Google Tools to Add Power to Your JavaScript

Closure

The Definitive Guide

O'REILLY®

Michael Bolin

Closure: The Definitive Guide
by Michael Bolin

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Simon St.Laurent and Julie Steele
Production Editor: Kristen Borg
Copyeditor: Nancy Kotary
Proofreader: Kristen Borg

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
September 2010: First Edition.

ISBN: 978-1-449-38187-5

[M]

1283888246

Table of Contents

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Foreword

I was sitting on a balcony on the west side of Manhattan, sipping on a warm glass of scotch with a few others. Michael Bolin joined us. Michael wrote this book. At the time, Michael was working on Google Tasks. I was the tech lead on our JavaScript optimizer, later named Closure Compiler. Michael didn't join us to talk about JavaScript optimization though. He didn't want to talk scotch either, to his detriment. He wanted to talk JavaScript-driven text editing, and thus he wanted to talk to Julie.

You will receive a proper introduction to Julie in Chapter 9, but for now, just know that Julie is our expert on how text editors are implemented in each web browser.

Michael found that, when managing a task list in a web browser, you want a few features built into your plain text editor. You want to make words bold for emphasis. You want a keyboard shortcut to move your cursor to the next task item. He didn't want to have to write a whole editor. He just wanted a few tweaks on top of what the browser provides, to make the experience smoother for the user. How would you implement this?

Julie explained that there are many, many choices for such a thing. "Should you use a textarea?" "Should you use a contentEditable region?" "Should you rely on the browser's built-in rich text functions?" "Should you implement the 'bold' function in JavaScript?" "How do you make sure the cursor ends up on the right line, given that browsers each implement cursor selection differently?" "Should you put all the text editing in an iframe to isolate it from the rest of the page?"†

"Is there code you can reuse for this?"

You don't really want to implement all these things from scratch. A lot of them will need to call into esoteric browser APIs in complex ways. Many of those APIs are buggy, poorly documented, or simply do not perform very well. For some of those APIs, it's easier to read the browser source code than to find reasonable documentation.

† Fun fact: as the number of JavaScript developers in a room increases, the probability that someone will suggest "iframes" as the solution to your problem asymptotically approaches 1.

You'll find answers to many of those specific questions throughout this book. But I think the question that the book is most interested in (and rightly so) is about how to make it easy to reuse code for Ajax apps. It spins off into a few other equally substantial questions.

How do you share JavaScript code? How do you organize large amounts of common JavaScript, often built for highly specialized tasks? How do you weigh one team's need for boatloads of new features and customizations against another team's need to keep the size of the JavaScript they're sending to the user small?

The Closure Tools were designed to solve many of these problems. Maybe that's understating the point. These problems are at the very core of their design. Many of the tools were started by our friends on Gmail. Gmail began as a relatively modest JavaScript app. Then they added more and more features, and watched it grow beyond any hope of control or maintainability. Frederick P. Brooks, Jr., famously described large-system programming as "a tar pit, and many great and powerful beasts have thrashed violently in it." In a language like JavaScript, a highly dynamic environment where almost everything can be mutated and there's no standard way to specify contracts (type checking or otherwise), the tar is fast and can suck down even a small group of developers.

The Closure Tools developers tried to bring "closure" to this mess. (I agree the pun is terrible. It is not mine.) They followed strict idioms for namespacing code and defining classes. They adopted ECMAScript 4's type language for specifying contracts. The compiler forced the developer to declare their variables, and emitted warnings for other frowned-upon idioms. The Closure Tools, in short, tried to add some structure to the language. Many engineering teams at Google found this structure useful, and built their products on top of it.

A long time passed. The Closure Tools remained proprietary for years. This wasn't meant to be. Both the compiler and the libraries were always designed to be open source projects. But more importantly, they were designed for building Google apps first, and to be open source projects second. So releasing them publicly took a back seat to other things.

Have you ever tried to publicly open up the code of a proprietary project? Several engineers had tried to release Closure Compiler. They had all given up. It is surprisingly difficult. There are two major parts. First, you have to release the code: port it to a public build system like Apache Ant, remove all of its nonopen dependencies, and rewrite any dependencies that you can't remove. Second, you have to write documentation: loads of documentation.

You can imagine how skeptical I was when Michael first came by my desk to talk about making Closure Compiler an open source project. This was early 2009. By this point, "publicly releasing Closure Compiler" was the sort of daunting chore that you've procrastinated forever and a half. We'd work on it for a month, realize that we seemed no

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Preface

JavaScript borrows many great ideas from other programming languages, but its most unique, and perhaps most powerful, feature is that any code written in JavaScript can run as-is in any modern web browser. This is a big deal, and it is unlikely to change anytime soon.

As web browsers improve and become available on more devices, more applications are being ported from desktop applications to web applications. With the introduction of HTML5, many of these applications will be able to work offline for the first time. In order to create a superior user experience, much of the logic that was previously done on the server will also have to be available on the client. Developers who have written their server logic in Java, Python, or Ruby will have to figure out how to port that server logic to JavaScript. Tools like Google Web Toolkit, which translate Java to JavaScript can help with this, though such tools are often clumsy because the idioms from one programming language do not always translate smoothly into that of another. However, if your server code is written in JavaScript, this is not an issue.

I believe that the use of server-side JavaScript (SSJS) is just beginning. Previously, most implementations of JavaScript were too slow to be considered a viable option for server code. Fortunately, the recent competition among browser vendors to have the fastest JavaScript engine makes that difference far less significant (*http://shootout.alioth.debian.org*).

Because of the emerging support for offline web applications, it is compelling to write both the client and the server in the same programming language to avoid the perils associated with maintaining parallel implementations of the same logic. Because it is extremely unlikely that all of the major browser vendors will adopt widespread support for a new programming language, that will continue to force the client side of a web application to be written in JavaScript, which in turn will pressure developers to write their servers in JavaScript as well. This means that the size of the average JavaScript codebase is likely to increase dramatically in the coming years, so JavaScript developers will need better tools in order to manage this increased complexity. I see Closure as the solution to this problem.

Closure is a set of tools for building rich web applications with JavaScript, and brings with it a new approach to writing JavaScript and maintaining large JavaScript applications. Each tool in the suite is designed to be used independently (so jQuery developers can make use of the Closure Compiler and Closure Templates, even if they are not interested in the Closure Library), but they are most effective when used together.

Many JavaScript toolkits today focus on DOM utilities and UI widgets. Such functionality is incredibly useful when building the interface for a web application, but the emergence of SSJS will require an equivalent effort in building server-side JavaScript libraries. There, the focus is likely to be on data structures and efficient memory usage, both of which are already woven into the Closure framework.

I believe that Closure will play an important part in making web applications faster and more reliable. As an active user of the Web, I have a vested interest in making sure this happens. That's why I had to write this book. Rather than document every API in Closure, I have tried to provide detailed explanations for the most commonly used APIs, particularly those that are unique to the Closure approach.

Indeed, learning Closure will change the way you develop JavaScript applications.

My Experiences with Closure

When I worked at Google from 2005 to 2009, I used Closure to help build Google Calendar and Google Tasks. When the initial work on Calendar was done in 2005, only the Compiler was available, and it was (and is) known internally as the JavaScript Compiler. At the time, there were a number of common JavaScript utilities that teams would copy from one another. This led to many forked versions, so improvements to one copy did not propagate to the others.

Meanwhile, the JavaScript codebase for Gmail had grown so large and complex that developers complained that it was too hard for them to add new features. This triggered a rewrite of the Gmail client, which precipitated the development of the two other major tools in the Closure suite: the Library and Templates. The Library was simply named "Closure," as it was a play on the programming construct used so frequently in JavaScript, as well as the idea that it would bring "closure" to the nightmare that was JavaScript development at Google.

Like many other JavaScript toolkits, the goal of Closure was to provide a comprehensive cross-browser library. Instead of adopting an existing solution, such as Dojo, Google decided to roll its own. By having complete control of its library, it could ensure that the API would be stable and that the code would work with its (then) secret weapon: the Closure Compiler. This made it possible to buck the trend established by libraries like Prototype that encouraged the use of absurdly short function names. In Closure, nondescript function names such as `$` were eschewed in favor of more descriptive ones because the Compiler would be responsible for replacing longer names with shorter ones.

The build system at Google was amended to express dependencies between JavaScript files (these relationships are reflected by `goog.provide()` and `goog.require()` statements in the Closure Library). For the first time, dependencies were organized into well-named packages, which introduced a consistent naming scheme and made utilities easier to find. In turn, this made code reuse more straightforward, and the Library quickly achieved greater consistency and stability than the previous dumping ground of JavaScript utilities. This new collection of common code was far more trustworthy, so teams started to link to it directly rather than fork their own versions, as they were no longer afraid that it would change dramatically out from under them.

Finally, Closure Templates (known internally as Soy) were created to address the problem that most existing templating systems were designed to generate server code, but not JavaScript code. The first version of Soy generated only JavaScript, but it was later extended to generate Java as well, to provide better support for the "HTML Decorator" pattern described in Chapter 8, *User Interface Components*.

By the time I started work on Google Tasks, these tools had matured considerably. They were invaluable in creating Tasks. While the Calendar team was busy replacing their original utility functions with Closure Library code and swapping out their home-brewed (or Bolin-brewed) template solution with Soy, I was able to make tons of progress on Tasks because I was starting with a clean slate. Because Gmail has been stung by hard-to-track-down performance regressions in the past, the barrier for getting code checked in to Gmail is high. In integrating Tasks with Gmail, I was forced to gain a deeper understanding of the Closure Tools so I could use them to optimize Tasks to the satisfaction of the Gmail engineers. Later, when I integrated Tasks in Calendar, I learned how to organize a sizable JavaScript codebase so it could be incorporated by even larger JavaScript projects.

One of my major takeaways from using Closure is that trying to address limitations of the JavaScript programming language with a JavaScript library is often a mistake. For example, JavaScript does not have support for multiline strings (like triple-quote in Python), which makes it difficult to create templates for HTML. A bad solution (which is the one I created for Google Calendar back in 2005 that they were still trying to phase out so they could replace it with Soy in 2009) is to create a JavaScript library like jQuery Templates (*http://plugins.jquery.com/project/jquerytemplate*). Such a library takes a string of JavaScript as the template and parses it at runtime with a regular expression to extract the template variables. The appeal, of course, is that implementing something like jQuery Templates is fairly easy, whereas implementing a template solution that is backed by an actual parser is fairly hard (Closure Templates does the latter). In my experience, it is much better to create a tool to do exactly what you want (like Closure Templates) than it is to create a construct within JavaScript that does almost what you want (like jQuery Templates). The former will almost certainly take longer, but it will pay for itself in the long run.

