>function f() { return; }</pre>	No return value
Array	<pre>['foo', 0.3, null] []</pre>	Untyped Array
Array.<number>	<pre>[11, 22, 33]</pre>	An Array of numbers
Array.<Array.<string>>	<pre>[['one', 'two', 'three'], ['foo', 'bar']]</pre>	Array of Arrays of strings
Object	<pre>{} {foo: 'abc', bar: 123, baz: null}</pre>	
Object.<string>	<pre>{'foo': 'bar'}</pre>	An Object in which the values are strings.
Object.<number, string>	<pre>var obj = {}; obj[1] = 'bar';</pre>	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, so <code>obj['1'] == obj[1]</code> . 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	<pre>function(x, y) { return x * y; }</pre>	Function object
function(number,	<pre>function(x, y) {</pre>	function

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

number): number	<pre>return x * y; }</pre>	value
SomeClass	<pre>/** @constructor */ function SomeClass() {} new SomeClass();</pre>	
SomeInterface	<pre>/** @interface */ function SomeInterface() {} SomeInterface.prototype.draw = function() {};</pre>	
project.MyClass	<pre>/** @constructor */ project.MyClass = function () {} new project.MyClass()</pre>	
project.MyEnum	<pre>/** @enum {string} */ project.MyEnum = { /** The color blue. */ BLUE: '#0000dd', /** The color red. */ RED: '#dd0000' };</pre>	Enumeration JSDoc comments on enum values are optional.
Element	<pre>document.createElement('div')</pre>	Elements in the DOM.
Node	<pre>document.body.firstChild</pre>	Nodes in the DOM.
HTMLInputElement	<pre>htmlDocument.getElementsByTagName('input')[0]</pre>	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

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
/**
 * 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.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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}
 */
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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 undefined this warning.
goog.bind(function() { this.someProperty; });
```

Comments

[link](#) ▾

Use JSDoc

We follow the [C++ style for comments](#) in spirit.

All files, classes, methods and properties should be documented with [JSDoc](#) comments 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.
 */
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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>`, ``, ``, ``, ``, `<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
 */
```

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

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	<p>@author username@google.com (first last)</p> <p><i>For example:</i></p> <pre>/** * @fileoverview Utilities for handling textareas. * @author kuth@google.com (Uthur Pendragon) */</pre>	Document the author of a file or the owner of a test, generally only used in the @fileoverview comment.
@code	<pre>{@code ...}</pre> <p><i>For example:</i></p> <pre>/** * Moves to the next position in the selection. * Throws {@code goog.iter.StopIteration} when it * passes the end of the</pre>	Indicates that a term in a JSDoc description is code so it may be correctly formatted in generated documentation.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<pre>range. * @return {Node} The node at the next position. */ goog.dom.RangeIterator.prototype.next = function() { // ... };</pre>	
@const	<pre>@const @const {type}</pre> <p><i>For example:</i></p> <pre>/** @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. };</pre>	<p>Marks a variable (or property) as read-only and suitable for inlining.</p> <p>A <code>@const</code> variable is an immutable pointer to a value. If a variable or property marked as <code>@const</code> is overwritten, JSCompiler will give warnings.</p> <p>The type declaration of a constant value can be omitted if it can be clearly inferred. An additional comment about the variable is optional.</p> <p>When <code>@const</code> is applied to a method, it implies the method is not only not overwritable, but also that the method is <i>finalized</i> — not overridable in subclasses.</p> <p>For more on <code>@const</code>, see the Constants section.</p>
@constructor	<pre>@constructor</pre> <p><i>For example:</i></p> <pre>/** * A rectangle. * @constructor */ function GM_Rect() { ... }</pre>	<p>Used in a class's documentation to indicate the constructor.</p>
@define	<pre>@define {Type} description</pre> <p><i>For example:</i></p>	<p>Indicates a constant that can be overridden by the compiler at compile-time. In the example, the compiler flag <code>--define='goog.userAgent.ASSUME_IE=tr</code></p>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01


