# 编码规范

## javascript

### 1.1 命名规范

#### 1.1.1 公有变量、方法名称采用驼峰命名

例如 sProductName , fGetUserPassword

#### 1.1.2 私有变量、方法名称加下划线前缀

例如 \_sName, \_fGetUser

#### 1.1.3 普通变量前缀

以变量类型的第一个字母为前缀，类名、属性及没有限制类型的变量无需前缀，某些作用域很小的变量，如for循环中的变量i，允许无前缀：

1. 加强代码的可阅读性；
2. 拓展命名空间，如获取sHtml，转化为aHtml，就不需要另取名称

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 标志 | 前缀 | 示例 |
| 布尔值 | {boolean} | b | bSample |
| 数值 | {number} | n | nSample |
| 字符串 | {string} | s | sSample |
| 数组 | {Array} | a | aSample |
| 对象 | {Object} | o | oSample |
| 元素 | {element} | e | eSample |
| 函数 | {function} | f | fSample |

### 注释规范

PS：注释遵守google clouse规范

<https://developers.google.com/closure/compiler/docs/js-for-compiler>

另见：附录1

#### 1.2.1 文件声明：

包括类说明、作者、类名

类说明  
@author 作者  
@class Example

如：

1. /\*\*
2. \* 示例类
3. \* @author 郑银辉
4. \* @class Example
5. \*/

#### 1.2.2 方法注释：

包括方法描述、方法名称、参数说明、返回值说明  
无参数或返回值不需要写@param、@return，  
方法名称中[]里表示可选参数，  
参数是object类型时，后面加{具体说明},类型是array时，后面加[具体说明]  
参数可以多种类型的情况用|分开，如:{string|boolean}

方法描述  
@method example(param)  
@param param{type} 说明  
@return {type} 说明

如:

1. /\*\*
2. \* 示例方法
3. \* @method fFunc(sName,[oData,]aArray)
4. \* @param sName{string} 名称
5. \* @param oData{object} 绑定的数据(可选){
6. \* nId{number}:id号,
7. \* sContent{string}:内容
8. \* }
9. \* @param aArray{array} 元素数组[
10. \* {
11. \* element{element|string}:要删除的元素或元素id,
12. \* sTag{string}:要删除的元素的图片标签名,
13. \* }
14. \* ]
15. \* @return {void}
16. \*/

## 2 css

### 2.1 提高页面渲染效率

#### 2.1.1 html代码中尽量不用样式相关的代码

#### 2.1.2 css类名的层级尽量浅 最多3级

#### 2.1.3 class和id前面不要添加标签名

#### 2.1.4 尽量不在css中直接使用html标签匹配

#### 2.1.5 颜色值使用大写字母

### 2.2 其它

#### 2.2.1 class命名时：s-表示html结构(structure)，m-表示模块(module)、w-表示组件(widget)，f-表示函数(function)，r-表示重用功能(reuse)

#### 2.2.2 class命名同时体现父子关系

如s-sd表示左栏结构，s-sd-bd表示左栏body结构，s-sd-bd-menu表示左栏body体内的菜单组件

#### 2.2.3 关于背景图片和颜色的css样式只能在皮肤的css中

## 3 后端接口数据

统一ajax返回数据

code:{String}操作返回的状态,成功为success, 其它都为异常

data:{Array} 操作返回的数据,单个数据也包装在array里面,统一格式.

(其中更新操作,必须包含主键值和有更新的属性. 删除操作,必须包含主键值)

其它数据(如分页相关属性)

例如:

{

code : "S\_OK",

data :{results: [

{

type: "",

val : "",

mid: "mid1"

},

{

type: "",

val : "",

mid: "mid2"

}

]},

其它数据(如分页相关属性)

}

## 附录1：

# Annotating JavaScript for the Closure Compiler

## Overview

The Closure Compiler can use data type information about JavaScript variables to provide enhanced optimization and warnings. JavaScript, however, has no way to declare types.

Because JavaScript has no syntax for declaring the type of a variable, you must use comments in the code to specify the data type.

The Closure Compiler's type language derives from the annotations used by the [JSDoc](http://code.google.com/p/jsdoc-toolkit/) document-generation tool. This document describes the set of annotations and type expressions that the Closure Compiler understands.

