Dojo – The Only JavaScript Library Compatible with The Closure Compiler

(other than the *Closure Library*, that is)

or

How to Use the Closure Compiler in Advanced Mode with the Dojo Toolkit

Author: Stephen Chung (Stephen.Chung@intexact.com)

Last Edited: 2011-03-06

Table of Contents

Preamble

This document is the result of a series of experiments by the author to use the Closure Compiler in *Advanced Optimizations* mode with the Dojo Toolkit, version 1.6.

The Dojo Toolkit is not written in the (extremely restrictive) style that takes full benefits of the Closure Compiler. However, the gap can be bridged quite successfully via a number of tricks and hacks, mostly in modifying the standard Build script.

The author has deployed a medium-scale mobile web application targeted for iPad's and Android tablets. Through this experience, the author believes that the Closure Compiler (in Advanced mode) provides performance and other benefits that should not be ignored, especially by mobile web applications.

On the other hand, however, the massive infrastructure provided by the Dojo Toolkit is also too good to ignore. In a perfect world, there should be a way to seamlessly marry the two. This writing is an on-going documentation of this attempt.

Design Concept

There are a few central concepts when designing the following process of using the Dojo Toolkit with the Closure Compiler:

- The process must make as few changes to the Dojo Toolkit as possible – this is an attempt to use the Dojo Toolkit primarily, *with* the Closure Compiler secondarily, not the other way round
- The programmer must be able to use all features in the Dojo Toolkit without much restrictions, and in pretty much the same way as normal – although he/she must take special concern regarding the restrictions of the Closure Compiler in his/her own code
- Any program written must run without change in the raw *without* being compiled by the Closure Compiler
- The programmer must be able to produce a *normal* Dojo Build (without using the Closure Compiler) and such build must also run without change

Caveat

The author's application only uses a small subset of the Dojo Toolkit's capabilities – e.g. most of Dijit isn't used due to it being a mobile application. It uses **dojox.mobile** quite extensively, though. The application is also built with the **webkitMobile** flag turned on, essentially eliminating large sections of code related to other browsers.

The author does not claim to have found every last place in the Dojo Toolkit source tree that must be modified, nor does he claim that the procedure outlined in this document suffices in getting every web program to work with the Closure Compiler.

The reader is warned that a fair bit of experimentation, debugging and tweaking *will* be needed for any new web project that uses both the Dojo Toolkit and the Closure Compiler in Advanced mode. Nevertheless, this document should provide strong guidance towards solving some of the major recurring issues.

In addition, several tricks and hacks to make this work depends on the fact that the Closure Compiler always converts the same name to the same mangled name, even though they may be properties of different classes. In other words, experimental compilation flags such as **disambiguateProperties** and **ambiguateProperties** are not supported.

Contributions back to this document are strongly encouraged and should be sent to the author's email address.

Who is This For

This document is for programmers who have a capable understanding of the Closure Compiler in *Advanced Optimizations* mode, who have successfully used the compiler in *Advanced* mode for other, non-Dojo-based projects, and who would now like to do the same for Dojo-based projects.

The reader is also assumed to have a working understanding of how to invoke the Dojo Build process to make merged builds and multiple layer files.

Why Closure Compiler

Why is the Closure Compiler beneficial to projects based on the Dojo Toolkit? In general, the Closure Compiler in *Simple Optimizations* mode performs no better than current general top-of-the-line JavaScript compressors (e.g. Uglify), and there really is no compelling reason to use it with the Dojo Toolkit. The standard Dojo Build system has an option to use the Closure Compiler in Simple mode for compression.

However, in *Advanced Optimizations* mode, the Closure Compiler offers a whole range of additional benefits:

- Syntax checking eliminates many typo and careless bugs – a task also fulfilled by using JavaScript linters.
- Type checking eliminates many obscure argument bugs – although it can also be a pain to use sometimes; purists will say that it takes away some of the flexibility of JavaScript…

- Dead code removal – smaller downloads, higher performance – although this is actually less of a potential benefit for heavily-modularized libraries like Dojo, and Dojo Core functions contain substantial amounts of cross-calling to prevent much dead code removal, and user code should not have much dead code anyway…
- Renaming of *all* properties and variables – smaller downloads, superior obfuscation – pretty-print a compressed JavaScript file and you can figure out its logic based on the un-mangled public property/function names; pretty-print a Closure-compiled JavaScript file and it is really difficult to figure out anything (for example, even the top namespace objects "**dojo**", "**dijit**" and "**dojox**" can be removed).
- Functions and constants in-lining – higher performance, superior obfuscation. Another major benefit is that this enables writing highly configurable enterprise software systems, for example multiple layers of factory abstractions and dependency injection driven by a configuration/setup file and the ability to produce "optimized" builds for each particular config with the Compiler automatically unrolling all the abstraction layers via in-lining.
- Flattening of namespaces – higher performance, especially on mobile devices.
- Virtualization of prototype methods – higher performance, especially on mobile devices.

The Catch

As with any good thing, there are costs. The programmer must be **ultra careful** in order to run *Advanced Mode* compilations. It is not the purpose of this document to outline the list of extremely strict requirements. However, most difficulties can be resolved via:

- A special build process – a new command-line flag is added to the Dojo Build script called "**closure**" which, when set to true, will make a range of necessary optimizations and output build files in a special format that is capable of being processed by the Closure Compiler. This is the "easy" solution as it is reasonably automatic.
- Specially marked comments that indicate to the Dojo Build script that special care should be taken. This is especially necessary when using "**dojo.declare**" to declare new classes – a "property names map" must be created by the special Build script in order to convert un-mangled property names (used in **get**/**set** calls) to mangled names.
- An "extern" file (used by the Closure Compiler) that lists out the property names that should *not* be renamed. This list should be as short as possible for obvious reasons, but it is not always possible to completely eliminate it because the Dojo Toolkit is sprinkled with property accesses via string names – a huge no-no for the Closure Compiler.

The Special Build Process

The special Build process is responsible for converting normal Dojo-style source code files into a format acceptable to the Closure Compiler.