	<pre>/** @define {boolean} */ var TR_FLAGS_ENABLE_DEBUG = true; /** * @define {boolean} * Whether we know at compile- * time that * the browser is IE. */ goog.userAgent.ASSUME_IE = false;</pre>	<p>ue ' could be specified in the BUILD file to indicate that the constant goog.userAgent.ASSUME_IE should be replaced with true.</p>
@depre- cated	<p>@deprecated Description</p> <p><i>For example:</i></p> <pre>/** * 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.isTopEditableFi eld = function(node) { // ... };</pre>	<p>Used to tell that a function, method or property should not be used any more. Always provide instructions on what callers should use instead.</p>
@dict	<p>@dict Description</p> <p><i>For example:</i></p> <pre>/** * @constructor * @dict */ function Foo(x) { this['x'] = x; } var obj = new Foo(123); var num = obj.x; // warning (** @dict */ { x: 1 }).x = 123; // warning</pre>	<p>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.</p>
@enum	<p>@enum {Type}</p> <p><i>For example:</i></p>	

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01


	<pre>/** * Enum for tri-state values. * @enum {number} */ project.TriState = { TRUE: 1, FALSE: -1, MAYBE: 0 };</pre>	
@export	<pre>@export For example: /** @export */ foo.MyPublicClass.prototype .myPublicMethod = function() { // ... };</pre>	<p>Given the code on the left, when the compiler is run with the --generate_exports flag, it will generate the code:</p> <pre>goog.exportSymbol('foo.MyPublicClass.prototype.myPublicMethod', foo.MyPublicClass.prototype.myPublicMethod);</pre> <p>which will export the symbols to uncompiled code. Code that uses the @export annotation must either</p> <ol style="list-style-type: none"> 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	<pre>@expose For example: /** @expose */ MyClass.prototype.exposedProperty = 3;</pre>	<p>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.</p> <p>@expose should never be used in library code, because it will prevent that property from ever getting removed.</p>
@extends	<pre>@extends Type @extends {Type} For example: /** * Immutable empty node</pre>	<p>Used with @constructor to indicate that a class inherits from another class. Curly braces around the type are optional.</p>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<pre>list. * @constructor * @extends goog.ds.BasicNodeList */ goog.ds.EmptyNodeList = function() { ... };</pre>	
@externs	<p>@externs</p> <p><i>For example:</i></p> <pre>/** * @fileoverview This is an externs file. * @externs */ var document;</pre>	Declares an externs file.
@fileoverview	<p>@fileoverview Description</p> <p><i>For example:</i></p> <pre>/** * @fileoverview Utilities for doing things that require this very long * but not indented comment. * @author kuth@google.com (Uthur Pendragon) */</pre>	Makes the comment block provide file level information.
@implements	<pre>@implements Type @implements {Type}</pre> <p><i>For example:</i></p> <pre>/** * A shape. * @interface */ function Shape() {}; Shape.prototype.draw = function() {}; /** * @constructor * @implements {Shape} */ function Square() {}; Square.prototype.draw =</pre>	Used with @constructor to indicate that a class implements an interface. Curly braces around the type are optional.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<pre>function() { ... };</pre>	
@inheritDoc	<p>@inheritDoc</p> <p><i>For example:</i></p> <pre>/** @inheritDoc */ project.SubClass.prototype. toString() { // ... };</pre>	<p>Deprecated. Use @override instead.</p> <p>Indicates that a method or property of a subclass intentionally hides a method or property of the superclass, and has exactly the same documentation. Notice that @inheritDoc implies @override</p>
@interface	<p>@interface</p> <p><i>For example:</i></p> <pre>/** * A shape. * @interface */ function Shape() {}; Shape.prototype.draw = function() {}; /** * A polygon. * @interface * @extends {Shape} */ function Polygon() {}; Polygon.prototype.getSides = function() {};</pre>	<p>Used to indicate that the function defines an interface.</p>
@lends	<p>@lends objectName</p> <p>@lends {objectName}</p> <p><i>For example:</i></p> <pre>goog.object.extend(Button.prototype, /** @lends {Button.prototype} */ { isButton: function() { return true; } });</pre>	<p>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".</p> <p>The JSDoc Toolkit docs have more information on this annotation.</p>
@license or @preserve	<p>@license Description</p> <p><i>For example:</i></p> <pre>/** * @preserve Copyright 2009</pre>	<p>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.</p>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	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: */	
@noalias	@noalias <i>For example:</i> <pre>/** @noalias */ function Range() {}</pre>	Used in an externs file to indicate to the compiler that the variable or function should not be aliased as part of the alias externs pass of the compiler.
@nocompile	@nocompile <i>For example:</i> <pre>/** @nocompile */ // JavaScript code</pre>	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 <i>For example:</i> <pre>/** @nosideeffects */ function noSideEffectsFn1() { // ... } /** @nosideeffects */ var noSideEffectsFn2 = function() { // ... }; /** @nosideeffects */ a.prototype.noSideEffectsFn 3 = function() { // ... };</pre>	This annotation can be used as part of function and constructor declarations to 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 <i>For example:</i> <pre>/** * @return {string} Human-</pre>	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.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01