## Table of Contents

1. [JSDoc Tags](https://developers.google.com/closure/compiler/docs/js-for-compiler#tags)
2. [Type Expressions](https://developers.google.com/closure/compiler/docs/js-for-compiler#types)
3. [Generic Types](https://developers.google.com/closure/compiler/docs/js-for-compiler#generics)

## JSDoc Tags

The Closure Compiler looks for type information in JSDoc tags. Use the JSDoc tags described in the reference table below to help the compiler optimize your code and check it for possible type errors and other mistakes.

This table includes only tags that affect on the behavior of the Closure Compiler. For information about other JSDoc tags see the [JSDoc Toolkit documentation](http://code.google.com/p/jsdoc-toolkit/wiki/TagReference).

| **Tag** | **Syntax & Examples** | **Description** |
| --- | --- | --- |
| @const | @const  *For example:*  /\*\* @const \*/ var MY\_BEER = 'stout'; /\*\* \* My namespace's favorite kind of beer. \* @const \* @type {string} \*/ mynamespace.MY\_BEER = 'stout'; /\*\* @const \*/ MyClass.MY\_BEER = 'stout'; | Marks a variable as read-only. The compiler can inline @const variables, which optimizes the JavaScript code.  The type declaration and additional comment are optional. If you provide a type declaration, put the declaration on a separate line.  The compiler produces a warning if a variable marked with @const is assigned a value more than once. If the variable is an object, note that the compiler does not prohibit changes to the properties of the object. |
| @constructor | @constructor  *For example:*  /\*\* \* A rectangle. \* @constructor \*/ function GM\_Rect() { ... } | Marks a function as a constructor.  The compiler requires a @constructor annotation for any function that is used with the new keyword |
| @define | @define {Type} description  *For example:*  /\*\* @define {boolean} \*/ var ENABLE\_DEBUG = true; /\*\* @define {boolean} \*/ goog.userAgent.ASSUME\_IE = false; | Indicates a constant that can be overridden by the compiler at compile-time. With the example on the left, you can pass the flag --define='ENABLE\_DEBUG=false' to the compiler to change the value of ENABLE\_DEBUG to false. |
| @deprecated | @deprecated Description  *For example:*  /\*\* \* Determines whether a node is a field. \* @return {boolean} True if the contents of \* the element are editable, but the element \* itself is not. \* @deprecated Use isField(). \*/ BN\_EditUtil.isTopEditableField = function(node) { ... }; | Warns against using the marked function, method, or property should not be used. Using a deprecated method produces a compiler warning. |
| @dict | @dict  *For example:*  /\*\* \* @constructor \* @dict \*/ function Foo() {} var obj1 = new Foo(); obj1['x'] = 123; obj1.x = 234; // warning var obj2 = /\*\* @dict \*/ { 'x': 321 }; obj2.x = 123; // warning | @dict is used to create objects with a variable number of properties. When a constructor (Foo in the example) is annotated with @dict, you can only use the bracket notation to access the properties of Foo objects. The annotation can also be used directly on object literals. |
| @enum | @enum {Type}  *For example:*  /\*\* \* Enum for tri-state values. \* @enum {number} \*/ project.TriState = { TRUE: 1, FALSE: -1, MAYBE: 0 }; | Specifies the type of an enum. An enum is an object whose properties constitute a set of related constants. The @enum tag must be followed by a [type expression](https://developers.google.com/closure/compiler/docs/js-for-compiler#types).  The type label of an enum applies to each property of the enum. For example if an enum has type number, each of its enumerated properties must be a number.  If the type of an enum is omitted, number is assumed. |
| @expose | @expose  *For example:*  /\*\* @expose \*/ MyClass.prototype.exposedProperty = 3; | Declares an exposed property. Exposed properties will not be removed, or renamed, or collapsed, or optimized in any way by the compiler. No properties with the same name will be able to be optimized either.  @expose should never be used in library code, because it will prevent that property from ever getting removed. |