Audience

As this is a book about Closure, a suite of JavaScript tools, it assumes that you are already familiar with JavaScript. Nevertheless, because so many JavaScript programmers learn the language by copying and pasting code from existing websites, Appendix B is included to try to identify incorrect assumptions you may have made about JavaScript when coming from your favorite programming language. Even those who are quite comfortable with the language are likely to learn something.

Other than the Closure Tools themselves, this book does not assume that you are already familiar with other JavaScript tools (such as JSLint and YUI Compressor) or libraries (such as Dojo and jQuery), though sometimes parallels will be drawn for the benefit of those who are trying to transfer their knowledge of those technologies in learning Closure. The one exception is Firebug, which is a Firefox extension that helps with web development. In addition to being considered an indispensable tool for the majority of web developers, it must be installed in order to use the Closure Inspector. Unlike the other tools in the suite, the use of the Closure Inspector is tied to a single browser: Firefox. Because Firebug is updated frequently and has comprehensive documentation on its website, this book does not contain a tutorial on Firebug because it would likely be outdated and incomplete. *http://getfirebug.com* should have everything you need to get started with Firebug.

Finally, this book makes a number of references to Java when discussing Closure. Although it is not necessary to know Java in order to learn Closure, it is helpful to be familiar with it, as there are elements of Java that motivate the design of the Closure Library. Furthermore, both Closure Templates and the Closure Compiler are written in Java, so developers who want to modify those tools will need to know Java in order to do so. This book will not teach you Java, though a quick search on Amazon will reveal that there of hundreds of others that are willing to do so.

ECMAScript Versus JavaScript

This book includes several references to ECMAScript, as opposed to JavaScript, so it is important to be clear on the differences between the two. ECMAScript is a scripting language standardized by Ecma International, and JavaScript is an implementation of that standard. Originally, JavaScript was developed by Netscape, so Microsoft developed its own implementation of ECMAScript named JScript. This means that technically, "ECMAScript" should be used to refer to the scripting language that is universally available on all modern web browsers, though in practice, the term "JavaScript" is used instead. To quote Brendan Eich, the creator of JavaScript: "ECMAScript was always an unwanted trade name that sounds like a skin disease." To be consistent with colloquial usage (and honestly, just because it sounds better), JavaScript is often used to refer to ECMAScript in this book.

However, ECMAScript is mentioned explicitly when referring to the standard. The third edition of the ECMAScript specification (which is also referred to as ES3) was published in December 1999. As it has been around for a long time, it is implemented by all modern web browsers. More recently, the fifth edition of the ECMAScript specification (which is also referred to as ES5) was published in December 2009. (During that 10-year period, there was an attempt at an ES4, but it was a political failure, so it was abandoned.) As ES5 is a relatively new standard, no browser implements it fully at the time of this writing. Because Closure Tools are designed to create web applications that will run on any modern browser, they are currently designed around ES3. However, the Closure developers are well aware of the upcoming changes in ES5, so many of the newer features of Closure are designed with ES5 in mind, with the expectation that most users will eventually be using browsers that implement ES5.

Using This Book

This book explains all of the Closure Tools in the order they are most likely to be used.

- Chapter 1, *Introduction to Closure*, introduces the tools and provides a general overview of how they fit together with a complete code example that exercises all of the tools.

When working on a JavaScript project, you will spend the bulk of your time designing and implementing your application. Because of this, the majority of the book is focused on how to leverage the Closure Library and Closure Templates to implement the functionality you desire. Of all the topics covered in this part of the book, the rich text editor is the one that appears most frequently in the Closure Library discussion group. To that end, I recruited `goog.editor` expert Julie Parent as a contributing author, so fortunately for you and for me, Julie wrote Chapter 9.

- Chapter 2, *Annotations for Closure JavaScript*, explains how to annotate JavaScript code for use with the Closure Compiler.
- Chapter 3, *Closure Library Primitives*, provides documentation and commentary on every public member of `base.js` in the Closure Library.
- Chapter 4, *Common Utilities*, surveys functionality for performing common operations with the Closure Library, such as DOM manipulation and user agent detection.
- Chapter 5, *Classes and Inheritance*, demonstrates how classes and inheritance are emulated in Closure.
- Chapter 6, *Event Management*, explains the design of the Closure Library event system and the best practices when using it.
- Chapter 7, *Client-Server Communication*, covers the various ways the `goog.net` package in the Closure Library can be used to communicate with the server.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

way back in 2005 has become a critical widget for many Google Apps today (most notably, Gmail). If they gave out doctorates for the field of "little-known browser bugs that make rich text editing in the browser nearly impossible," then Julie would be a leader in the field and Chapter 9 could have been used as her dissertation. Julie, thank you so much for putting the same amount of diligence into writing your chapter as you did in developing the rich text editor in the first place.

Next, I owe a tremendous amount of thanks (and a nice bottle of scotch) to Nick Santos, who has been a phenomenal technical reviewer. He responded to the call for reviewers with alacrity and his enthusiasm in the project never waned. In doing a review of this book, Nick effectively engaged in a 35,000-line code review, and provided so many corrections and helpful suggestions that this book probably would not even be worth reading if Nick had not read it first. In addition to all of his work as a reviewer, Nick played (and continues to play) an active role in open-sourcing the Closure Compiler as well as its development. You can see the breadth and depth of Nick's knowledge in the Closure Compiler discussion group, as he is an extremely active member there, as well.

In addition to Nick, I was fortunate enough to have two other Google engineers who helped build pieces of the Closure Tools suite to participate in the review process. Erik Arvidsson (who co-created the Closure Library with Dan Pupius—thanks, Dan!) provided lots of valuable feedback on the chapters on the Library. Likewise, the creator of Closure Templates, Kai Huang, provided detailed criticisms of the chapter on Soy. Many thanks to both Erik and Kai for lending their time and expertise to ensure that the story of their work was told correctly.

As Nick explained in the foreword, taking a closed source project and turning it into an open source one is a lot of work, so I would also like to recognize those who played an important role in that process. Nathan Naze, Daniel Nadasi, and Shawn Brenneman all pitched in to open source the Closure Library. Robby Walker and Ojan Vafai also helped out by moving the rich text editor code into the Library so that it could be open-sourced, as well. Extra thanks to Nathan for continuing to manage the open-sourcing effort and for giving talks to help get the word out about the Library. It is certainly an example of well-spent 20% time at Google.

In that same vein, I would also like to thank Dan Bentley for helping ensure that all of this Closure code made it out into the open. Google is lucky to have him working in their Open Source Programs Office, as his genuine belief and interest in open source benefits the entire open source community.

I would also like to thank my former teammates on the Closure Compiler team who all contributed to the open source effort as well as Compiler development: Robert Bowdidge, Alan Leung, John Lenz, Nada Amin, and Antonio Vincente. Also, thanks to our manager, Ram Ramani, who supported this effort the whole way through and helped coordinate the open source launch. I also want to give credit to our intern, Simon Mathieu, who worked with me to create the Closure Compiler Service.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

This can have dramatic effects on the Compiler's ability to minify code, as a simple minification experiment finds that Closure Library code can be 85 percent smaller when using the Closure Compiler in place of the YUI Compressor (*http://blog.bolinfest.com/2009/11/example-of-using-closure-compiler-to.html*).

The Closure Library is also implemented with a strong emphasis on performance and readability. It is frugal in creating objects, but generous in naming and documenting them. It also has an elegant event system, support for classes and inheritance, and a broad collection of UI components, including a rich text editor. Closure Library code is regularly tested across browsers, and to the extent that it can, will also work in non-browser JavaScript environments, such as Rhino (*http://www.mozilla.org/rhino/*) and the Microsoft Windows Script Host. Because the Library is a resource for Google engineers first and an open source project second, it is a safe bet that every line of code in the Library was developed to support at least one Google product. The style of the Library will first be introduced in Chapter 2, and the functionality of the Library will be covered in the following eight chapters.

Closure Templates

Closure Templates provide an intuitive syntax for creating efficient JavaScript functions (or Java objects) that generate HTML. This makes it easier to create a large string of HTML that can in turn be used to build up the DOM. Unfortunately, most programming languages do not have native support for templates, so creating a separate templating solution is a common practice for web frameworks (J2EE has JSP, Python developers frequently use Django's template system, etc.). A unique aspect of Closure Templates is that the same template can be compiled into both Java and JavaScript, so those running servers written in Java (or JavaScript!) can use the same template on both the server and the client. The benefits of this, along with Closure Templates, will be covered in Chapter 11.

Closure Compiler

The Closure Compiler is a JavaScript optimizing compiler: it takes JavaScript source code as input and produces behaviorally equivalent source code as output. That is, when the output code is used in place of the input code, the observable effect will be the same (though the output code is likely to execute faster than the original). As a simple example, if the input code were:

```
/**
 * @param {string} name
 */
var hello = function(name) {
  alert('Hello, ' + name);
};
hello('New user');
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
/**
 * An array of keys. This is necessary for two reasons:
 *   1. Iterating the keys using for (var key in this.map_) allocates an
 *      object for every key in IE which is really bad for IE6 GC perf.
 *   2. Without a side data structure, we would need to escape all the keys
 *      as that would be the only way we could tell during iteration if the
 *      key was an internal key or a property of the object.
 *
 * This array can contain deleted keys so it's necessary to check the map
 * as well to see if the key is still in the map (this doesn't require a
 * memory allocation in IE).
 * @type {!Array.<string>}
 * @private
 */
this.keys_ = [];
```

Now that Microsoft has provided a patch for the problem with IE6, such micromanagement of string allocation is less compelling. However, as more mobile devices are running web browsers with fewer resources than their desktop equivalents, attention to memory management in general is still merited.

Make It Possible to Catch Errors at Compile Time

The Closure Compiler is not the first tool to try to identify problems in JavaScript code by performing static checks; however, there is a limit to how much can be inferred by the source code alone. To supplement the information in the code itself, the Compiler makes use of developer-supplied annotations which appear in the form of JavaScript comments. These annotations are explained in detail in Chapter 2.

By annotating the code to indicate the parameter and return types of functions, the Compiler can identify when an argument of the incorrect type is being passed to a function. Similarly, annotating the code to indicate which data are meant to be private makes it possible for the Compiler to identify when the data are illegally accessed. By using these annotations in your code, you can use the Compiler to increase your confidence in your code's correctness.

Code Must Work Without Compilation

Although the Compiler provides many beneficial transformations to its input, the code for the Closure Library is also expected to be able to be run without being processed by the Compiler. This not only ensures that the input language is pure JavaScript, but also makes debugging easier, as it is always possible to use the deobfuscated code.