	<pre>readable representation of project.SubClass. * @override */ project.SubClass.prototype. toString = function() { // ... };</pre>	
@param	<pre>@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) { // ... };</pre>	<p>Used with method, function and constructor calls to document the arguments of a function.</p> <p>Type names must be enclosed in curly braces. If the type is omitted, the compiler will not type-check the parameter.</p>
@private	<pre>@private @private {type} For example: /** * Handlers that are listening to this logger. * @private {!Array.<Function>} */ this.handlers_ = [];</pre>	<p>Used in conjunction with a trailing underscore on the method or property name to indicate that the member is private and final.</p>
@protected	<pre>@protected @protected {type} For example: /** * Sets the component's root element to the given element. * @param {Element} element Root element for the component. * @protected</pre>	<p>Used to indicate that the member or property is protected. Should be used in conjunction with names with no trailing underscore.</p>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<pre>*/ goog.ui.Component.prototype .setElementInternal = function(element) { // ... };</pre>	
@public	<pre>@public @public {type}</pre> <p><i>For example:</i></p> <pre>/** * Whether to cancel the * event in internal * capture/bubble processing. * @public {boolean} * @suppress {visibility} * Referencing this outside * this package is strongly * discouraged. */ goog.events.Event.prototype .propagationStopped_ = false;</pre>	<p>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.</p>
@return	<pre>@return {Type} Description</pre> <p><i>For example:</i></p> <pre>/** * @return {string} The hex * ID of the last item. */ goog.Baz.prototype.getLastI d = function() { // ... return id; };</pre>	<p>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.</p> <p>Type names must be enclosed in curly braces. If the type is omitted, the compiler will not type-check the return value.</p>
@see	<pre>@see Link</pre> <p><i>For example:</i></p> <pre>/** * Adds a single item, * recklessly. * @see #addSafely * @see goog.Collect * @see * goog.RecklessAdder#add * ...</pre>	<p>Reference a lookup to another class function or method.</p>
@struct	@struct Description	When a constructor (Foo in the example) is

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<p><i>For example:</i></p> <pre>/** * @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</pre>	<p>annotated with <code>@struct</code>, you can only use the dot notation to access the properties of <code>Foo</code> objects. Also, you cannot add new properties to <code>Foo</code> objects after they have been created. The annotation can also be used directly on object literals.</p>
@supported	<p>@supported Description</p> <p><i>For example:</i></p> <pre>/** * @fileoverview Event Manager * Provides an abstracted interface to the * browsers' event systems. * @supported So far tested in IE6 and FF1.5 */</pre>	<p>Used in a fileoverview to indicate what browsers are supported by the file.</p>
@suppress	<pre>@suppress {warning1 warning2} @suppress {warning1,warning2}</pre> <p><i>For example:</i></p> <pre>/** * @suppress {deprecated} */ function f() { deprecatedVersionOfF(); }</pre>	<p>Suppresses warnings from tools. Warning categories are separated by <code> </code> or <code>,</code>.</p>
@template	<p>@template</p> <p><i>For example:</i></p>	<p>This annotation can be used to declare a template typename.</p>