| @extends | @extends {Type}  *For example:*  /\*\* \* Immutable empty node list. \* @constructor \* @extends goog.ds.BasicNodeList \*/ goog.ds.EmptyNodeList = function() { ... }; | Marks a class or interface as inheriting from another class. A class marked with @extends must also be marked with either @constructor or @interface.  Note: @extends does not cause a class to inherit from another class. The annotation simply tells the compiler that it can treat one class as a subclass of another during type-checking.  For an example implementation of inheritance, see [the Closure Library function goog.inherits()](https://developers.google.com/closure/library/docs/introduction#oop). |
| @final | @final  *For example:*  /\*\* \* A class that cannot be extended. \* @final \* @constructor \*/ sloth.MyFinalClass = function() { ... } /\*\* \* A method that cannot be overridden. \* @final \*/ sloth.MyFinalClass.prototype.method = function() { ... }; | Indicates that this class is not allowed to be extended. For methods, indicates that no subclass is allowed to override that method. |
| @implements | @implements {Type}  *For example:*  /\*\* \* A shape. \* @interface \*/ function Shape() {}; Shape.prototype.draw = function() {}; /\*\* \* @constructor \* @implements {Shape} \*/ function Square() {}; Square.prototype.draw = function() { ... }; | Used with @constructor to indicate that a class implements an interface.  The compiler produces a warning if you tag a constructor with @implements and then fail to implement all of the methods and properties defined by the interface. |
| @inheritDoc | @inheritDoc  *For example:*  /\*\* @inheritDoc \*/ project.SubClass.prototype.toString = function() { ... }; | Indicates that a method or property of a subclass intentionally hides a method or property of the superclass, and has exactly the same documentation. Note that the @inheritDoc tag implies the [@override](https://developers.google.com/closure/compiler/docs/js-for-compiler#tag-override) tag. |
| @interface | @interface  *For example:*  /\*\* \* A shape. \* @interface \*/ function Shape() {}; Shape.prototype.draw = function() {}; /\*\* \* A polygon. \* @interface \* @extends {Shape} \*/ function Polygon() {}; Polygon.prototype.getSides = function() {}; | Marks a function as an interface. An interface specifies the required members of a type. Any class that implements an interface must implement all of the methods and properties defined on the interface's prototype. See [@implements](https://developers.google.com/closure/compiler/docs/js-for-compiler#tag-implements).  The compiler verifies that interfaces are not instantiated. If the new keyword is used with an interface function, the compiler produces a warning. |
| @lends | @lends {objectName}  *For example:*  goog.object.extend( Button.prototype, /\*\* @lends {Button.prototype} \*/ ({ isButton: function() { return true; } })); | Indicates that the keys of an object literal should be treated as properties of some other object. This annotation should only appear on object literals.  Notice that the name in braces is not a type name like in other annotations. It's an object name. It names the object to which the properties are lent. For example, @type {Foo} means "an instance of Foo", but @lends {Foo} means "the constructor Foo".  The [JSDoc Toolkit docs](http://code.google.com/p/jsdoc-toolkit/wiki/TagLends) have more information on this annotation. |
| @license or @preserve | @license Description  *For example:*  /\*\* \* @preserve Copyright 2009 SomeThirdParty. \* Here is the full license text and copyright \* notice for this file. Note that the notice can span several \* lines and is only terminated by the closing star and slash: \*/ | Tells the compiler to insert the associated comment before the compiled code for the marked file. This annotation allows important notices (such as legal licenses or copyright text) to survive compilation unchanged. Line breaks are preserved. |
| @nosideeffects | @nosideeffects  *For example:*  /\*\* @nosideeffects \*/ function noSideEffectsFn1() { ... }; /\*\* @nosideeffects \*/ var noSideEffectsFn2 = function() { ... }; /\*\* @nosideeffects \*/ a.prototype.noSideEffectsFn3 = function() { ... }; | Indicates that a call to the declared function has no side effects. This annotation allows the compiler to remove calls to the function if the return value is not used. The annotation is only allowed in [extern files](https://developers.google.com/closure/compiler/docs/api-tutorial3#externs). |