Code Must Be Browser-Agnostic

The Closure Library is designed to abstract away browser differences and should work in all modern browsers (including IE6 and later). It should also work in non-browser environments, such as Rhino and the Windows Script Host (though historically the

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

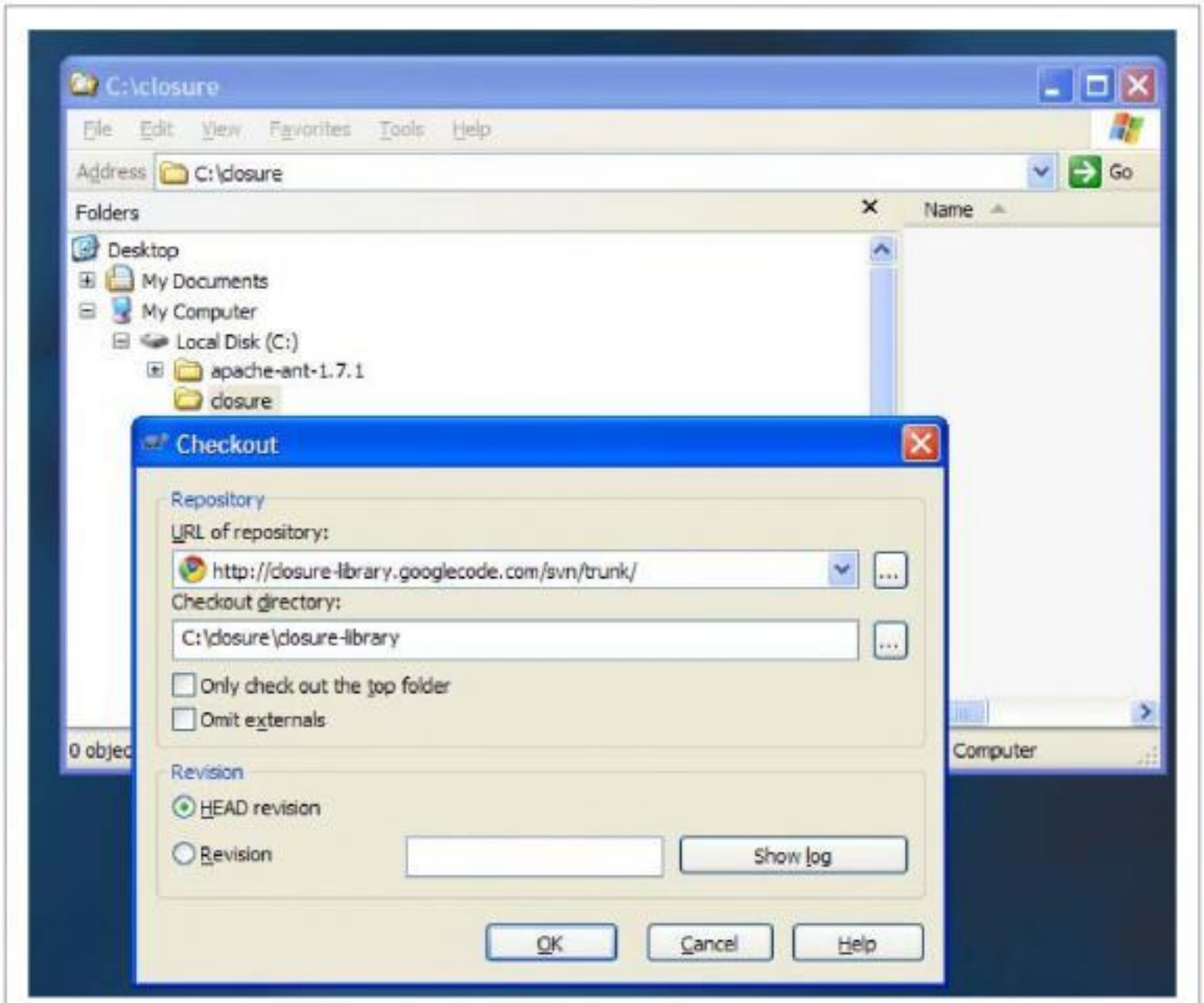


Figure 1-2. Using TortoiseSVN to check out the Closure Library on Windows.

The Closure Library also contains the Closure Testing Framework. Open the URI `file:///C:/closure/closure-library/all_tests.html` in a web browser and press the "Start" button to kick off the test suite. At the time of this writing, not all of the tests pass, so do not be worried that you downloaded a "bad" version of the Library if you see several test failures. The status of each failure is tracked as an issue on *http://code.google.com/p/closure-library/issues/list*.

Closure Templates

The primary binary for Closure Templates is used to compile templates into JavaScript. It can be downloaded from *http://closure-templates.googlecode.com/files/closure-templates-for-javascript-latest.zip*.

It is also fairly easy to build the Templates binary from source. Download the code using Subversion by following the Closure Library example, but use `http://closure-templates.googlecode.com/svn/trunk/` as the URL of the repository to check out and `closure-templates` as the destination. All Closure Templates binaries can be built using

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
{template .welcome}
  <h1 id="greeting">{$greeting}</h1>
  The year is {$year}.
{/template}
```

Assuming that `SoyToJsSrcCompiler.jar` is in `closure-templates/build/`, run the following command from your `hello-world` directory in the Command Prompt on Windows or the Terminal on Mac or Linux:

```
java -jar ../closure-templates/build/SoyToJsSrcCompiler.jar \
    --outputPathFormat hello.soy.js \
    --shouldGenerateJsdoc \
    --shouldProvideRequireSoyNamespaces hello.soy
```

This should generate a file named `hello.soy.js` with the following content:

```
// This file was automatically generated from hello.soy.
// Please don't edit this file by hand.

goog.provide('example.templates');

goog.require('soy');
goog.require('soy.StringBuilder');


/**
 * @param {Object.<string, *>=} opt_data
 * @param {soy.StringBuilder=} opt_sb
 * @return {string|undefined}
 * @notypecheck
 */
example.templates.welcome = function(opt_data, opt_sb) {
  var output = opt_sb || new soy.StringBuilder();
  output.append('<h1 id="greeting">', soy.$$escapeHtml(opt_data.greeting),
      '</h1>The year is ', soy.$$escapeHtml(opt_data.year), '.');
  if (!opt_sb) return output.toString();
};
```

Now update `hello.js` so it uses the function available in `hello.soy.js` and includes another `goog.require()` call to reflect the dependency on `example.templates`:

```
goog.provide('example');

goog.require('example.templates');
goog.require('goog.dom');

example.sayHello = function(message) {
  var data = {greeting: message, year: new Date().getFullYear()};
  var html = example.templates.welcome(data);
  goog.dom.getElement('hello').innerHTML = html;
};
```

In order to use `hello.soy.js`, both it and its dependencies must be loaded via `<script>` tags in the `hello.html` file:

```
<!doctype html>
<html>
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

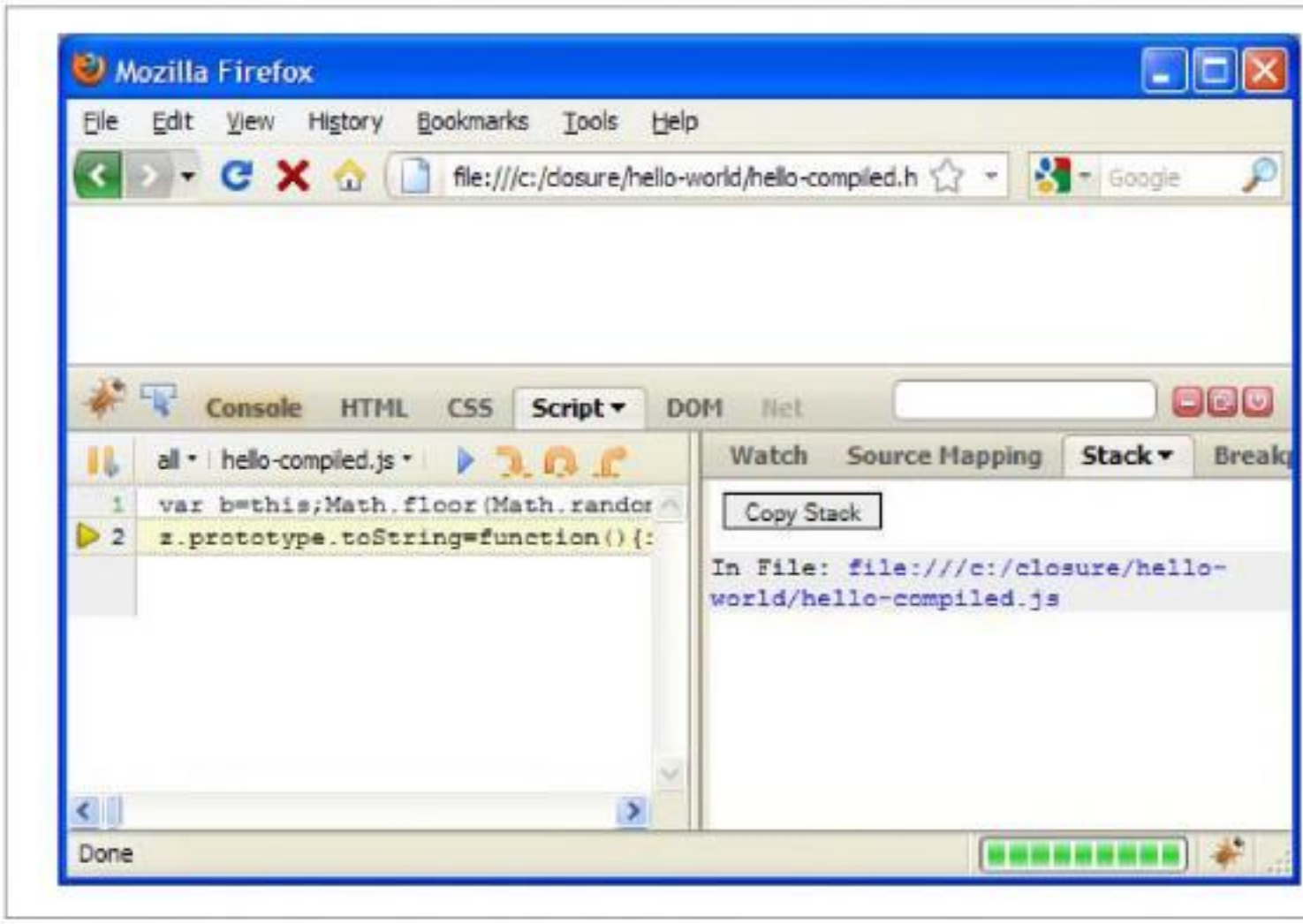


Figure 1-7. Closure Inspector hitting a breakpoint.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

allowed anywhere that text is allowed in the JSDoc comment. These two inline tags have the same meaning that they do in Javadoc, so they are not discussed in this chapter.

Getting back to the example, the doc comment for `goog.bind()` contains the two most commonly encountered JSDoc tags: `@param` and `@return`. Like their Javadoc equivalents, these tags are used to document parameters and return values, respectively. However, these JSDoc tags are more than just documentation when used with the Closure Compiler.