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<pre>/** * @param {function(this:T, ...)} fn * @param {T} thisObj * @param {...*} var_args * @template T */ goog.bind = function(fn, thisObj, var_args) { ... };</pre>	
@this	<pre>@this Type @this {Type}</pre> <p><i>For example:</i></p> <pre>pinto.chat.RosterWidget.ext ern('getRosterElement', /** * Returns the roster widget element. * @this pinto.chat.RosterWidget * @return {Element} */ function() { return this.getWrappedComponent_() .getElement(); });</pre>	The type of the object in whose context a particular method is called. Required when the <code>this</code> keyword is referenced from a function that is not a prototype method.
@type	<pre>@type Type @type {Type}</pre> <p><i>For example:</i></p> <pre>/** * The message hex ID. * @type {string} */ var hexId = hexId;</pre>	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	<pre>@typedef</pre> <p><i>For example:</i></p> <pre>/** @typedef { (string number)} */ goog.NumberLike; /** @param {goog.NumberLike} x A number or a string. */ goog.readNumber =</pre>	This annotation can be used to declare an alias of a more complex type .

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

	<pre>function(x) { ... }</pre>	
--	------------------------------------	--

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

[link](#) ▾

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');
```

El presente documento es propiedad de *Corporación Yanbal International*, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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

[link](#) ▾

Required

Use of JS compilers such as the [Closure Compiler](#) is required for all customer-facing code.

Tips and Tricks

[link](#) ▾

JavaScript tidbits

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:

El presente documento es propiedad de *Corporación Yanbal International*, prohibida su distribución ó copia parcial o total salvo expresa autorización del representante legal

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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:

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
/** @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 $O(n^2)$.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

```
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);
}
```

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 2.93

Aaron Whyte

Bob Jervis

Dan Pupius

Erik Arvidsson

Fritz Schneider

Robby Walker

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

17. Anexo C – Estándares Programación y Angular JS.

Extraído y modificado de:

<http://google-styleguide.googlecode.com/svn/trunk/angularjs-google-style.html>

1.1. Estructura de archivos y directorios


Para la organización de archivos HTML, estos se deben agrupar por funcionalidad o características.

Se evaluará incluir el directorio “partials” para fragmentos de código que incluirían tipo directivas de elementos o inclusiones dentro de un html.

Incorrecto	Correcto
Index.html Pages ... Layout.html ... About.html ... Contact.html ... Listado.html	Index.html Pages ... About.html ... Contact.html ... Clientes.html Web Componentes ... modal.html ... busqueda.html ... alert.html Templates ... mantenimiento.html ... forms.html ... principal.html

1.2. Generales:

1. Utilizar DTOs (data transfer objects) como moneda entre los datos, controladores y GUI.
2. Considerar implementar herencia a través de Class.js en los casos que aplique
3. La aplicación principal debe estar en el directorio raíz.
4. Un módulo nunca debe ser alterado, excepto en donde es definido.
5. Los módulos deben ser definidos en el mismo archivo que sus componentes o en un archivo separado para colocar ambas partes juntas.
6. Los módulos deben referenciar otros módulos usando la propiedad “name”

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

1.3. Reglas:

1. Closure

Manejar las dependencias con los cierres (Closure's) `goog.provide` y `goog.require`.
Escoja un espacio de nombres (namespace) para su proyecto y utilice los los cierres adecuados.