| @override | @override  *For example:*  /\*\* \* @return {string} Human-readable representation of \* project.SubClass. \* @override \*/ project.SubClass.prototype.toString = function() { ... }; | Indicates that a method or property of a subclass intentionally hides a method or property of the superclass. If no other annotations are included, the method or property automatically inherits annotations from its superclass. |
| @param | @param {Type} varname Description  *For example:*  /\*\* \* Queries a Baz for items. \* @param {number} groupNum Subgroup id to query. \* @param {string|number|null} term An itemName, \* or itemId, or null to search everything. \*/ goog.Baz.prototype.query = function(groupNum, term) { ... }; | Used with method, function and constructor definitions to specify the types of function arguments.  The @param tag must be followed by a [type expression](https://developers.google.com/closure/compiler/docs/js-for-compiler#types). |
| @private | @private  *For example:*  /\*\* \* Handlers that are listening to this logger. \* @type Array.<Function> \* @private \*/ this.handlers\_ = []; | Marks a member as private. Only code in the same file can access global variables and functions marked @private. Constructors marked @private can only be instantiated by code in the same file and by their static and instance members.  The public static properties of constructors marked @private may also be accessed anywhere, and the instanceof operator can always access @private members. |
| @protected | @protected  *For example:*  /\*\* \* Sets the component's root element to the given element. \* Considered protected and final. \* @param {Element} element Root element for the component. \* @protected \*/ goog.ui.Component.prototype.setElementInternal = function(element) { // ... }; | Indicates that a member or property is protected.  A property marked @protected is accessible to:   * all code in the same file * static methods and instance methods of any subclass of the class on which the property is defined. |
| @return | @return {Type} Description  *For example:*  /\*\* \* Returns the ID of the last item. \* @return {string} The hex ID. \*/ goog.Baz.prototype.getLastId = function() { ... return id; }; | Specifies the return types of method and function definitions. The @return tag must be followed by a [type expression](https://developers.google.com/closure/compiler/docs/js-for-compiler#types).  If there is no return value, do not use a @return tag. |
| @struct | @struct  *For example:*  /\*\* \* @constructor \* @struct \*/ function Foo(x) { this.x = x; } var obj1 = new Foo(123); obj1['x'] = "asdf"; // warning obj1.y = 5; // warning var obj2 = /\*\* @struct \*/ { x: 321 }; obj2['x'] = 123; // warning | @struct is used to create objects with a fixed number of properties. When a constructor (Foo in the example) is annotated with @struct, you can only use the dot notation to access the properties of Foo objects. The annotation can also be used directly on object literals. |
| @this | @this {Type}  *For example:*  chat.RosterWidget.extern('getRosterElement', /\*\* \* Returns the roster widget element. \* @this {Widget} \* @return {Element} \*/ function() { return this.getComponent().getElement(); }); | Specifies the type of the object to which the keyword this refers within a function. The @this tag must be followed by a [type expression](https://developers.google.com/closure/compiler/docs/js-for-compiler#types).  To prevent compiler warnings, you must use a @this annotation whenever this appears in a function that is neither a prototype method nor a function marked as a @constructor. |
| @template | @template T  *For example:*  /\*\* \* @param {T} t \* @constructor \* @template T \*/ Container = function(t) { ... }; | See [Generic Types](https://developers.google.com/closure/compiler/docs/js-for-compiler#generics). |
| @type | @type {Type}  *For example:*  /\*\* \* The message hex ID. \* @type {string} \*/ var hexId = hexId; | Identifies the type of a variable, property, or expression. The @type tag must be followed by a [type expression](https://developers.google.com/closure/compiler/docs/js-for-compiler#types). |
| @typedef | @typedef {Type}  *For example:*  /\*\* @typedef {(string|number)} \*/ goog.NumberLike; /\*\* @param {goog.NumberLike} x A number or a string. \*/ goog.readNumber = function(x) { ... } | Declares an alias for a more complex type. |