In Java, doc comments are purely documentation: the Java compiler ignores them completely when compiling code. The Java compiler gets its type information from method signatures that are required by the Java language. If a Java programmer wants to annotate Java code, he can use *Java annotations*, a language feature introduced in Java 1.5 for adding metadata to source code that is available to the Java compiler and other tools.

In JavaScript, the story is completely different. There is no language-level support for type information or annotations. In fact, JavaScript is an interpreted language, so there is no compiler to process such information were it to exist. But such information would be useful so that static checks about a program's correctness could be made.

This is where the Closure Compiler comes in. With it, JSDoc serves a dual purpose: it documents code for the developer and annotates code for the Compiler. Note that both the `@param` and `@return` tags in the `goog.bind()` example contain type information in curly braces. These curly braces delimit a *type expression*, which will be discussed in greater detail in the next section. The Closure Compiler uses type expressions to form the basis of its type system. By annotating variables with type information, the Compiler can perform compile-time checks to ensure that all functions will receive arguments of the specified type. This can help catch a large class of errors, as explained in "Type Checking" on page 408.

Type information can also be specified for variables via the `@type` annotation:

```
/** @type {number} */
example.highestRecordedTemperatureForTodayInFahrenheit = 33;
```

The Compiler also supports the idea of an enumerated type or enum. An *enum* is defined as an object literal whose properties represent the named values for the enum. Like enums in Java, the name of an enum in Closure should be a singular noun in camel case, but the names of the enum values should be in all caps. The type of the enum is specified using the `@enum` annotation, and each value of the enum must be of the type that corresponds to the annotation:

```
/** @enum {number} */
example.CardinalDirection = {
  NORTH: Math.PI / 2,
  SOUTH: 3 * Math.PI / 2,
  EAST: 0,
  WEST: Math.PI
};
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
example.convertStringToInteger = function(str) {
  var value = parseInt(str, 10);
  return isNaN(value) ? 0 : value;
};
```

By using a union type, `example.convertStringToInteger` declares that it will accept a `string`, `null`, or `undefined`, but it is guaranteed to return a number that is not `null`. Because it is common to specify whether or not `null` is accepted, type expressions have a shorthand for a union type that includes `null`, which is the type name prefixed with a `?` character:

```
/**
 * @param {?string} str1 is a string or null
 * @param {?string|undefined} str2 is a string, null, or undefined. The ?
 *     prefix does not include undefined, so it must be included explicitly.
 */
```

All types other than primitive types, such as `Object`, `Array`, and `HTMLDocument`, are nullable by default. These types are also called *object types* to distinguish them from primitive types. Therefore, the `?` prefix is redundant for object types:

```
/**
 * @param {Document} doc1 is a Document or null because object types are
 *     nullable by default.
 * @param {?Document} doc2 is also a Document or null.
 */
```

This necessitates a way to specify non-nullable object types. This is accomplished using the `!` prefix:

```
/**
 * @param {!Array} array must be a non-null Array
 * @param {!Array|undefined} maybeArray is an Array or undefined, but
 *     cannot be null.
 */
```

Likewise, because primitive types are non-null by default, using the `!` prefix with a primitive type is redundant. Therefore, `{!number}` and `{number}` both specify a non-null number.

Function Types

In JavaScript, functions are first-class objects that can be passed as parameters to other functions. As such, the Closure type system supports a rich syntax for describing function types. As expected, it is possible to specify a function with no qualifications as follows:

```
/** @param {Function} callback is some function */
```

This simple type expression specifies a function, but it does not specify how many parameters it takes or its return type. The type system does not enable such details to be specified when using `Function` with a capital `F`. More recently, the type system has been expanded to support richer type expressions for functions, which are denoted by

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
found   : {title: string}
required: { author : (string|undefined), title : (null|string),
    numRows : (number|undefined) }
example.createNewSpreadsheetWithRows({title: '2010 Taxes'});
                                      ^

1 error(s), 0 warning(s), 86.7% typed
```

Because it is not currently possible to specify an optional object property in the Closure type system, the workaround is to declare the parameter as a generic `Object`, and to document the properties informally:

```
/**
 * Creates a new spreadsheet.
 * @param {Object} properties supports the following options:
 *   author (string): email address of the spreadsheet creator
 *   title (string): title of the spreadsheet
 *   numRows (number): number of rows the spreadsheet should have
 * @notypecheck
 */
example.createNewSpreadsheetWithRows = function(properties) {
  var author = properties.author || 'bolinfest@gmail.com';
  var title = properties.title || 'New Spreadsheet';
  var numRows = properties.numRows || 1024;
  // Create a new spreadsheet using author, title, and numRows...
};

// This no longer results in a type checking error from the Compiler.
example.createNewSpreadsheetWithRows({title: '2010 Taxes'});
```

The `@notypecheck` annotation instructs the Compiler to ignore type checking on `example.createNewSpreadsheetWithRows()`. Without it, the Compiler would produce the following type checking errors:

```
spreadsheet-without-notypecheck.js:13: ERROR - Property
author never defined on Object
  var author = properties.author || 'bolinfest@gmail.com';
               ^

spreadsheet-without-notypecheck.js:15: ERROR - Property
numRows never defined on Object
  var numRows = properties.numRows || 1024;
                ^

2 error(s), 0 warning(s), 86.3% typed
```

Although using a function that takes a single object with many optional properties is more convenient to use than a function that declares many optional parameters, it does not work as well with the type checking logic of the Closure Compiler. It is possible that a new annotation will be introduced in the future to facilitate the use of this pattern.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

example.Date, it would issue an error because the function presumably relies on properties of example.DateTime that may not be true for objects of type example.Date. In practice, because example.DateTime is a subtype of example.Date, objects of type example.DateTime can be substituted for objects of type example.Date.

Note that this relationship is inverted when comparing function types whose arguments are subtypes of one another:

```
/** @typedef {function(example.Date)} */
example.DateFunction;

/** @typedef {function(example.DateTime)} */
example.DateTimeFunction;
```

In this case, example.DateFunction is a subtype of example.DateTimeFunction because all objects of type example.DateFunction exhibit all of the properties that are true for all objects of type example.DateTimeFunction. This is deeply counterintuitive and merits an example:

```
/** @type {example.DateFunction} */
example.writeDate = function(date) {
  document.write(date.year + '-' + date.month + '-' + date.date);
};

/** @type {example.DateTimeFunction} */
example.writeDateTime = function(dateTime) {
  document.write(dateTime.year + '-' + dateTime.month + '-' + dateTime.date +
      ' ' + dateTime.hour + ':' + dateTime.minute + ':' + dateTime.second);
};

/**
 * @param {example.DateFunction} f
 * @param {example.Date} date
 */
example.applyDateFunction = function(f, date) { f(date); };

/**
 * @param {example.DateTimeFunction} f
 * @param {example.DateTime} dateTime
 */
example.applyDateTimeFunction = function(f, dateTime) { f(dateTime); };

/** @type {example.Date} */
var date = {year: 2010, month: 12, date: 25};

/** @type {example.DateTime} */
var dateTime = {year: 2010, month: 12, date: 25, hour: 12, minute: 13, second: 14};
```

It is fairly straightforward that the following two calls will work without issue, as the objects match the specified parameter types exactly:

```
// Writes out: 2010-12-25
example.applyDateFunction(example.writeDate, date);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Deprecated Members

Like in Java, `@deprecated` can be used to identify functions and properties that should no longer be used. It is best practice to follow `@deprecated` with information on how to remove the dependency on the deprecated item. This will help developers transition away from deprecated members:

```
/** @deprecated Use example.setEnabled(true) instead. */
example.enable = function() { /*...*/ };

/** @deprecated Use example.setEnabled(false) instead. */
example.disable = function() { /*...*/ };
```

The Compiler can be configured to produce a warning when a deprecated member is used.

License and Copyright Information

As part of the Compiler's JavaScript minification process, it scrubs all comments from the input code; however, some comments are meant to be included with the compiled output. Frequently, source code contains legal licenses or copyright declarations that are meant to appear at the top of the compiled JavaScript file. Either `@license` or `@preserve` can be used to tag a doc comment so that it will appear before the compiled code for the marked file (line breaks will be preserved):

```
/**
 * @preserve Copyright 2010 SomeCompany.
 * The license information (such as Apache 2.0 or MIT) will also be included
 * here so that it is guaranteed to appear in the compiled output.
 */
```

In practice, many JavaScript source files that belong to the same project contain identical copyright and licensing information. Instead of using `@license` or `@preserve`, it is better to create a build process that prepends the appropriate comment to the beginning of the JavaScript generated by the Compiler so that it only appears once in the generated file rather than once per input file.

Is All of This Really Necessary?

Between the influence of the Javadoc tool and the heavy use of Java keywords as annotations, "They're trying to turn JavaScript into Java!" is a common outcry.

There are certainly concepts from Java (and other languages) that Closure tries to bring to JavaScript: type checking, information hiding, and inheritance, to name a few. Previous attempts to support these features in JavaScript have often led to increased code size and wasted memory. (See Appendix A for a detailed example.)

Rest assured that the heavy use of JSDoc in Closure is optional. It is still possible to use the Compiler with unannotated code; however, the Compiler can catch more errors

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goog.global

In practice, `goog.global` is primarily an alias for the `window` object of the frame in which the Closure Library is loaded. This is because in a web browser, `window` is an alias for the *global object*, which is a required, universally accessible object in any environment in which JavaScript is executed. As explained in section 15.1 of the ES5 specification, the global object is the object on which many global properties are defined: `NaN`, `Infinity`, `undefined`, `eval()`, `parseInt()`, etc.

The global object is mutable, so any values that are meant to be globally accessible should be defined as properties of `goog.global`. By comparison, properties that are specific to `window`, but not the browser-agnostic global object as defined in ES5, should be explicitly accessed via `window`:

```
// Define a global function for a JSONP callback.
goog.global['callback'] = function(json) { /* ... */ };

// setTimeout() is not defined by the ES5 specification, but by the W3C
// Window Object specification: http://www.w3.org/TR/Window/
window.setTimeout(someFunction, 100);

// Similarly, location is a property specific to a browser-based environment,
// and is defined explicitly for window by the W3C.
window.location = 'http://www.example.com/';
```

Finer details of goog.global

As explained previously, in web programming, the global object is `window`, though in `base.js`, `goog.global` is assigned to `this`. In a browser, both `this` and `window` refer to the same object when used in the global scope, so in that case, either could be used as the value of `goog.global`.

However, recall that JavaScript is a dialect of ECMAScript and that Closure Tools are designed to work for all ECMAScript code, not just JavaScript. Specifically, Closure Tools are designed to work with code that adheres to the third edition of the ECMA-Script Language Specification, which most closely matches the implementation of JavaScript in modern web browsers.