```
goog.provide('hello.about.AboutCtrl');
goog.provide('hello.versions.Versions');
```

Esto permitirá una mejor integración con el BUILD de google.

2. Módulos

El modulo principal debe estar en el directorio raíz, por tanto un módulo nunca debe de ser alterado en otro lugar más de donde es declarado.

Un módulo debe ser consistente para quien desee incluirlo como un componente re-utilizable, si un módulo puede tener distintos comportamientos (significar cosas distintas) de acuerdo a que archivos se incluyan entonces no es consistente.

3. Referencias a otros Módulos

Justificación: Usar la propiedad "name" del submódulo, previene fallas de Closure, también evita duplicar cadenas de texto.

Incorrecto	Correcto
<pre>//File submodule.js goog.provide("my.submodule"); my.submodule = angular.module("my.submodule",[]); //... //File app.js goog.require("my.submodule"); my.application.module = angular.module("hello", ["my.submodule"]);</pre>	<pre>//File submodule.js goog.provide("my.submodule"); My.submodule = angular.module("my.submodule",[]); //... //File app.js goog.require("my.submodule"); my.application.module = angular.module("hello", [my.submodule.name]);</pre>

4. Controladores y Scopes (contextos)

Los métodos deben ser definidos en el prototipo del controlador. Colocar los métodos y propiedades directamente en el controlador en vez de construir un objeto scope, resulta mejor con el estilo de clase de Google Closure. Además hace obvio qué controlador se está accediendo, en el caso de que múltiples controladores se apliquen a un elemento.

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

Incorrecto	Correcto
<pre>hello.mainpage.HomeCtrl = function(\$scope) { \$scope.homeCtrl = this; this.myColor = 'blue'; }; hello.mainpage.HomeCtrl.prototype.add = function(a, b) { return a + b; }; //template <div ng- controller="hello.mainpage.HomeCtrl"/> I'm in a color! {{homeCtrl.add(5, 6)}} </div></pre>	<pre>hello.mainpage.HomeCtrl = function() { this.myColor = 'blue'; }; hello.mainpage.HomeCtrl.prototype.add = function(a, b) { return a + b; }; //template <div ng- controller="hello.mainpage.HomeCtrl as homeCtrl"/> I'm in a color! {{homeCtrl.add(5, 6)}} </div></pre>

7. Sobre el uso de Scope:

El \$scope en angular se recomienda para compartir datos entre controller/ templates, Controller/directivas, directivas / directivas, el resto de la lógica debe ser propia del controlador y no depender del \$scope.

- El scope debe ser de sólo escritura en los controllers, es decir, el controller se encarga de llamar a otro componente, como un servicio, para obtener la data que se requiere mostrar y esta data es escrita en un objeto del scope.
 - El scope debe ser de sólo lectura en los templates, es decir, si bien AngularJS permite escribir código que modifique el scope en los templates, es algo con lo que tenemos que tener mucho cuidado y no se debería hacer.
 - No crear propiedades en el scope sino objetos, es un error común pensar que el scope es el modelo del que AngularJS, la verdad, el scope es sólo la forma en que el modelo es relacionado con el template, entonces, el modelo debe ser un objeto de javascript, usar una simple propiedad puede y va a traer problemas con la jerarquía de scopes.
- Toda manipulación del DOM debe ser hecha dentro de directivas a excepción de servicios para elementos del DOM desconectados del resto de la vista, como por ejemplo: ejemplo, ventanas de diálogo o atajos de teclado.
 - Usar module.service en vez de module.provider o module.factory; salvo las *excepción: Cuando se necesite inicializar más allá de la creación de una nueva instancia de clase, además se debe considerar :*

- El Services se debe instanciar

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

- El factory devuelve una instancia de objeto
- El provider crear la instancia de objeto cuando se invoca una funcion del mismo.

10. El carácter “\$” está reservado solo para propiedades y servicios de Angular

Justificación: No se debe usar el carácter “\$” para los propios objetos e identificadores de servicio. El estilo de nombrado está reservado para AngularJS y jQuery.

Incorrecto	Correcto
<pre>\$scope.\$myModel = { value: 'foo' } // INCORRECTO \$scope.myModel = { \$value: 'foo' } // INCORRECTO myModule.service('\$myService', function() { ... }); // INCORRECTO var MyCtrl = function(\$http) {this.\$http_ = \$http;}; // INCORRECTO</pre>	<pre>\$scope.myModel = { value: 'foo' } myModule.service('myService', function() { /*...*/ }); var MyCtrl = function(\$http) {this.http_ = \$http;};</pre>

11. Inyección de Dependencias :

Las dependencias se deben inyectar manteniendo la predefinición de las mismas es decir, no se recomienda inyectar dependencias de esta manera:

```
.controller('MiCtrl, function($scope, $http, $q ){
// Código;
});
```


Esto dado que en el momento de minificar el código este realiza un cambio del nombre de variables, lo que puede resultar en que por ejemplo la variable inyectada \$http se llame posteriormente \$h, al utilizar \$http en nuestra aplicación no se sabrá a que hace referencia, en lugar de esto al inyectarlas de la siguiente forma siempre se tendrá la definición de la variable.