## Type Expressions

You can specify the data type of any variable, property, expression or function parameter with a type expression. A type expression consists of curly braces ("{ }") containing some combination of the type operators described below.

Use a type expression with the [@param](https://developers.google.com/closure/compiler/docs/js-for-compiler#tag-param) tag to declare the type of a function parameter. Use a type expression with the [@type](https://developers.google.com/closure/compiler/docs/js-for-compiler#tag-type) tag to declare the type of a variable, property, or expression.

The more types you specify in your code, the more optimizations the compiler can make and the more mistakes it can catch.

The compiler uses these annotations to type-check your program. Note that the Closure Compiler does not make any promises that it will be able to figure out the type of every expression in your program. It makes a best effort by looking at how variables are used, and at the type annotations attached to their declarations. Then, it uses a number of type inference algorithms to figure out the type of as many expressions as possible. Some of these algorithms are straightforward ("if x is a number, and we see y = x;, then y is a number"). Some are more indirect ("if f's first parameter is documented as a callback that must take a number, and we see f(function(x) { /\*\* ... \*/ });, then x must be a number").

| **Operator Name** | **Syntax Examples** | **Description** |
| --- | --- | --- |
| Type Name | {boolean} {Window} {goog.ui.Menu} | Specifies the name of a type. |
| Type Application | {Array.<string>} An array of strings.  {Object.<string, number>}  An object in which the keys are strings and the values are numbers. | Parameterizes a type with a set of type arguments. Similar to Java generics. |
| Type Union | {(number|boolean)} A number or a boolean.   Note the parentheses, which are required. | Indicates that a value might have type A OR type B. |
| Record Type | {{myNum: number, myObject}}  An anonymous type with both a property named myNum that has a value of type number and a property named myObject that has a value of any type. | Indicates that the value has the specified members with values of the specified types.  Braces are part of the type syntax. For example, to denote an Array of objects that have a length property, you might write:  Array.<{length}>. In the example on the left, the outer braces indicate that this is a type expression and the inner braces indicate that this is a record type. |
| Nullable type | {?number} A number or null. | Indicates that a value is type A or null.  All object types are nullable by default whether or not they are declared with the Nullable operator. An object type is defined as anything except a function, string, number, or boolean. To make an object type non-nullable, use the [Non-nullable](https://developers.google.com/closure/compiler/docs/js-for-compiler#nonnull) operator. |
| Non-nullable type | {!Object} An Object, but never the null value. | Indicates that a value is type A and not null.  Functions and all value types (boolean, number, and string) are non-nullable by default whether or not they are declared with the Non-nullable operator. To make a value or function type nullable, use the [Nullable](https://developers.google.com/closure/compiler/docs/js-for-compiler#null) operator. |
| Function Type | {function(string, boolean)} A function that takes two parameters (a string and a boolean), and has an unknown return value. | Specifies a function and the types of the function's parameters. |
| Function Return Type | {function(): number} A function that takes no parameters and returns a number. | Specifies the type of a function's return value. |
| Function this Type | {function(this:goog.ui.Menu, string)} A function that takes one parameter (a string), and executes in the context of a goog.ui.Menu. | Specifies the type of the value of this within the function. |
| Function new Type | {function(new:goog.ui.Menu, string)} A function that takes one parameter (a string), and creates a new instance of goog.ui.Menu when called with the 'new' keyword. | Specifies the constructed type of a constructor. |
| Variable parameters | {function(string, ...[number]): number} A function that takes one parameter (a string), and then a variable number of parameters that must be numbers. | Indicates that a function type takes a variable number of parameters, and specifies a type for the variable parameters. |
| Variable parameters (in @param annotations) | @param {...number} var\_args A variable number of parameters to an annotated function. | Indicates that the annotated function accepts a variable number of parameters, and specifies a type for the variable parameters. |
| Optional parameter in a @param annotation | @param {number=} opt\_argument An optional parameter of type number. | Indicates that the argument described by a @param annotation is optional. A function call can omit an optional argument. An optional parameter cannot precede a non-optional parameter in the parameter list.  If a method call omits an optional parameter, that argument will have a value of undefined. Therefore if the method stores the parameter's value in a class property, the type declaration of that property must include a possible value of undefined, as in the following example:  /\*\* \* Some class, initialized with an optional value. \* @param {Object=} opt\_value Some value (optional). \* @constructor \*/ function MyClass(opt\_value) { /\*\* \* Some value. \* @type {Object|undefined} \*/ this.myValue = opt\_value; } |
| Optional argument in a function type | {function(?string=, number=)} A function that takes one optional, nullable string and one optional number as arguments. | Indicates that an argument in a function type is optional. An optional argument can be omitted from the function call. An optional argument cannot precede a non-optional argument in the argument list. |
| 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. |

## Generic Types

Much like Java, the Closure Compiler supports generic types, functions, and methods. Generics operate on objects of various types while preserving compile-time type safety.

You can use generics to implement generalized collections that hold references to objects of a particular type, and generalized algorithms that operate over objects of a particular type.