Step 1: Invoking the special Build process

Do either of the following:

- In your Build profile, specify the following parameter under **dependencies**:
  closure: true,

- Use the following command-line parameter when running the build script:
  closure=true

A sample command line executed on Windows:

```
build profileFile=profile.js action=release closure=true
```

Step 2: Constructing the Build profile

All **dojo**, **dijit**, **dojox** and user code *must* be separated into different layers. In particular, the **dojo** and **dijit** layers *must* be separated due to i18n bundle resources.

Separating the code base into different layers make it easier to use different Closure settings for each layer – for example, user code should have full type-checking turned on, but not for Dojo Toolkit layers (otherwise there will be a **large** number of type errors).

For each Dojo Toolkit layer, include a *copyright* file, which should contain the following:

```
/**
 * @fileoverview
 * @suppress {checkTypes}
 */
```

Doing the above will prevent the Closure Compiler from doing type-checking of these layer files. Otherwise, if type-checking is turned on, there will be a lot of errors from the Dojo Toolkit.

Include the file **closure.js** in the *first* user code layer (first layer *only*), *before* any user code – it contains a number of necessary changes (especially with regards to property names mapping) to the Dojo Toolkit.

For a detailed map of the layers structure, see the next section.

Step 3: Running the Compiler on the Build Output

The Dojo Build process creates a number of merged JavaScript files, one for each layer. They are named: <*layer*>**.js.uncompressed.js**. The file **dojo.js.uncompressed.js** always exists.

It is necessary that the *uncompressed* version be fed to the Closure Compiler instead because some optimizations made by other optimizers may conflict with the Closure Compiler. Pass each uncompressed layer file, in the correct order, to the "**--js**" parameter of the Closure Compiler. The reader is assumed to understand how to run the Closure Compiler with its numerous command-line options.

The Dojo Build process loads the i18n bundle resources at the **end** of any layer that uses i18n (which should be the **dijit** layer). Therefore, all i18n bundle resource files (usually under the **nls** subdirectory) should be included **before** the **dijit** layer ends, and the only way to do this is to separate **dojo** and **dijit** into different layers.

Special code must be added to the *end* of the **dojo** layer to handle certain i18n objects, because the Dojo Build process does not touch the i18n bundle resource files, and the i18n bundle resources will assume these objects are provided for their declarations:

i18n Code Set #1

```
dojo.provide("dojo.cldr.nls.number");          // If using dojo.number
dojo.provide("dojo.cldr.nls.gregorian");       // If using dojo.date.locale

dojo.provide("dijit.nls.loading");
dojo.provide("dijit.nls.common");
```

Also, include code like the following at the *very beginning* of the **dijit** layer:

i18n Code Set #2

```
dojo._loadedModules["dijit.nls.loading"] = dijit.nls.loading;
dojo._loadedModules["dijit.nls.common"] = dijit.nls.common;
```

This is because **dojo.i18n.getLocalization** actually loads resource bundles from the **dojo._loadedModules** hash, which is skipped by the special Build process to avoid aliases.

Certain Dojo Core modules (e.g. **dojo.number**) depend on i18n, so they must be included *after* the **dojo** layer (i.e. in the **dijit** layer instead). How to know whether any Dojo Core module depends on i18n? After building the project and running it with the layer files, there will be i18n-related errors (e.g. not finding a particular resource bundle) if there is a module that depends on i18n which is included in the **dojo** layer.

As a result, the Dojo Build layers (bracketed in thick borders) and Closure Compiler **--js** loading order should be:

Copyright file with @suppress	
Dojo Core	dojo layer
i18n Code Set #1	
i18n bundle resource files	i18n section
Copyright file with @suppress	
i18n Code Set #2	dijit layer
Dojo modules that require i18n	
Dijit modules	
Copyright file with @suppress	dojox layer
Dojox modules	
Copyright file with @preserve	
closure.js	1st user layer
User Code	
User Code	user layer(s)
User Code	

A sample command line executed on Windows:

```
java -jar compiler.jar
    --compilation_level ADVANCED_OPTIMIZATIONS
    --warning_level VERBOSE
    --create_name_map_files true
    --formatting PRINT_INPUT_DELIMITER
    --jscomp_error=checkTypes
    --jscomp_error=accessControls
    --externs externs.js                    ← Externs file
    --js dojo.js.uncompressed.js
    --js nls\dijit_en.js
    --js nls\dijit_en_us.js
    --js nls\dijit_en_gb.js                 ← Add all necessary i18n bundle files
    --js dijit.js.uncompressed.js
    --js dojox.js.uncompressed.js
    --js user1.js.uncompressed.js           ← The first user layer should include closure.js
    --js user2.js.uncompressed.js           ← User layer files
    --js user3.js.uncompressed.js
    --js_output_file output.js
```

Step 4: Debugging Builds

When things don't work – and it never works 100% on the first try unless you are either very good or very lucky – there are a few debugging steps that should take care of 90% of the problems:

- Rerun the Closure Compiler with the following command line options:
  --formatting PRETTY_PRINT
  --debug true
- Rerun the program, note down the line number of the error and the stack trace.
- In 90% of the case, it will be a "property not found" error – something has been renamed by the Closure Compiler which is eventually accessed via string name. Turning on debug mode will allow you to pin-point the property that is the culprit.
- If this fails to identify the source the problem, you just have to debug it like any normal programming error.[1]

[1] This may get frustrating for users new to the Closure Compiler. The author has faced a case that, when using an external third-party library, the object passed into **addEventHandler** was discovered to have been completely removed by the Closure Compiler as dead code because it thinks that the object's only property, **handleEvent**, is never used! Of course, this reflects poorly on the people maintaining the Closure Compiler, but we have to take what we get…

Step 5: Required "Externs"