```
.controller ('MiCtrl, [ '$scope', '$http', '$q', function($scope, $http, $q){
// Código
});
```

12. Uso del Closure :

El uso de closure permite encapsular las funcionalidades que quieren en un solo método, lo que asegura que dependa de su propio scope y que ningún otro JS pueda modificar nuestro JS, evita conflicto entre dependencias y plug ins de terceros por tener funciones o variables similares.

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

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

13. Controladores ligeros

Cuando se creen controladores se debe asegurar que no se maneje lógica pesada, es decir se debe manejar servicios o factory para realizar validaciones, consume de servicios y demás lógica que no tenga un contexto único del controlador que se referencia, esto disminuirá el código repetido y facilitar el mantenimiento de los mismos.

1.4. Practicas:

Se deberán de seguir las siguientes prácticas para el desarrollo en Angularjs

Extraído de (<https://github.com/angular/angular.js/wiki/Best-Practices>) el 04/04/2015

Best Practices

PatrickJS edited this page on 23 Jan · [22 revisions](#)

Pages 19

Navigation


- [FAQ](#)
 - [Understanding Directives](#)
 - [Understanding Scopes](#)
 - [Understanding Dependency Injection](#)
 - [When to use \\$scope.\\$apply\(\)](#)
 - [Best Practices](#)
 - [Anti-Patterns](#)
- [Design Discussions](#)
- [Writing AngularJS Documentation](#)
- [Upcoming Events](#)
- [Resources](#)
 - [JSFiddle Examples](#)
 - [Training Courses](#)
 - [Projects using AngularJS](#)

Clone this wiki locally

<https://github.com>

Related: [Anti-Patterns](#)

- **Namespace distributed code**
You shouldn't worry about prefixing internal code, but anything you plan to OpenSource should be namespaced
 - The `ng-` is reserved for core directives.
 - Purpose-namespacing (`i18n-` or `geo-`) is better than owner-namespacing (`djs-` or `igor-`)
 - Checkout [ui-alias](#) to remove 3rd party prefixes
- **Only use `.$broadcast()`, `.$emit()` and `.$on()` for atomic events**
Events that are relevant globally across the entire app (such as a user authenticating or the app closing). If you want events specific to modules, services or widgets you should consider Services, Directive Controllers, or 3rd Party Libs

	Guía de Prácticas y Estándares – Aplicaciones Móviles	Código ARQ-EST-DB-001	
		Fecha de emisión 07.01.2015	Versión 01

- `$scope.$watch()` should replace the need for events
- Injecting services and calling methods directly is also useful for direct communication
- Directives are able to directly communicate with each other through directive-controllers
- **Always let users use expressions whenever possible**
 - `ng-href` and `ng-src` are plaintext attributes that support `{{}}`
 - Use `$attrs.$observe()` since expressions are *async* and could change
- **Extend directives by using Directive Controllers**
You can place methods and properties into a directive-controller, and access that same controller from other directives. You can even override methods and properties through this relationship
- **Add teardown code to controllers and directives**
Controller and directives emit an event right before they are destroyed. This is where you are given the opportunity to tear down your plugins and listeners and pretty much perform garbage collection.
 - Subscribe to the `$scope.$on('$destroy', ...)` event
- **Leverage modules *properly***
Instead of slicing your app across horizontals that can't be broken up, group your code into related bundles. This way if you remove a module, your app still works.
 - Checkout [angular-app/angular-app](#) for a good example
 - `app.controllers`, `app.services`, etc will break your app if you remove a module
 - `app.users`, `app.users.edit`, `app.users.admin`, `app.projects`, etc allows you to group and nest related components together and create loose coupling
 - Spread route definitions across multiple module `.config()` methods
 - Modules can have their own dependencies (including external)
 - **Folder structure *should* reflect module structure**
- **Add NPM and Bower Support**
This has become the standard for AngularJS so it's a good idea to familiarize yourself