### Declaring a Generic Type

A type can be made generic by adding a @template annotation to the type's constructor (for classes) or interface declaration (for interfaces). For example:

/\*\* \* @constructor \* @template T \*/ Foo = function() { ... };

The annotation @template T indicates that Foo is a generic type with one template type, T. The template type T can be used as a type within the scope of the definition of Foo. For example:

/\*\* @return {T} \*/ Foo.prototype.get = function() { ... }; /\*\* @param {T} t \*/ Foo.prototype.set = function(t) { ... };

The method get will return an object of type T, and the method set will only accept objects of type T.

### Instantiating a Generic Type

Reusing the example above, a templated instance of Foo can be created in several ways:

/\*\* @type {!Foo.<string>} \*/ var foo = new Foo(); var foo = /\*\* @type {!Foo.<string>} \*/ (new Foo());

Both of the above constructor statements create a Foo instance whose template type T is string. The compiler will enforce that calls to foo's methods, and accesses to foo's properties, respect the templated type. For example:

foo.set("hello"); // OK. foo.set(3); // Error - expected a string, found a number. var x = foo.get(); // x is a string.

Instances can also be implicitly typed by their constructor arguments. Consider a different generic type, Bar:

/\*\* \* @param {T} t \* @constructor \* @template T \*/ Bar = function(t) { ... }; var bar = new Bar("hello"); // bar is a Bar.<string>

The type of the argument to the Bar constructor is inferred as string, and as a result, the created instance bar is inferred as Bar.<string>.

### Multiple Template Types

A generic can have any number of template types. The following map class has two template types:

/\*\* \* @constructor \* @template Key, Val \*/ MyMap = function() { ... };

All template types for a generic type must be specified in the same @template annotation, as a comma-separated list. The order of the template type names is important, since templated type annotations will use the ordering to pair template types with the values. For example:

/\*\* @type {MyMap.<string, number>} \*/ var map; // Key = string, Val = number.

### Invariance of Generic Types

The Closure Compiler enforces invariant generic typing. This means that if a context expects a type Foo.<X>, you cannot pass a type Foo.<Y> when X and Y are different types, even if one is a subtype of the other. For example:

/\*\* \* @constructor \*/ X = function() { ... }; /\*\* \* @extends {X} \* @constructor \*/ Y = function() { ... }; /\*\* @type {Foo.<X>} \*/ var fooX; /\*\* @type {Foo.<Y>} \*/ var fooY; fooX = fooY; // Error fooY = fooX; // Error /\*\* @param {Foo.<Y>} fooY \*/ takesFooY = function(fooY) { ... }; takesFooY(fooY); // OK. takesFooY(fooX); // Error

### Inheritance of Generic Types

Generic types can be inherited, and their template types can either be fixed or propagated to the inheriting type. Here is an example of an inheriting type fixing the template type of its supertype:

/\*\* \* @constructor \* @template T \*/ A = function() { ... }; /\*\* @param {T} t \*/ A.prototype.method = function(t) { ... }; /\*\* \* @constructor \* @extends {A.<string>} \*/ B = function() { ... };

By extending A.<string>, B will have a method method that takes a parameter of type string.

Here is an example of an inheriting type propagating the template type of its supertype:

/\*\* \* @constructor \* @template U \* @extends {A.<U>} \*/ C = function() { ... };

By extending A.<U>, templated instances of C will have a method method that takes a parameter of the template type U.

Interfaces can be implemented and extended in a similar fashion, but a single type cannot implement the same interface multiple times with different template types. For example:

/\*\* \* @interface \* @template T \*/ Foo = function() {}; /\*\* @return {T} \*/ Foo.prototype.get = function() {}; /\*\* \* @constructor \* @implements {Foo.<string>} \* @implements {Foo.<number>} \*/ FooImpl = function() { ... }; // Error - implements the same interface twice

### Generic Functions and Methods

Similar to generic types, functions and methods can be made generic by adding a @template annotation to their definition. For example:

/\*\* \* @param {T} a \* @return {T} \* @template T \*/ identity = function(a) { return a; }; /\*\* @type {string} \*/ var msg = identity("hello") + identity("world"); // OK /\*\* @type {number} \*/ var sum = identity(2) + identity(2); // OK /\*\* @type {number} \*/ var sum = identity(2) + identity("2"); // Type mismatch

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](http://creativecommons.org/licenses/by/3.0/), and code samples are licensed under the [Apache 2.0 License](http://www.apache.org/licenses/LICENSE-2.0). For details, see our [Site Policies](https://developers.google.com/site-policies).

Last updated October 31, 2013.