The following is a list of "externs" required by the Closure Compiler. This is by no means an exhaustive list. There may be other symbols used throughout the Dojo Toolkit code base that requires adding externs.

```
// Since HTML attributes are mapped to Dijit properties CASE-INSENSITIVE, we need to
// make sure that no property is renamed by the Closure Compiler to a name that differs
// from some HTML attribute by case only!

// In practice, we only need to consider up to two-character attribute names (since the
// Closure Compiler will always try to use the shortest variable names and two
// characters usually suffice. Which means the only danger is really "id"!

// Similarly, be careful with custom attribute names like "to", "at" etc. if they will be loaded
// into a custom Dijit.

var Id = null;     // If this is a mangled name, it will conflict with "id" in an HTML node
var ID = null;     // If this is a mangled name, it will conflict with "id" in an HTML node
var iD = null;     // If this is a mangled name, it will conflict with "id" in an HTML node


// Redundant DOM global externs as "window" may not be the global object

var location = null;
var console = null;
var scroll = null;
var scrollBy = null;
var scrollTo = null;
var resizeTo = null;
var resizeBy = null;
var load = null;
var Components = {// Fire-Fox
   classes: null,
   interfaces: null,
   mozIJSSubScriptLoader: null
};
var Jaxer = null;      // Jaxer


// RequireJS

var define = null;
var require = null;
var currentModule = null;


// HTML5 externs

var localStorage = null;
var JSON = null;
```

```
// Externs for WebKit-specific styles

var webkitTypes =
{
   webkitAnimation: null,
   webkitAnimationDelay: null,
   webkitAnimationDirection: null,
   webkitAnimationDuration: null,
   webkitAnimationFillMode: null,
   webkitAnimationIterationCount: null,
   webkitAnimationName: null,
   webkitAnimationPlayState: null,
   webkitAnimationTimingFunction: null,
   webkitAppearance: null,
   webkitBackfaceVisibility: null,
   webkitBackgroundClip: null,
   webkitBackgroundComposite: null,
   webkitBackgroundOrigin: null,
   webkitBackgroundSize: null,
   webkitBorderBottomLeft: null,
   webkitBorderBottomRight: null,
   webkitBorderHorizontalSpacing: null,
   webkitBorderImage: null,
   webkitBorderRadius: null,
   webkitBorderTopLeft: null,
   webkitBorderTopRight: null,
   webkitBorderVerticalSpacing: null,
   webkitBoxAlign: null,
   webkitBoxDirection: null,
   webkitBoxFlex: null,
   webkitBoxFlexGroup: null,
   webkitBoxLines: null,
   webkitBoxOrdinalGroup: null,
   webkitBoxOrient: null,
   webkitBoxPack: null,
   webkitBoxReflect: null,
   webkitBoxShadow: null,
   webkitBoxSizing: null,
   webkitColumnBreakAfter: null,
   webkitColumnBreakBefore: null,
   webkitColumnBreakInside: null,
   webkitColumnCount: null,
   webkitColumnGap: null,
   webkitColumnRule: null,
   webkitColumnRuleColor: null,
   webkitColumnRuleStyle: null,
   webkitColumnRuleWidth: null,
   webkitColumns: null,
   webkitColumnWidth: null,
   webkitDashboardRegion: null,
   webkitLineBreak: null,
   webkitMarginBottomCollapse: null,
   webkitMarginCollapse: null,
   webkitMarginStart: null,
```

```
  webkitMarginTopCollapse: null,
  webkitMarquee: null,
  webkitMarqueeDirection: null,
  webkitMarqueeIncrement: null,
  webkitMarqueeRepetition: null,
  webkitMarqueeSpeed: null,
  webkitMarqueeStyle: null,
  webkitMask: null,
  webkitMaskAttachment: null,
  webkitMaskBoxImage: null,
  webkitMaskClip: null,
  webkitMaskComposite: null,
  webkitMaskImage: null,
  webkitMaskOrigin: null,
  webkitMaskPosition: null,
  webkitMaskPositionX: null,
  webkitMaskPositionY: null,
  webkitMaskRepeat: null,
  webkitMaskSize: null,
  webkitNbspMode: null,
  webkitPaddingStart: null,
  webkitPerspective: null,
  webkitPerspectiveOrigin: null,
  webkitRtlOrdering: null,
  webkitTapHighlightColor: null,
  webkitTextFillColor: null,
  webkitTextSecurity: null,
  webkitTextSizeAdjust: null,
  webkitTextStroke: null,
  webkitTextStrokeColor: null,
  webkitTextStrokeWidth: null,
  webkitTouchCallout: null,
  webkitTransform: null,
  webkitTransformOrigin: null,
  webkitTransformOriginX: null,
  webkitTransformOriginY: null,
  webkitTransformOriginZ: null,
  webkitTransformStyle: null,
  webkitTransition: null,
  webkitTransitionDelay: null,
  webkitTransitionDuration: null,
  webkitTransitionProperty: null,
  webkitTransitionTimingFunction: null,
  webkitUserDrag: null,
  webkitUserModify: null,
  webkitUserSelect: null,
  animationName: null       // in webkitAnimationEnd
};


// Dojo config

var djConfig = { Any config option used… };
var dojoConfig = { Any config option used… };
```

// The following are needed for dojo.Animation

```
var beforeBegin = null;
var onBegin = null;
var onAnimate = null;
var onEnd = null;
var onPlay = null;
var onPause = null;
var onStop = null;
var play = null;
var pause = null;
var stop = null;
var gotoPercent = null;
```

// The following are needed for Dijit

```
var node = null;
var domNode = null;              // Template attach point
var containerNode = null;        // Template attach point
```

// The following is needed for i18n (each locale that is loaded must be included)
// i18n is particularly troublesome because it loads bundle resource files via text names

```
var nls = {
   loading: { en, en_us, en_gb, <locale>, <locale>, <locale> },
   common: null
};
```

Watch-Out's

Since most of the processing necessary to make the Dojo Toolkit compatible with the Closure Compiler occurs in the Dojo Build process (via the Build script), there are a number of issues to watch out for in addition to the normal (long) list of restrictions regarding writing JavaScript for the Closure Compiler with Advanced Mode.

The Build script does simple text search-and-replace. It does **not** attempt to parse the JavaScript source files. As a result, certain care must be observed to make sure that the necessary statements are caught by the text search patterns.

Note: Anyone using the Closure Compiler in Advanced mode should *always* be careful regarding what this frivolous Compiler will do to user code, so limitations and restrictions are the norm.

Avoid names with "$"

The special Build process tries not to touch variables and properties with names that are not made up of only letters, digits and the underscore character. Using the dollar sign "**$**" as part of a variable/property name will likely bypass any processing.