According to Section 10 of the third edition (*https://developer.mozilla.org/En/JavaScript_Language_Resources*), when `this` is referenced in the global code, it will return the global object, which again in the case of a web browser is `window`. However, there are environments besides web pages where ECMAScript can run, such as Firefox extensions and the Microsoft Windows Script Host. Such environments may not use `window` as a reference to the global object, or they may not provide any reference at all (this is the case in JScript). Assigning `goog.global` to `this` rather than `window` makes it easier to port Closure to other environments and provides an alias to the global object that can be renamed by the Compiler.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
example.radius.isWithinRadius = function(radius, point) {
  var origin = new goog.math.Coordinate(0, 0);
  var distance = goog.math.Coordinate.distance(point, origin);
  return distance <= radius;
};
```

Recall the "Hello World" example from Chapter 1 in which a web page with two `<script>` tags was created, one pointing to `base.js` and the other defining the `sayHello()` function. In that uncompiled usage of the Library, `goog.require()` did not throw an error even though `goog.provide('goog.dom')` had not been called yet—why did this happen?

Because `COMPILED` was set to its default value, `false`, `base.js` ran the following code when it was evaluated:

```
document.write('<script type="text/javascript" src="deps.js"></script>');
```

Note that `deps.js` lives in the same directory as `base.js` and that it contains many calls to `goog.addDependency()` (which is explained in the next section). These calls load Closure's dependency graph as created by `calcdeps.py`. When `COMPILED` is `false`, `goog.require()` looks at this dependency graph to determine all of `goog.dom`'s dependencies. For each dependency it finds, it adds another `<script>` tag pointing to the file that contains the corresponding call to `goog.provide()`.

Chapter 11 discusses generating the equivalent JavaScript file for a Closure Template. Each generated JavaScript file contains a `goog.provide()` call, so files that use such templates can `goog.require()` them like any other Closure Library file.

goog.addDependency(relativePath, provides, requires)

`goog.addDependency()` is used to construct and manage the dependency graph used by `goog.require()` in uncompiled JavaScript. When JavaScript is compiled with the Closure Compiler, `goog.provide()` and `goog.require()` are analyzed at compile time to ensure that no namespace is required before it is provided. If this check completes successfully, calls to `goog.provide()` are replaced with logic to construct the object on which the namespace will be built, and calls to `goog.require()` are removed entirely. In the compiled case, there is no need for the dependency graph to be constructed on the client, so `goog.addDependency()` and the global constants it depends on are defined in such a way that they will be stripped from the output when `COMPILED` is `true`.

By comparison, uncompiled Closure code relies on the information from `goog.addDependency()` to determine which additional JavaScript files to load. Consider the following example, in which the `example.View` namespace depends on the `example.Model` namespace:

```
// File: model.js
goog.provide('example.Model');
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Using `goog.partial()` can also help prevent memory leaks. Consider the following example:

```
var createDeferredAlertFunction = function() {
  // Note the XMLHttpRequest constructor is not available in Internet Explorer 6.
  var xhr = new XMLHttpRequest();

  return function() {
    alert('Hello world!');
  };
};

var deferredAlertFunction = createDeferredAlertFunction();
```

The `deferredAlertFunction()` that is created maintains references to all variables that were in scope in the enclosing function in which it was defined. This includes the `XMLHttpRequest` object that it did not (and will never) use. For those unfamiliar with declaring a function within a function, this may come as a surprising result. If you are unconvinced, try running the following code:

```
var createDeferredEval = function() {
  // Note the XMLHttpRequest constructor is not available in Internet Explorer 6.
  var xhr = new XMLHttpRequest();

  return function(handleResult, str) {
    // Note there are no references to the XMLHttpRequest within this function.
    var result = eval(str);
    handleResult(result);
  };
};

var deferredFunction = createDeferredEval();

// Alerts the toString() of XMLHttpRequest.
deferredFunction(alert, 'xhr');
```

Normally, after `createDeferredEval()` is finished executing, `xhr` would fall out of scope and get garbage collected because there are no more references to `xhr`. Yet even though the deferred function does not reference the `XMLHttpRequest`, calling it proves that it can still access it. This is because the deferred function maintains a reference to the scope in which it was defined. Therefore, as long as a reference to the deferred function exists, the reference to its scope will continue to exist, which means none of the objects in that scope can be garbage collected.

What may be even more surprising is that the following code does not fix the problem:

```
var createDeferredEval = function() {
  var xhr = undefined;

  var theDeferredFunction = function(handleResult, str) {
    // Note there are no references to the XMLHttpRequest within this function.
    var result = eval(str);
    handleResult(result);
  };
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

property will be available under that name, even after compilation. Consider the following uncompiled source code:

```
goog.provide('Lottery');

Lottery.doDrawing = function() {
  Lottery.winningNumber = Math.round(Math.random() * 1000);
};

// In uncompiled mode, this is redundant.
goog.exportProperty(Lottery, 'doDrawing', Lottery.doDrawing);
```

Now consider the compiled version of the code:

```
var a = {}; // Lottery namespace
a.a = function() { /* ... */ }; // doDrawing has been renamed to 'a'
a.doDrawing = a.a; // doDrawing exported on Lottery
```

In the compiled version of the code, the `Lottery` object has two properties defined on it (`a` and `doDrawing`), both of which refer to the same function. Exporting the `doDrawing` property does not replace the renamed version of the property. This is done deliberately so that code that was compiled with `Lottery` can use the abbreviated reference to the function, `a.a`, thereby reducing code size.

Note that exporting mutable properties is unlikely to have the desired effect. Consider exporting a mutable property named `winningNumber` that is defined on `Lottery`:

```
Lottery.winningNumber = 747;
goog.exportProperty(Lottery, 'winningNumber', Lottery.winningNumber);

Lottery.getWinningNumber = function() { return Lottery.winningNumber; };
goog.exportProperty(Lottery, 'getWinningNumber', Lottery.getWinningNumber);
```

When compiled together with the original code, this becomes:

```
var a = {};
a.b = function() { a.a = Math.round(Math.random() * 1000); };
a.doDrawing = a.b;
a.a = 747;
a.winningNumber = a.a;
a.c = function() { return a.a; };
a.getWinningNumber = a.c;
```

Now consider using the following code with the `Lottery` library (assume `Lottery` has been exported, as well):

```
var hijackLottery = function(myNumber) {
  Lottery.doDrawing();
  Lottery.winningNumber = myNumber;
  return Lottery.getWinningNumber();
};
```

When using the uncompiled version of `Lottery`, `hijackLottery()` will return the value of `myNumber`. But when using the uncompiled version of `hijackLottery()` with the compiled version of the `Lottery` library, a random number will be returned. This is

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
goog.isDef(undefined); // false
goog.isDef(0);         // true
goog.isDef(false);     // true
goog.isDef(null);      // true
goog.isDef('');        // true
```

`goog.isDef()` is most frequently used to test whether an optional argument was specified:

```
/**
 * @param {number} bill The cost of the bill.
 * @param {number=} tipAmount How much to tip. Defaults to 15% of the bill.
 */
var payServer = function(bill, tipAmount) {
  if (goog.isDef(tipAmount)) {
    pay(bill + tipAmount);
  } else {
    pay(bill * 1.15);
  }
};
```

goog.isNull(value)

`goog.isNull(value)` returns `true` if `value === null`, so it will return `false` for `undefined` and all values that are not strictly equals to `null`.

goog.isDefAndNotNull(value)

`goog.isDefAndNotNull(value)` returns `true` if `value` is neither `null` nor `undefined`.

goog.isArray(obj)

It is surprisingly difficult in JavaScript to decisively determine whether an object is an array. Many would expect the following test to be sufficient:

```
var isArray = function(arr) {
  return arr instanceof Array;
};
```

The problem is that `Array` refers to a function defined in the `window` in which the JavaScript is executed. On web pages that create objects in different frames on the same domain, each frame has its own `Array` function. If array objects are passed between frames, an array created in one frame will not be considered an `instanceof Array` in another frame because its constructor function does not match the one used with the `instanceof` operator. Because of this, `instanceof` is only the first test of many that `goog.isArray()` uses to detect whether an object is an array. Because of this complexity, always use `goog.isArray()` rather than a home-grown solution.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

library that has had access to `obj` may have stored its UID. A common case where a UID may need to be removed is when it is being serialized as JSON in order to avoid the private property added by `goog.getUid()` from appearing in the output.

Internationalization (i18n)

At the time of this writing, not all of the code for doing internationalization (i18n) in Closure has been open-sourced, which is why existing solutions for inserting translated messages into Closure Library code are somewhat awkward. An alternative to declaring messages with `goog.getMsg()` in Closure Library code is to store each message as a Closure Template (where i18n is fully supported), and to use the appropriate template from Closure Library code when a localized message is needed.

In the Closure Library, the intended technique for doing i18n is for the Compiler to produce one JavaScript file per locale with the translations for the respective locale baked into the file. This differs from other schemes, in which there is one file per locale that contains only translated messages assigned to variables and one JavaScript file with the bulk of the application logic that uses the message variables. Implementing the "two file" approach is also an option for doing i18n in the Closure Library today: the file with the translated messages should redefine `goog.getMsg()` as follows:

```
// File with application logic:
var MSG_HELLO_WORLD = goog.getMsg('hello world');

// File with translations:
goog.getMsg = function(str, values) {
  switch (str) {
    case 'hello world': return 'hola mundo';
    default: throw Error('No translation for: ' + str);
  }
};
```

The Compiler is not designed for the "two file" scheme because the "single file" scheme results in less JavaScript for the user to download. The drawback is that locales cannot be changed at runtime in the application (note how Google applications, such as Gmail, reload themselves if the user changes her locale setting), but this is fairly infrequent and is considered a reasonable trade-off.

goog.LOCALE

`goog.LOCALE` identifies the locale for a compiled JavaScript file and defaults to `"en"`. Its value should be in the canonical Unicode format using a hyphen as a delimiter (e.g., `"fr"`, `"pt-BR"`, `"zh-Hans-CN"`). By treating `goog.LOCALE` as a constant, the Compiler can eliminate a lot of dead code by stripping out messages that are used with locales other than the one identified by `goog.LOCALE`. `goog.i18n.DateTimeSymbols` would be too large to be usable if this were not the case.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

It may be tempting to use `COMPILED` instead of `goog.DEBUG` to identify blocks of code that can be removed during compilation; however, this is strongly discouraged. As Chapter 13 will explain, whether code is compiled and whether it contains debug information are orthogonal concepts and therefore they should be able to be toggled independently during development.

goog.now()

Returns an integer value representing the number of milliseconds between midnight, January 1, 1970, and the current time. It is basically an alias for `Date.now()`, but it has the advantage that it can be mocked out in a unit test.

goog.globalEval(script)

Takes a string of JavaScript and evaluates it in the global context. JavaScript that is delay-loaded as a string (rather than sourced via a script tag) should be evaluated using this function so variables that are currently in local scope will be protected from the JavaScript being evaluated.