Although there is no reason why the Build script cannot be written to handle names with "**$**" characters, user code is *strongly* discouraged from using such names, as it makes it difficult to recognize variables and properties with the Closure Compiler's **--debug** mode turned on – which in turn uses "**$**" characters to generate mangled names.[1]

JsDoc's

All user code should use JsDoc's for type-checking by the Closure Compiler.

Dojo-style argument type comments will be automatically converted by the special Build process into JsDoc comments. However, this conversion is not perfect and the expressiveness of the Dojo-style comments is less than that provided by JsDoc.

As the Dojo Toolkit is also experimenting with moving to JsDoc type annotations, it is recommended that JsDoc's be used from the beginning.

The special Build process automatically converts most Dojo-style type comments into JsDoc comments.

[1] Also, writing code sprinkled with dollar signs makes it looks more like jQuery or Prototype than Dojo – and if you want your programs to look like jQuery code, you probably should be using jQuery anyway.

dojo._hasResource, dojo._loadedModules

dojo._hasResource and **dojo._loadedModules** are removed by the special Build process – more accurately *ignored*. This is to avoid creating unnecessary aliases.

User code is *strongly* discouraged from relying on them.

dojo.provide

All **dojo.provide** calls will be turned into **goog.provide** calls (which does the same thing, but is internally in-lined by the Closure Compiler). Putting anything other than a string with a valid scope made up only of letters, digits and the underscore character (i.e. no dollar sign "$") will bypass any processing.

Another way to bypass processing is to do it indirectly via a local variable:

```
var dp = dojo.provide;
dp("foo.bar.Baz");          // This line will be bypassed
```

The Closure Compiler is very fussy about **goog.provide**, however. It will generate an error if a symbol is provided twice – so overlapping **dojo.provide**'s are not supported.

Properties accessed via string name

Some user code has its own functions for creating connections, which in turn calls **dojo.connect** or **this.connect**. For example, the following:

```
function myConnect (obj, evt, scope, method) {
    dojo.connect(obj, evt, scope, method);
}
myConnect(foo, "bar", this, "hello");      // This will not be processed
```

will not be processed because it does not match the strict patterns recognized by the special Build process, and will cause a "property-not-found" error after the real methods "**bar**" and "**hello**" are renamed by the Closure Compiler.

Whenever a user program must access a certain property via its string name, putting the comment **/*remap*/** in front of the *text* causes the special Build process to convert it into a call to **closureGetMappedPropertyName**[1], which then returns the mangled name. For example:

```
myConnect(foo, /*remap*/ "bar", this, this.hello);   // Recommend way to call
```

[1] Don't worry about such long function names. The Closure Compiler will shorten it to a one-character name.

will be converted into:

```
myConnect(foo, closureGetMappedPropertyName({bar:null}), this, this.hello);
```

and the code will work fine. For most types of string-based property access, similar tactics should also work. For example:

```
var prop = [“bar”, “Baz”, “hello”][index];
var value = foo[prop];
```

can be written as:

```
var prop;
switch (index) {
      case 0: prop = /*remap*/ “bar”; break;
      case 1: prop = /*remap*/ “Baz”; break;
      case 2: prop = /*remap*/ “hello”; break;
}
var value = foo[prop];
```

However, it *cannot* be written as:

```
var propnames = [“bar”, “Baz”, “hello”];
var prop = /*remap*/ propnames[index];          // WRONG!!!
var value = foo[prop];
```

The comment **/*remap*/** must be followed immediately by a *text string*.

dojo.connect, this.connect, dojo.hitch
dojo.subscribe, this.subscribe

Calls to **dojo.hitch**, **dojo.connect**, **dojo.subscribe**, **this.connect** (used in Dijit’s) and **this.subscribe** (used in Dijit’s) are the primary culprits when using the Dojo Toolkit with the Closure Compiler in Advanced mode. Handler functions are often passed by string name and the event function is *always* passed by string name.

The special Build process takes care of converting most of these calls into the correct form for processing by the Closure Compiler. For example:

```
dojo.connect(obj, “foo”, scope, “bar”);
```

will be converted to:

```
dojo.connect(obj, closureGetMappedPropertyName({foo:null}), scope, scope.bar);
```

The special global function, **closureGetMappedPropertyName**, is used to map a property name that has been mangled by the Closure Compiler into its original, un-mangled name, or vice versa. This function, our primary workhorse, *depends* upon the fact that the Closure Compiler always converts the same name to the same mangled version (see the Caveat section above).

The pattern used by the special Build process is not very intelligent. It does not parse the JavaScript source code, but simply attempts to match arguments specified either as simple variables and properties (made up only of letters, digits, the underscore character and dots) or simple text strings. Anything more complicated with be bypassed, e.g.:

```
dojo.connect(obj, (happy ? "foo" : "boo"), scope, "bar");
dojo.connect(obj[x], "foo", scope, "bar");
dojo.connect(obj, "foo", scope, getMethodName(scope));
```

Another way to bypass processing is to do it indirectly via a local variable (other than "**d**" which is automatically aliased to **dojo**):

```
var dc = dojo.connect;
dc(obj, "event", scope, "method");    // This line will be bypassed
```

IMPORTANT!!!

The special Build process *skips* event names that are all lower-case and starting with "**on**", or names that are mixed-case and starting with "**webkit**". For example, **onmouseenter**, **onchange**, **webkitAnimationEnd** etc. will all be left alone, as the special Build process assumes that they refer to normal DOM events. Therefore, avoid naming any user events in such manner – always use the recommended Dojo-style: **onChange**, **onMouseEnter** etc.

Dojo classes and dojo.declare

Problem

Declaring classes should be done directly via **dojo.declare**. The special Build process takes care of adding the correct JsDoc comments for type-checking purposes (so that the Closure Compiler will not complain about trying to "**new**" a non-class object).

Dojo classes have special functionalities. In general, properties are read and set via getter/setter functions (typically **get** and **set**); this is to enable ad hoc processing especially on the setter side. Property values are read/set by passing the name of the property in **text** as the first argument to **get** and **set**.

In addition, user-specified functions named **_get***Xxx***Attr** and **_set***Xxx***Attr** will automatically be used as the getter and setter for the "*xxx*" property.

Declaring a public property

Needless to say, such large-scale usage of string property names precludes the type of ad hoc property name mapping via the **/*remap*/** comment. The solution to this is to build a global "property names map"[1] which contains a one-to-one mapping of un-mangled (original) property names to the mangled names provided by the Closure Compiler. The following syntax must be strictly observed:

```
dojo.declare ("my.New.Class", myBase, {     // Base classes in an array also OK
    /*public*/ prop1: value,                // Public property
    /*public*/ prop2: value,                // Public property
    /*public*/ prop3: value,                // Public property
        :
    _prop4: value,                          // Private use, not processed
        :

    method1: function (...) { ... },
    method2: function (...) { ... },
    method3: function (...) { ... },
        :
});
```

The special comment **/*public*/** is used to create the "property names map" for this class. Mappings for getter/setter functions (e.g. **_get***Xxx***Attr**, **_set***Xxx***Attr** etc.) will also be included, *if they are used*.

Any property (properties #1 to #3 in the example above) marked with a **/*public*/** comment is settable declaratively via HTML and automatically propagated into the class by the Dojo parser. Any property not marked with **/*public*/** (property #4 in the example above) is assumed to be private, internal-use only. In order to use such a property with **get** and **set**, the **/*remap*/** comment must be added before the text of the property name, e.g.:

```
widget.set(/*remap*/ "_prop4", "Hello World");
```

Global nature of property names mapping

A *global* property names map is built instead of one map per class. This is done mainly for performance reasons. A by-product is that whenever a property in a class is marked with **/*public*/**, it will be inserted into the *global* map object. Properties in other unrelated classes that happen to have the same *name* as this property will automatically be covered as well, *even though* they may not be marked with **/*public*/**.

In most usage, this is a *good* feature, as the **get**/**set** set of functions always expects *unmangled* text property names, and the difference will be for a getter or setter to work as expected instead

[1] The global variable is named **closurePropertyNamesMap**. Obviously, this name should be avoided in user code!

of failing after compilation. However, for ease of maintenance and debugging, it is always proper to include all the necessary **/*public*/** in each public property.

Bypassing property names mapping

Any class declaration not matching this exact style will be bypassed. Therefore, a way to bypass such processing is to simply do it indirectly via a local variable (other than "**d**" which is automatically aliased to **dojo**):

```
var dc = dojo.declare;
dc("foo.bar.Baz", my.base, { ...   // This will be bypassed
```

Using closureAddPropertyNamesMap

Although it would be beneficial to add such comments to *all* Dijit class declarations, it is too large a change to be considered practical. Therefore, none of the Dijit classes currently have properties marked with **/*public*/**, so in order to use them declaratively "extern" symbols must be provided to the Closure Compiler for *each public property* in order to avoid renaming.

Alternatively, the global function **closureAddPropertyNamesMap**[1] defined in **closure.js** can be used to add entries to the global property names map. The argument is a hash object with unquoted names of properties mapping to the same names in quoted string format, for example to add a "**title**" property to the property names map:

```
closureAddPropertyNamesMap({
    title: "title",
    _getTitleAttr: "_getTitleAttr",    // ← Needed if _getTitleAttr is defined
    _setTitleAttr: "_setTitleAttr"     // ← Needed if _setTitleAttr is defined
});
```

Standard mappings

In **closure.js**, some standard property names are already added:

dijit._WidgetBase: **value**, **widgetId**, **disabled**, **hidden**
dijit.layout.ContentPane: **content**
dijit.Dialog: **duration**, all template attach points and attach events
dijit.DialogUnderlay: **dialogId**

[1] Don't worry about such long function names. The Closure Compiler will shorten it to a one-character name.

Necessary Modifications to Dojo Build Scripts

The following file is updated to add special processing for the Closure Compiler:

dojo-toolkit/util/buildscripts/jslib/buildUtil.js

The following files are added:

dojo-toolkit/util/buildscripts/jslib/dojoGuardStart_Closure
dojo-toolkit/util/buildscripts/jslib/dojoGuardEnd_Closure

Necessary Modifications to Dojo Core

The Dojo Toolkit is not written to satisfy the Closure Compiler's many restrictions and limitations. The special Build process takes care of most of these differences. Unfortunately, some minimal changes still must be made to the Dojo source tree.

Public functions missing argument type comments

Notes: Correct argument type comments aid in type-checking of user code by the Closure Compiler and should be fixed.

Function	File & line number	Comments
dojo.byId	dojo/_base/html.js:50,76	Missing argument type comments.
dojo.create	dojo/_base/html.js:1482	Missing argument type comments.
dojo.place	dojo/_base/html.js:217	Missing argument type comments.

Incorrect/incomplete argument type comments

Notes: Some argument type comments are incorrect or incomplete. In particular, some optional arguments are in the front of the arguments list, so the types of the arguments following these optional arguments must be "promoted" in order to pass the Closure Compiler's type-check. This is most pronounced in **dojo.connect** for which many arguments can be omitted.

Function	File & line number	Comments
dojo._getText	dojo/_base/_loader/ hostenv_browser.js:253	Argument #2 should be marked optional.
dojo.connect	dojo/_base/connect.js:82	Handle optional arguments in the front by promoting types from the back to the front.
dojo.publish	dojo/_base/connect.js:257	Argument #2 should be marked optional.
dojo.subscribe	dojo/_base/connect.js:225	Handle optional argument #2 by promoting types from argument #3 to #2.
dojo.Stateful	dojo/Stateful.js:71	Handle optional argument #1 by promoting types from argument #2 to #1.

Eliminate property accesses via string value

Notes: These properties are accessed via string value in very few places, so it is worthwhile to rewrite them.

Alternative: Provide externs to prevent renaming of these properties.

File & line number	Comments
dojo/_base/browser.js:17 dojo/i18n.js: 245	Eliminate **dojo["require"]** (which is a trick used to avoid the loader loading the dependency) by defining a variable.
dojo/i18n.js:138,149	Eliminate **dojo["provide"]** (which is a trick used to avoid the loader loading the dependency) by defining a variable.
dojo/_base/html.js:1845	Eliminate references to "**addClass**" and "**removeClass**" via an if-statement.
dojo/_base/Deferred.js:180	Eliminate references to "**reject**" and "**resolve**" via an if-statement.