goog.getCssName(className, opt_modifier), goog.setCssNameMapping(mapping)

These functions can be used with the Compiler to rename CSS classes referenced in JavaScript code. Unfortunately, at the time of this writing, Closure does not provide a tool to rename the classes in the stylesheets in which they are declared. However, members of the Closure Compiler discussion group have suggested tools for renaming CSS classes in stylesheets that can be used with the Compiler's CSS renaming logic: see the website *http://groups.google.com/group/closure-compiler-discuss/browse_thread/thread/1eba1d4f9f4f6475/aff6de7330df798a*.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goog.string

`goog.string` contains many functions for dealing with strings. As explained in Chapter 1, Closure does not modify function prototypes, such as `String`, so functions that would be methods of `String` in other programming languages are defined on `goog.string` rather than on `String.prototype`. For example, `goog.string.startsWith()` is a function that takes a string and prefix and returns `true` if the string starts with the specified prefix. It could be added as a method of `String` by modifying its prototype:

```
// THIS IS NOT RECOMMENDED
String.prototype.startsWith = function(prefix) {
  return goog.string.startsWith(this, prefix);
};
```

This would make it possible to rewrite `goog.string('foobar', 'foo')` as `'foobar'.startsWith('foo')`, which would be more familiar to Java programmers. However, if Closure were loaded in a top-level frame and operated on strings created in child frames, those strings would have a `String.prototype` different from the one modified by Closure in the top-level frame. That means that the strings created in the top-level frame would have a `startsWith()` method, and those created in child frames would not. This would be a nightmare for client code to deal with; using pure functions rather than modifying the prototype avoids this issue.

goog.string.htmlEscape(str, opt_isLikelyToContainHtmlChars)

Offering a function to escape a string of HTML is fairly common in a JavaScript library, yet Closure's implementation is worth highlighting. Most importantly, escaping strings of HTML can prevent security vulnerabilities and save your website considerable bad press. Note that the solution is not to liberally sprinkle `goog.string.htmlEscape()` throughout your code—that may prevent security vulnerabilities, but it may also result in displaying doubly escaped text, which is also embarrassing. The first argument to `goog.string.htmlEscape()` should be a string of text, not a string of HTML. Unfortunately, there is not yet a type expression to use to discriminate between text and HTML: both arguments are simply annotated as string types. The description of a string parameter should always specify whether its content is text or HTML if it is ambiguous.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Like most comparison functions in JavaScript, if `version1` is less than `version2`, then `goog.string.compareVersions` returns -1, if they are the same it returns 0, and if `version1` is greater than `version2` it returns 1. In Closure, this is most commonly used when comparing the version numbers for user agents. Some examples:

```
// goog.string.compareVersions takes numbers in addition to strings.
goog.string.compareVersions(522, 523); // evaluates to -1

// Here the extra zero is not significant.
goog.string.compareVersions('3.0', '3.00'); // evaluates to 0

// Because letters often indicate beta versions that are released before the
// final release, 3.6b1 is considered "less than" 3.6.
goog.string.compareVersions('3.6', '3.6b1'); // evaluates to 1
```

goog.string.hashCode(str)

The `goog.string.hashCode(str)` function behaves like `hashCode()` in Java. The hash code is computed as a function of the content of `str`, so strings that are equivalent by `==` will have the same hash code.

Because string is an immutable type in JavaScript, a string's hash code will never change. However, its value is not cached, so `goog.string.hashCode(str)` will always recompute the hash code of `str`. Because `goog.string.hashCode` is O(n) in the length of the string, using it could become costly if the hash codes of long strings are frequently recomputed.

goog.array

Like `goog.string`, `goog.array` defines a number of functions for dealing with arrays rather than modifying `Array.prototype`. This is even more important in the case of `goog.array` because many functions in `goog.array` do not operate on objects of type `Array`: they operate on objects of the ad-hoc type `goog.array.ArrayLike`. Because `goog.array.ArrayLike` is an ad-hoc type, it has no corresponding prototype to which array methods could be added. This design makes it possible to use common array methods such as `indexOf()` and `filter()` on Array-like objects such as `NodeList` and `Arguments`.

Note that some functions in `goog.array` take only an `Array` as an argument rather than `goog.array.ArrayLike`. This is because not all `ArrayLike` types are mutable, so functions such as `sort()`, `extend()`, and `binaryInsert()` restrict themselves to operating on `Arrays`. However, such functions can be applied to a copy of an `ArrayLike` object:

```
// images is a NodeList, so it is ArrayLike.
var images = document.getElementsByTagName('IMG');

// goog.array.toArray() takes an ArrayLike and returns a new Array.
var imagesArray = goog.array.toArray(images);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goog.setIfUndefined(obj, key, value)

`goog.setIfUndefined()` creates a property on `obj` using `key` and `value` if `key` is not already a key for `obj`. Like `goog.object.get()`, `goog.setIfUndefined()` also has curious behavior if `obj` has already has `key` as a property, but whose value is `undefined`:

```
var chessboard = { 'a1': 'white_knight', 'a2': 'white_pawn', 'a3': undefined };

// Try to move the pawn from a2 to a3.
goog.object.setIfUndefined(chessboard, 'a3', 'white_pawn');
if (chessboard['a3'] != 'white_pawn') {
  throw Error('Did not move pawn to a3');
}
```

In the previous example, an error is thrown because `goog.object.setIfUndefined` does not modify `chessboard` because it already has a property named `a3`. To use `goog.object.setIfUndefined` with this abstraction, properties must be assigned only for occupied squares:

```
// Do not add a key for 'a3' because it does not have a piece on it.
var chessboard = { 'a1': 'white_knight', 'a2': 'white_pawn' };

// Now this will set 'a3' to 'white_pawn'.
goog.object.setIfUndefined(chessboard, 'a3', 'white_pawn');

// This will free up 'a2' so other pieces can move there.
delete chessboard['a2'];
```

goog.object.transpose(obj)

`goog.object.transpose()` returns a new object with the mapping from keys to values on `obj` inverted. In the simplest case where the values of `obj` are unique strings, the behavior is straightforward:

```
var englishToSpanish = { 'door': 'puerta', 'house': 'casa', 'car': 'coche' };
var spanishToEnglish = goog.object.transpose(englishToSpanish);
// spanishToEnglish is { 'puerta': 'door', 'case': 'house', 'coche': 'car' }
```

If `obj` has duplicate values, then the result of `goog.object.transpose` can vary depending on the environment. For example, because most browsers will iterate the properties of an object literal in the order in which they are defined and because `goog.object.transpose` assigns mappings using iteration order, most browsers would produce the following:

```
var englishToSpanish1 = { 'door': 'puerta', 'goal': 'puerta' };
var spanishToEnglish1 = goog.object.transpose(englishToSpanish1);
// spanishToEnglish1 is { 'puerta': 'goal' }

var englishToSpanish2 = { 'goal': 'puerta', 'door': 'puerta' };
var spanishToEnglish2 = goog.object.transpose(englishToSpanish2);
// spanishToEnglish2 is { 'puerta': 'door' }
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
// Aliases for document.getElementById() in popular JavaScript libraries:
var el = goog.dom.getElement('header');     // Closure
var el = goog.dom.$('header');              // alias for goog.dom.getElement()
var el = dojo.byId('header');               // Dojo
var el = $('header');                       // Prototype, MooTools, and jQuery
```

As usual, Closure does not win first place for having the shortest function name, but after compilation, it will not make a difference. Like the other libraries listed previously, Closure's `goog.dom.getElement` function accepts either a string id or an `Element` object. In the case of the former, it looks up the element by id and returns it; in the case of the latter, it simply returns the `Element`.

`goog.dom.getElement()` and `goog.dom.$()` are references to the same function. Because the Closure Library was designed with the Compiler in mind, it does not engage in the practice of other JavaScript libraries that try to save bytes by using abbreviated function names. (Using more descriptive names is more to type, but also makes the code more readable.) Nevertheless, the use of `$` as an alias for `document.getElementById` is so prevalent in other libraries that `goog.dom.$()` is supported because it is familiar to JavaScript developers. However, if you choose to take this shortcut, be aware that `goog.dom.$()` is marked deprecated, so using it will yield a warning from the Compiler if you have deprecation warnings turned on.

goog.dom.getElementsByTagNameAndClass(nodeName, className, elementToLookIn)

It is common to add a CSS class to an element as a label so that it can be used as an identifier when calling `goog.dom.getElementsByTagNameAndClass`. When such a class has no style information associated with it, it is often referred to as a *marker class*. Although the `id` attribute is the principal identifier for an element, ids are meant to be distinct, so it is not possible to label a group of elements with the same id. In this way, CSS classes can act as identifiers in addition to being directives for applying CSS styles.

Each argument to `goog.dom.getElementsByTagNameAndClass()` is optional. When no arguments are specified, all elements in the DOM will be returned. If only the first argument is specified, `goog.dom.getElementsByTagNameAndClass()` will behave like the built-in `document.getElementsByTagName()`, which returns all elements in the document with the specified node name. The `nodeName` argument is case-insensitive.

```
// To select all elements, supply no arguments or use '*' as the first argument.
var allElements = goog.dom.getElementsByTagNameAndClass('*');

// allDivElements is an Array-like object that contains every DIV in the document.
var allDivElements = goog.dom.getElementsByTagNameAndClass('div');

// The previous statement is equivalent to:
var allDivElements = goog.dom.getElementsByTagNameAndClass('DIV');
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In this example, `row` is used as a marker class to facilitate the use of `goog.dom.getAncestorByTagNameAndClass()`.

goog.dom.createDom(nodeName, attributes, var_args)

`goog.dom.createDom` creates a new `Element` with the specified node name, attributes, and child nodes. Using Closure Templates to build up a string of HTML and assigning it as the `innerHTML` of an existing `Element` is generally the most efficient way to build up a DOM subtree, but for projects that are not using Templates, this is the next best option. (Note that to maintain the Library's independence from Templates, the Library uses `goog.dom.createDom` heavily.) When building up small subtrees, the performance difference should be negligible.

The `nodeName` identifies the type of element to create:

```
// Creates a <span> element with no children
var span = goog.dom.createDom('span');
```

The second argument, `attributes`, is optional. If `attributes` is an object, then the properties of the object will be used as the key-value pairs for the attributes of the created element. If `attributes` is a string, `goog.dom.createDom` uses the string as the CSS class to add to the element and adds no other attributes:

```
// Example of creating a new element with multiple attributes.
// This creates an element equivalent to the following HTML:
// <span id="section-1" class="section-heading first-heading"></span>
var span = goog.dom.createDom('span', {
    'id': 'section-1'
    'class': 'section-heading first-heading',
    });

// Example of creating a new element with only the CSS class specified.
// Creates an element equivalent to the following HTML:
// <span class="section-heading first-heading"></span>
var span = goog.dom.createDom('span', 'section-heading first-heading');
```

Although `"class"` is used as the key in the attributes object in the example, `"className"` will also work. The keys in the attributes object can be either attribute names or the scriptable property names for Elements that correspond to such attributes, such as `"cssText"`, `"className"`, and `"htmlFor"`. See the implementation of `goog.dom.setProperties` for more details.