Other Necessary Code modifications

Function	File & line number	Comments
dojo.version.toString()	dojo/_base/_loader/ bootstrap:258	"**with**" keyword is not supported by the Closure Compiler.
dojo.parser.instantiate()	dojo/parser.js:232,258	Add support for property name maps.
"handlers" hash	dojo/_base/xhr.js: 240-305	Change all property names to *quoted* – because **dojo.xhr*** functions map the **handleAs** parameter (passed as string) to a property of this hash to get a handler function.

dojo/parser.js line 232, 258:

Add support for property name maps by reverse-mapping the parameter's short name to its original full name in order to search for attributes with the full name.

This function is defined inside a closure, impossible to override later on – so it is necessary to modify the source.

Starting from line 232 – Create a new "**extra**" object that contains the same parameters but in mangled property names:

```
//>>excludeStart("closure", kwArgs.closure);
if (false)
//>>excludeEnd("closure");
{
    var newextra = {};
    for (var propname in extra) {
        newextra[closureMapPropertyName(clsInfo.cls, propname)] = extra[propname];
    }
    extra = newextra;
}
```

Starting from line 258 – Reverse-map the mangled property names of the class to the un-mangled "full" name and check whether those attributes exist in the node:

```
var fullname = name;
//>>excludeStart("closure", kwArgs.closure);
if (false)
//>>excludeEnd("closure");
{ fullname = closureMapPropertyName(clsInfo.cls, name, true); }
var item = name in mixin ? { value:mixin[name], specified:true } :
                                        attributes.getNamedItem(fullname);
```

Necessary Modifications to Dijit's

Using Dijit modules usual involves providing the correct extern symbols to the Closure Compiler to avoid renaming of properties. This can either be done by scanning through the code tree, or by trial-and-error (usually errors).

The author has not used most of **dijit** (outside of **dijit.Dialog**) to know what changes need to be made (any volunteers?).

Eliminate property accesses via string value

Notes: These properties are accessed via string value in very few places, so it is worthwhile to rewrite them.

Alternative: Provide externs to prevent renaming of these properties.

File & line number	Comments
dijit/_base/wai.js:31	Eliminate references to "**addClass**" and "**removeClass**" by converting the call to **dojo.toggleClass()**

Handle templated widgets

Function	File & line number	Comments
dijit._Templated.prototype._attachTemplateNodes	dijit/_Templated.js:183-212	See below.

dojo/Templated.js line 183-212

Widgets based on **dijit._Templated** class generate properties based on the "**dojoAttachPoint**" (or "**data-dojo-attach-point**") and "**dojoAttachEvent**" (or "**data-dojo-attach-event**") attributes in the template HTML that must be mapped back to the mangled name.

It is necessary that the function **dijit._Templated.prototype._attachTemplateNodes** be changed in **dijit._Templated** instead of being overridden in **closure.js** because some **dijit** classes mix in **dijit._Templated** but not as the main base class – which means that this function gets copied to the new class's prototype.

```
if(attachPoint){
    var point, points = attachPoint.split(/\s*,\s*/);
    while((point = points.shift())){
//>>excludeStart("closure", kwArgs.closure);
        if (false)
//>>excludeEnd("closure");
        { point = closureMapPropertyName(this.constructor, point, true); }
        if(dojo.isArray(this[point])){
            this[point].push(baseNode);
        }else{
            this[point]=baseNode;
        }
        this._attachPoints.push(point);
    }
}

                                                    :
                                                    :
                                                    :

if(event.indexOf(":") != -1){
// oh, if only JS had tuple assignment
    var funcNameArr = event.split(":");
    event = trim(funcNameArr[0]);
    thisFunc = trim(funcNameArr[1]);
//>>excludeStart("closure", kwArgs.closure);
    if (false)
//>>excludeEnd("closure");
    {
        event = closureMapPropertyName(this.constructor, event, true);
        thisFunc = closureMapPropertyName(this.constructor, thisFunc, true);
    }
```

```
}else{
    event = trim(event);
//>>excludeStart("closure", kwArgs.closure);
    if (false)
//>>excludeEnd("closure");
    { event = closureMapPropertyName(this.constructor, event, true); }
}
```

Necessary Modifications to dojox.mobile

The author only used a very limited subset of **dojox.mobile** and none of the other **dojox** modules (any volunteers for the rest?).

Eliminate properties passed/created by string

File & line number	Comments
dojox/mobile/scrollable.js:544,582	Properties accessed via string – replace with if-statements conditioned on "**dir**".
dojox/mobile/_base.js:532	Eliminate **dojo["require"]** (which is a trick used to avoid the loader loading the dependency) by defining a variable.
dojox/mobile/_base.js:830-831	Events should be named "**ontouchstart**" and "**onmousedown**" instead of versions without the "**on**" prefix.
dojox/mobile/_base.js:859-860	Events should be named "**ontouchmove**" and "**ontouchend**" instead of versions without the "**on**" prefix.

Recommended Modifications to Dojo Core, Dijit and Dojox

Enable hard-coding of browser sniffing results

The Closure Compiler can eliminate dead code if it detects that certain variables are constants and that such blocks of code can never be reached. This is especially useful in removing browser-specific code sections.

For example, when compiling an application for WebKit-based mobile browsers, it customary to run Dojo Build with the "**webkitMobile**" flag, which excludes a lot of IE-specific and FireFox-specific code, in addition to removing some wrapper closures (but not all).

Running Dojo Build with the "**closure**" flag automatically creates the following global constants:

```
/** @const */ var ISIE;
/** @const */ var ISFF;
/** @const */ var ISAIR;
/** @const */ var ISWEBKIT;
/** @const */ var ISOPERA;
/** @const */ var ISKHTML;
/** @const */ var ISCHROME;
/** @const */ var ISMAC;
/** @const */ var ISMOZ;
/** @const */ var ISMOZILLA;
/** @type {number} */ var ISSAFARI;
```

and replaces all references to **dojo.isWebKit** with **ISWEBKIT**, **dojo.isIE** with **ISIE**, **dojo.isFF** with **ISFF**, etc. Therefore, all conditional statements depending on these browser-sniffing variables will be replaced by accesses to the corresponding global constants.