If specified, the remaining arguments to `goog.dom.createDom` represent the child nodes of the new element being created. Each child argument must be either a `Node`, a string (which will be interpreted as a text node), a `NodeList`, or an array that contains only `Node`s and strings. These child nodes will be added to the element in the order in which they are provided to `goog.dom.createDom`. Because `goog.dom.createDom` returns an `Element`, which is a type of `Node`, calls can be built up to create larger DOM structures:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goog.dom.classes

`goog.dom.classes` is a small collection of functions to help with manipulating an element's CSS classes. Because an element's `className` property returns the CSS classes as a space-delimited string, working with individual classes often requires splitting the string into individual names, manipulating them, and then putting them back together. The functions in `goog.dom.classes` abstract away the string manipulation and are implemented so that they touch the `className` property as little as possible. Accessing DOM properties is often more expensive than accessing properties of pure JavaScript objects, so DOM access is minimized in the Closure Library.

More importantly, it turns out that not all elements have a `className` property that is a string. `IFRAME` elements in Internet Explorer do not have a `className` property, and the `className` property of an `SVG` element in Firefox returns a function rather than a string. Using `goog.dom.classes` abstracts away these subtle cross-browser differences.

Much of the functionality in `goog.dom.classes` will also be available via the `classList` property proposed for HTML5. When available, the functions in `goog.dom.classes` will likely be reimplemented to take advantage of `classList`, but the contracts of the functions should remain unchanged.

goog.dom.classes.get(element)

`goog.dom.classes.get(element)` returns an array of class names on the specified element. If the element does not have any class names, a new empty array will be returned. It is this function that is largely responsible for hiding the cross-browser differences discussed in the overview for `goog.dom.classes`. Other utilities for working with CSS classes should be based on this function to benefit from the abstraction it provides.

```
// element is <span class="snap crackle pop"></span>

// evaluates to ['snap', 'crackle', 'pop']
goog.dom.classes.get(element);
```

Note that in HTML5, `classList` does not have an equivalent to `goog.dom.classes.get`, but it is not necessary because `classList` itself is an Array-like object. It is not mutable, like a true array, but it has a `length` property and its values are numerically indexed.

goog.dom.classes.has(element, className)

`goog.dom.classes.has` returns `true` if `element` has the specified `className`; otherwise, it returns `false`.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Rendering Engine Constants

Table 4-1 lists the values of `goog.userAgent` that can be tested to determine the user agent of the runtime environment. For each value, it also lists the value of `goog.userAgent` that can be set to `true` at compile time to predetermine the values of the other `goog.userAgent` rendering engine constants.

Table 4-1. Rendering engine constants in goog.userAgent.

Value of goog.userAgent	Compile-time constant	Description
`IE`	`ASSUME_IE`	`true` if the JavaScript is running in a browser that uses Microsoft's Trident rendering engine. Because Trident is embeddable in any Windows application, it is also embedded in many desktop applications on Windows, such as Internet Explorer and Google Desktop.
`GECKO`	`ASSUME_GECKO`	`true` if the JavaScript is running in a browser that uses Mozilla's Gecko rendering engine. In addition to Firefox, Gecko is also used to power Fennec and Camino.
`WEBKIT`	`ASSUME_WEBKIT`	`true` if the JavaScript is running in a browser that uses the WebKit rendering engine. This includes Safari and Google Chrome. This will also be set to `true` at compile time if `--define goog.userAgent.ASSUME_MOBILE_WEBKIT` is set.
`OPERA`	`ASSUME_OPERA`	`true` if the JavaScript is running on any Opera-based browser. This includes the browser that can be downloaded for the Nintendo Wii.
`MOBILE`	`ASSUME_MOBILE_WEBKIT`	`true` if the JavaScript is running in WebKit on a mobile device. Closure uses the existence of `"Mobile"` in the user agent string to make this determination. It is likely that this excludes a number of JavaScript-enabled mobile web browsers (such as the Palm Pre), but this heuristic will work for iPhones and Android-based devices. This will be set to `true` at compile time if `--define goog.userAgent.ASSUME_MOBILE_WEBKIT` is set, though the flag should be set only if the target rendering engine is indeed WebKit and not for other mobile browsers such as Opera Mini, as this has the side effect of also setting `goog.userAgent.WEBKIT` to `true`.

In general, it is preferable to use feature detection rather than browser detection to determine which browser-specific behavior should be used. In web development, *feature detection* is the practice of using JavaScript to determine a browser's capability at runtime and responding with the appropriate behavior. The alternative is known as *browser detection*, whereby different behaviors are hardcoded for different browsers. Generally speaking, feature detection is preferable because the features that a browser

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

That is not to say that `goog.userAgent.product` is useless—far from it. It makes it possible to differentiate iPhone devices from Android devices, which is important because each may have some custom APIs that the other does not. Although the Closure Library aims to provide a uniform API that works on all modern browsers, your own web applications may be tailored to work on a select handful of platforms for which testing for `goog.userAgent.product` is important. Table 4-3 lists the values in `goog.userAgent.product` that can be tested.

Table 4-3. Product constants in goog.userAgent.product.

Value of goog.userAgent.product	Compile-time constant	Description
`ANDROID`	`goog.userAgent.product.ASSUME_ANDROID`	`true` if the JavaScript is running on the built-in browser on an Android phone. If `ASSUME_ANDROID` is used, then `--define goog.userAgent.ASSUME_MOBILE_WEBKIT` should be used as well.
`IPHONE`	`goog.userAgent.product.ASSUME_IPHONE`	`true` if the JavaScript is running on an iPhone or an iPod touch. If `ASSUME_IPHONE` is used, then `--define goog.userAgent.ASSUME_MOBILE_WEBKIT` should be used as well.
`IPAD`	`goog.userAgent.product.ASSUME_IPAD`	`true` if the JavaScript is running on an iPad. If `ASSUME_IPAD` is used, then `--define goog.userAgent.ASSUME_MOBILE_WEBKIT` should be used as well.
`FIREFOX`	`goog.userAgent.product.ASSUME_FIREFOX`	`true` if the JavaScript is running on the Firefox web browser. Even though `goog.userAgent.GECKO` may be true, this could be Firefox or another Gecko-based browser such as Fennec or Camino. If `ASSUME_FIREFOX` is used, then `--define goog.userAgent.ASSUME_GECKO` should be used as well.
`CAMINO`	`goog.userAgent.product.ASSUME_CAMINO`	`true` if the JavaScript is running on the Camino web browser. Camino is a Gecko-based browser, like Firefox, but its UI is built using native Mac widgets in an attempt to make it more lightweight than Firefox. But because it uses a different UI toolkit than Firefox, far fewer browser extensions are written for Camino than Firefox. Because browsers on the Mac have improved so much in recent years, the advantages of using Camino over Firefox or Safari are minimal, and Camino's marketshare has dwindled. It is unlikely that Camino-specific logic will be necessary (though it has been known to happen). If `ASSUME_CAMINO` is used, then `--define goog.userAgent.ASSUME_GECKO` should be used as well.
`SAFARI`	`goog.userAgent.product.ASSUME_SAFARI`	`true` if the JavaScript is running on the Safari web browser. If `ASSUME_SAFARI` is used, then `--define goog.userAgent.ASSUME_WEBKIT` should be used as well.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

dependencies.) Assuming `goog-inline-block` is available, another way to implement this function would be:

```
goog.dom.classes.add(element, 'goog-inline-block');
```

Because there is no complementary `goog.style.removeInlineBlock` function, the previous implementation has the advantage that the `inline-block` style can be removed with one line of code:

```
goog.dom.classes.remove(element, 'goog-inline-block');
```

goog.style.setUnselectable(element, unselectable, opt_noRecurse)

`goog.style.setUnselectable` sets whether the text selection is enabled in `element` and all of its descendants (unless `opt_noRecurse` is `true`). Some rendering engines allow text selection to be controlled via CSS; others support a separate `unselectable` attribute (whose default value is `off`). Because of the cascading nature of CSS, rendering engines that support this setting via CSS need only to set the appropriate style on `element` and the effect will cascade to all of its descendants (assuming none if its descendants override the value of the style being set). Because DOM attributes are applied to only an element and not its descendants, rendering engines that support the `unselectable` attribute must explicitly set it to `on` for `element` and all of its descendants to disable text selection on an entire subtree.

There is a complementary `goog.style.isUnselectable(element)` function that determines whether the specified element is unselectable.

goog.style.installStyles(stylesString, opt_node)

`goog.style.installStyles` takes a string of style information and installs it into the window that contains `opt_node` (which defaults to `window` if `opt_node` is not specified). It returns either an `Element` or `StyleSheet` that can be used with `goog.style.setStyles` or `goog.style.uninstallStyles` to update or remove the styles, respectively:

```
var robotsTheme = 'body { background-color: gray; color: black }';
var sunsetTheme = 'body { background-color: orange; color: red }';

// Adds the "robot" theme to the page.
var stylesEl = goog.style.installStyles(robotsTheme);

// Replaces the "robot" theme with the "sunset" theme.
goog.style.setStyles(stylesEl, sunsetTheme);

// Removes the "sunset" theme.
goog.style.uninstallStyles(stylesEl);
```

This will change the styles in the page without requiring the page to be reloaded. Gmail uses this to let its user switch between themes.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CHAPTER 5

Classes and Inheritance

None of the modules in the previous section were written in an object-oriented programming (OOP) style, but many of the modules in Closure are. Although JavaScript provides support for prototype-based programming (as opposed to class-based programming, which Java uses), the Closure Library approximates class-based programming using prototypes. In *JavaScript: The Good Parts* (O'Reilly), Douglas Crockford refers to this as the *pseudoclassical* pattern. This pseudoclassical pattern is chosen over the traditional cloning technique for creating objects used in prototype-based programming in order to conserve memory. Its style should be familiar to Java programmers. For these reasons, types are frequently referred to as classes in Closure, even though classes are not formally supported in JavaScript.

Many JavaScript developers balk at the idea of using OOP in JavaScript, arguing that its true beauty lies in its use as a weakly-typed language with first-class functions. Others say outright that OOP is fundamentally flawed, claiming that even Java guru Joshua Bloch thinks OOP is a bad idea, citing Item 14 of his preeminent book *Effective Java* (Addison-Wesley): "Favor composition over inheritance." Those detractors fail to mention Bloch's subsequent point, Item 15: "Design and document for inheritance or else prohibit it." The two principal uses of inheritance in the Closure Library (`goog.Disposable`, explained in this chapter, and `goog.events.EventTarget` in the following chapter) are particularly well designed and would be considerably awkward to use if they were implemented using composition.