The following section of code in **dojo/_base/_loader/hostenv_browser.js** is changed to allow for hard-coding of these browser-detection variables:

```
//>>excludeStart("webkitMobile", kwArgs.webkitMobile);
if(dua.indexOf("Opera") >= 0){ d.isOpera = tv; }
if(dua.indexOf("AdobeAIR") >= 0){ d.isAIR = 1; }
d.isKhtml = (dav.indexOf("Konqueror") >= 0) ? tv : 0;
d.isWebKit = parseFloat(dua.split("WebKit/")[1]) || undefined;
d.isMac = dav.indexOf("Macintosh") >= 0;

if (false)
//>>excludeEnd("webkitMobile");
{
    d.isOpera = 0;
    d.isAIR = 0;
```

```
    d.isWebKit = 1;
    d.isKhtml = 0;
    d.isMac = false;
    // We leave isChrome alone because it is also WebKit-based
}
d.isChrome = parseFloat(dua.split("Chrome/")[1]) || undefined;
```

This change, for example, replaces browser-sniffing calls with hard-coded values when "**webkitMobile**" is defined for the Build. Other hard-coded browser constant sections can be added in a similar way.

When the output source file is processed by the Closure Compiler, all sections of code conditional upon **dojo.isIE**, **dojo.isOpera**, **dojo.isFF** etc. will automatically be removed as dead code, while conditionals depending upon **dojo.isWebKit** will be in-lined.

Eliminate Top-Level Aliases

Notes: Avoid passing in **dojo**, **dijit** and **dojox** as argument to any function. One such usage creates an "alias" which prevents the Closure Compiler from optimizing anything underneath those objects.

File & line number	Comments
dojo/_base/_loader/ bootstrap.js:169-194	Dojo Core is used, so there is no need to reassign **dojo**, **dijit** and **dojox** – which will prevent namespace flattening. "**d**" and "**_d**" are automatically provided as **goog.scope** aliases to "**dojo**", so these separate local variable definitions should be removed. Also, if **dojo._scopeArgs** is not used (should be true in a build) it should be removed to avoid creating aliases. It is suggested that the entire block be removed via an **excludeStart**/**excludeEnd** section.
dojo/_base/_loader/ hostenv_browser.js:88 dojo/_base/_loader/loader.js:7 dojo/_base/Color.js:5 dojo/_base/declare.js:4 dojo/_base/fx.js:9 dojo/_base/html.js:90	Remove "**d = dojo;**" and "**_d = dojo;**" (via an **excludeStart**/**excludeEnd** section) since everything in Dojo Core should be wrapped by a **goog.scope** with "**d**" and "**_d**" aliased to "**dojo**".[1] The need for these should go away when Dojo moved to real AMD modules format.

[1] Some of these statements are already removed by the **webkitMobile** Build flag, but not all.

dojo/_base/lang.js:4 dojo/_base/NodeList.js:6 dojo/_base/xhr.js:6 dojo/fx.js:10 dojo/html.js:9 dojo/parser.js:9 dojo/robot.js:21,65 dojo/uacss.js:14 dijit/_WidgetBase.js:373 dijit/_Widget.js:389	
dojo/_base/_loader/ hostenv_browser.js:326	Remove the statement "**d = null;**" via an **excludeStart**/**excludeEnd** section.
dojo/_base/html.js: 1079	Rewrite to remove " **... in d ? ...**" in order to avoid blocking optimization of the entire **dojo** tree.
dojo/i18n.js	Change code to avoid loading resource bundles from the **dojo._loadedModules** hash – which is eliminated by the special Build process to avoid creating aliases.
dojo/_base/_loader/loader.js dijit/_base/focus.js dijit/_base/wai.js	Replace **dojo.mixin** calls with individual assignments to the **dijit** object – using **dojo.mixin** creates an alias to **dijit** that prevents flattening of the **dijit** namespace.
dojox/mobile/ scrollable.js:61-66	Remove (via an **excludeStart**/**excludeEnd** section) the if-statement and everything in the else clause because Dojo Core is used and there is no need to reassign **dojo** and **dojox** – which will prevent namespace flattening.

dojo/_base/html.js line 1079**:**

```
dojo._isBodyLtr = function(){
    return (d._bodyLtr === undefined) ?
        d._bodyLtr = (d.body().dir || d.doc.documentElement.dir || "ltr")
            .toLowerCase() == "ltr" // Boolean
    : d._bodyLtr;
};
```

Eliminate property accesses via string value

File & line number	Comments
dojo/_base/array.js: 255-264	For **webkitMobile** builds, several **dojo** array functions (e.g. **forEach**) are remapped to native versions on the **Array** prototype. Properties are added to the **dojo** object by string names – dangerous but in this case harmless due to these keywords being standard externs in the Closure Compiler. However, none of these functions can be virtualized. The whole block should be rewritten to assign these functions directly to the properties of the **dojo** object.
dojo/_base/NodeList.js: 231-244	Some functions in **NodeList** map to their **dojo** counterparts. The whole section should be rewritten to assign these functions directly from the properties of the **dojo** object instead of using string property access.

dojo/_base/array.js lines 255-264:

```
//>>excludeStart("webkitMobile", kwArgs.webkitMobile);
if (false)
//>>excludeEnd("webkitMobile");
{
["indexOf", "lastIndexOf", "forEach", "map", "some", "every", "filter"].forEach(
    function(name, idx){
        var proto = Array.prototype[name];

        var func = function(/*Array*/arr, /*Function|String*/callback, /*Object?*/thisObj){
            if((idx > 1) && (typeof callback == "string")){
                callback = new Function("item", "index", "array", callback);
            }
            return proto.call(arr, callback, thisObj);
        };

        switch (idx) {
            case 0: dojo.indexOf = func; break;
            case 1: dojo.lastIndexOf = func; break;
            case 2: dojo.forEach = func; break;
            case 3: dojo.map = func; break;
            case 4: dojo.some = func; break;
            case 5: dojo.every = func; break;
            case 6: dojo.filter = func; break;
        }
    });
}
```

dojo/_base/NodeList.js lines 231-244

```
nlp.slice = function(){ return this._wrap(ap.slice.apply(this, arguments), this); };
nlp.splice = function(){ return this._wrap(ap.splice.apply(this, arguments), null); };
// concat should be here but some browsers with native NodeList have problems with it

// add array.js redirectors
nlp.indexOf=function(){return d.indexOf.apply(null,
                        [this].concat(aps.call(arguments, 0))); };
nlp.lastIndexOf=function(){return d.lastIndexOf.apply(null,
                        [this].concat(aps.call(arguments, 0))); };
nlp.every=function(){return d.every.apply(null,
                        [this].concat(aps.call(arguments, 0))); };
nlp.some=function(){return d.some.apply(null,
                        [this].concat(aps.call(arguments, 0))); };

// add conditional methods
nlp.attr = adaptWithCondition(dojo.attr, magicGuard);
nlp.style = adaptWithCondition(dojo.style, magicGuard);

// add forEach actions
nlp.connect = adaptAsForEach(dojo.connect);
nlp.addClass = adaptAsForEach(dojo.addClass);
nlp.removeClass = adaptAsForEach(dojo.removeClass);
nlp.replaceClass = adaptAsForEach(dojo.replaceClass);
nlp.toggleClass = adaptAsForEach(dojo.toggleClass);
nlp.empty = adaptAsForEach(dojo.empty);
nlp.removeAttr = adaptAsForEach(dojo.removeAttr);
```

Going All The Way – More Risky Modifications to Remove "dojo" Itself

In order to flatten the "**dojo**" root namespace itself, the modifications involved are more significant. They are outlined below.

WARNING!!!
Apply these modifications at your own risk. They are less than trivial changes and must go through extensive testing to ensure that they are safe.

dojo/_base/_loader/loader.js: 248 (**dojo.addOnLoad**)

dojo.addOnLoad does not appear to depend on "**this**" pointing to "**dojo**". Pass **null** as first argument to the **apply**/**call** function to avoid creating an alias to "**dojo**".

dojo/_base/_loader/loader.js: 512 (**dojo.platformRequire**)

dojo._loadModule does not appear to depend on "**this**" pointing to "**dojo**". Pass **null** as first argument to **apply** in order to avoid creating an alias to "**dojo**".

dojo/_base/_loader/loader.js: 533 (**dojo.requireIf**)

dojo.require does not appear to depend on "**this**" pointing to "**dojo**". Pass **null** as first argument to **apply** in order to avoid creating an alias to "**dojo**".

dojo/_base/_loader/bootstrap.js: 453 (**dojo.eval**)
dojo/_base/_loader/loader.js: 116 (**dojo._loadUri**)

Rewrite **dojo["eval"]** as **dojo.eval** in order to avoid creating an alias to "**dojo**".

dojo/_base/fx.js: 602

dojo.style does not appear to depend on "**this**" pointing to "**dojo**". Pass **null** as first argument to **apply** in order to avoid creating an alias to "**dojo**".

dojo/_base/lang.js: 122 (**dojo.hitch**)

Nobody really calls **dojo.hitch** without a starting scope, expecting it to be "**dojo**" itself…

Pass **null** as first argument to **apply** in order to avoid creating an alias to "**dojo**".

WARNING!!!

This will break strange code such as: **dojo.hitch(null, "connect", ...)**[1] because it has more than 2 arguments and thus gets passed to **dojo._hitchArgs()** which should then make the scope object to be "**dojo**" when it sees the **null**. This will no longer happen after compilation because the "**dojo**" object may be removed.

dojo/_base/lang.js: 230 (**dojo.partial**)

It is not necessary to call **dojo.hitch** with a context due to modifications in **dojo/_base/lang.js**.

Pass **null** as first argument to **apply** in order to avoid creating an alias to "**dojo**".

dojo/_base/Deferred.js: 293, 297, 301

It is not necessary to call **dojo.hitch** with a context due to modifications in **dojo/_base/lang.js**.

Pass **null** as first argument to **apply** in order to avoid creating an alias to "**dojo**".

dojo/_base/NodeList.js: 301

Rewrite to reference to "**dojo.string**" in order to avoid blocking optimization of everything under **dojo.string**. Does this rewrite work:

```
var templateFunc = content.templateFunc;
try {
    if (!templateFunc) templateFunc = dojo.string.substitute;
} catch (e) {
    templateFunc = null;
}
```

[1] However, if the user writes strange code like this (I can't think of a valid use case where he/she cannot pass in "**dojo**" as the first argument) and expects to compile this with the Closure Compiler under *Advanced* mode, he/she absolutely **deserves** this to be done to him/her!

dojo/parser.js: 80-82 (**dojo.parser** constructor)

dojo.connect is called on **dojo.extend**, creating an alias to "**dojo**". Does this rewrite work:

```
//>>excludeStart("closure", kwArgs.closure);
    d.connect(d, "extend", function(){
        instanceClasses = {};
    });

    if (false)
//>>excludeEnd("closure");
    {
        var origfunc = d.extend;
        d.extend = function(/*Object*/ constructor, /*Object...*/ props) {
            origfunc.apply(null, arguments);
            instanceClasses = {};
        }
    }
```

dijit/_Widget.js: 8-13 (deferred connects)

dojo.connect is called on **dojo._connect**, creating an alias to "**dojo**". Does this rewrite work:

```
//>>excludeStart("closure", kwArgs.closure);
dojo.connect(dojo, "_connect",
    function(/*dijit._Widget*/ widget, /*String*/ event){
        if(widget && dojo.isFunction(widget._onConnect)){
            widget._onConnect(event);
        }
    });

if (false)
//>>excludeEnd("closure");
{
    var origfunc = dojo._connect;
    dojo._connect = function(obj, event, context, method) {
        var retvalue = origfunc.apply(null, arguments);

        if(obj && dojo.isFunction(obj._onConnect)) {
            obj._onConnect(event);
        }

        return retvalue;
    };
}
```