Regardless of which side of debate you fall on, it is important to understand how classes are used in the Closure Library. This chapter will show how classes are represented in Closure, which should aid in creating new classes in the style of the Library and with understanding existing Closure source code.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Each method contains at least one reference to `this`, which will refer to the instance of the class when it is invoked. For this reason, these can also be referred to as *instance methods*. If a method does not contain any references to `this`, then it could be rewritten as a *static* method, which is explained later in this section. (Instance methods have the advantage that they can be overridden by subclasses or mocked out on a per-instance basis.) To understand how `this` works when invoking a method, consider the following code snippet:

```
var whiteHouse = new example.House('1600 Pennsylvania Avenue', 35);
whiteHouse.setNeedsPaintJob(true);
```

in which the second line is equivalent to the following:

```
whiteHouse.setNeedsPaintJob.call(whiteHouse, true);
```

The object `whiteHouse` does not have its own property named `setNeedsPaintJob`, but the first object in its prototype chain, `example.House.prototype`, does. The `setNeedsPaintJob` property on `example.House.prototype` is assigned to the following function:

```
function(needsPaintJob) {
  this.needsPaintJob_ = needsPaintJob;
}
```

This function has no idea that it is assigned to the prototype of some object. All it knows is that it takes one argument and sets it as the value of the `needsPaintJob_` property on whatever object `this` refers to. Recall that functions are first-class objects in JavaScript and that they have their own methods, such as `call()` and `apply()`. The use of `call()` in the previous example says to execute the function with one argument whose value is `true` and to bind `this` to `whiteHouse` during the execution.

Although there is some trickery with prototypes going on behind the scenes, the net effect is code that looks like traditional method invocation in class-based programming languages such as Java.

Equivalent Example in Java

By comparison, this is how the equivalent class would be defined in Java. Because Java does not support optional arguments, `House` has three constructor functions, but they delegate to one another so the constructor logic is not duplicated. Unlike JavaScript, every instance of `House` in Java will store its own values for `numberOfBathrooms` and `needsPaintJob`, even if the instance uses the default values for those fields:

```
package example;

/**
 * House contains information about a house, such as its address.
 * @author bolinfest@gmail.com (Michael Bolin)
 */
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example of a Subclass in Closure

Inheritance is also supported in Closure via subclassing. Members that are intended to be visible only to subclasses should be marked with the `@protected` annotation. By convention, private members in the Closure Library are named with a trailing underscore, but protected members are not.

The `@protected` annotation does not imply a member should also be visible to classes in the same package, as it does in Java, because JavaScript does not have packages. A better analogue is the notion of `protected` in C++.

Closure JavaScript Example

As an extension to the `House` example, consider its subclass, `Mansion`:

```
/**
 * @fileoverview A Mansion is a larger House that includes a guest house.
 * @author bolinfest@gmail.com (Michael Bolin)
 */
goog.provide('example.Mansion');

goog.require('example.House');

/**
 * @param {string} address Address of this mansion.
 * @param {example.House} guestHouse This mansion's guest house.
 * @param {number=} numberOfBathrooms Number of bathrooms in this mansion.
 *     Defaults to 10.
 * @constructor
 * @extends {example.House}
 */
example.Mansion = function(address, guestHouse, numberOfBathrooms) {
  if (!goog.isDef(numberOfBathrooms)) {
    numberOfBathrooms = 10;
  }
  example.House.call(this, address, numberOfBathrooms);

  /**
   * @type {example.House}
   * @private
   */
  this.guestHouse_ = guestHouse;
};
goog.inherits(example.Mansion, example.House);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

`goog.require()` helps keep code size down. The way `giveItemsToGoodwill()` was originally written, the Compiler does not have to bring in `goodwill.js` when it compiles `mansion.js`. When written the second way, the Compiler might be able to identify that `Goodwill` is unused (assuming `giveItemsToGoodwill()` is never called), even though it is included, but that analysis is not as straightforward and may not prove conclusive depending on the other code being compiled.

Also, having superfluous dependencies may inadvertently introduce circular dependencies from the perspective of `calcdeps.py`. Recall that circular dependencies prevent `calcdeps.py` from building a dependency graph or ordering your JavaScript files for compilation.

Equivalent Example in Java

Following is the equivalent subclass written in Java:

```
public class Mansion extends House {

  private House guestHouse;

  public Mansion(String address, House guestHouse) {
    this(address, guestHouse, 10);
  }

  public Mansion(String address, House guestHouse, int numberOfBathrooms) {
    super(address, numberOfBathrooms);
    this.guestHouse = guestHouse;
  }

  public void giveItemsToGoodwill(Goodwill goodwill) {
    for (Object item : itemsInTheGarage) {
      goodwill.acceptDonation(item);
    }
    itemsInTheGarage = new Object[0];
  }

  @Override
  public void paint(String color) {
    super.paint(color);
    guestHouse.paint(color);
  }
}
```

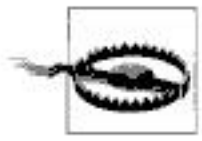
The only notable difference is that in Java, the super call to the superclass constructor must be the first line of the constructor, whereas in JavaScript it can appear anywhere in the constructor (or not at all, though that is not recommended). It is generally the first line of the constructor in JavaScript as well, but strange interactions between the constructor and overridden methods may require it to run after other fields are initialized. Although it does not appear as the first statement in the JavaScript `Mansion` example, it easily could have been:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the Compiler, properties should not be quoted, so this would seem to violate that principle. Fortunately, the Closure Compiler has special logic for processing `goog.base()`, so referring to a property as a string does not present a problem, in this case.



The one thing that `goog.base()` cannot do is call an instance method of the superclass from a constructor, as the heuristic for `goog.base()` will misinterpret that as a call to the superclass's constructor function.

At the time of this writing, `goog.base()` is a fairly new addition to the Closure Library, so much of the existing codebase has not had a chance to adopt it yet. There is no reason to hold off on using `goog.base()` in your own code, as it is easier to read and maintain than its verbose predecessor.

Abstract Methods

To define a method that a subclass is intended to override, give it the value `goog.abstractMethod` as follows:

```
goog.provide('example.SomeClass');

/** @constructor */
example.SomeClass = function() {};

/**
 * The JSDoc comment should explain the expected behavior of this method so
 * that subclasses know how to implement it appropriately.
 */
example.SomeClass.prototype.methodToOverride = goog.abstractMethod;
```

When implementing a subclass of a method that has an abstract method, failure to override the method will result in a runtime error when it is invoked:

```
goog.provide('example.SubClass');

/**
 * @constructor
 * @extends {example.SomeClass}
 */
example.SubClass = function() {
  goog.base(this);
};
goog.inherits(example.SubClass, example.SomeClass);

var subClass = new example.SubClass();

// There is no error from the Compiler saying this is an abstract/unimplemented
// method, but executing this code will yield a runtime error thrown by
// goog.abstractMethod.
subClass.methodToOverride();
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
goog.inherits(example.AndroidPhone, example.Phone);
goog.mixin(example.AndroidPhone.prototype, example.Mp3Player.prototype);
```

Now if a new `AndroidPhone` were created, it would have both `makePhoneCall()` and `playSong()` as methods. But consider what happens if `AndroidPhone` tries to override `playSong()`:

```
/** @inheritDoc */
example.AndroidPhone.prototype.playSong = function(fileName) {
  var mp3 = example.AndroidPhone.superClass_.playSong.call(this, fileName);
  this.displaySongName(mp3.getTitle());
};
```

Because `example.AndroidPhone.superClass_` points to `example.Phone.prototype`, `example.AndroidPhone.superClass_.playSong` will be `undefined` and a null pointer error will be thrown if `AndroidPhone`'s `playSong()` method is invoked. The following admittedly works, but requires the developer to keep track of which methods come from the true superclass and which come from the multiple inherited superclasses:

```
/** @inheritDoc */
example.AndroidPhone.prototype.playSong = function(fileName) {
  var mp3 = example.Mp3Player.prototype.playSong.call(this, fileName);
  this.displaySongName(mp3.getTitle());
};
```

Doing multiple inheritance using `goog.mixin()` also breaks down when it comes to the `instanceof` operator:

```
// This evaluates to false
((new example.AndroidPhone('8675309', '10GB')) instanceof example.Mp3Player)
```

A cleaner alternative would be to encapsulate the logic in a function that can be reused by both `Mp3Player` and `AndroidPhone`:

```
goog.provide('example.mp3');

/**
 * @this {example.Mp3Player|example.AndroidPhone}
 */
example.mp3.playSongFromFile = function(fileName) {
  // This implies that both Mp3Player and AndroidPhone have methods named
  // getPlayer() that return an object with a playFile() method.
  this.getPlayer().playFile(fileName);
};

example.Mp3Player.prototype.playSong = example.mp3.playSongFromFile;

example.AndroidPhone.prototype.playSong = example.mp3.playSongFromFile;
```

Using this technique, the logic is in one place, but it can be reused by both `Mp3Player` and `AndroidPhone` without having to create a common interface for the two classes, as would be the case in a language like Java that does not support union types.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

1. The superclass call does not have to be the first line of the method, but in practice, it almost always is.
2. `AutoSave` has two fields that extend `goog.Disposable`: `saveButton` and `failureDialog`. Only `failureDialog` is disposed in `disposeInternal` because, as the comment on the field implies, it is created by `AutoSave` and is therefore "owned by" it. In contrast, `saveButton` was created by another class and is passed in to `AutoSave`'s constructor. That means that even though `AutoSave` may be disposed, there could be other classes that still have references to `saveButton` and are using it, so `saveButton` should not be disposed yet.

   However, it is also possible that the class that created the instance of `AutoSave` also created the `saveButton` that it uses and wants `AutoSave` to be responsible for disposing of it. Assuming this is always true, the comment on `AutoSave`'s constructor should be updated as follows:

   ```
   * @param {!goog.ui.Button} saveButton Button to disable while saving.
   *     AutoSave is responsible for disposing of saveButton.
   ```

   or this sentiment could be captured explicitly through code:

   ```
   * @param {!goog.ui.Button} saveButton Button to disable while saving.
   * @param {boolean} ownsSaveButton true if this instance of AutoSave should
   *     be responsible for disposing of saveButton.
   ```

   Note that when disposing of `failureDialog`, `goog.dispose(this.failureDialog)` is called rather than `this.failureDialog.dispose()`. It is possible that `failureDialog` is never set during the lifetime of `AutoSave`, in which case `this.failureDialog.dispose()` would throw a null pointer error. `goog.dispose(arg)` checks to see whether `arg` is non-null and has a `dispose()` method, and if so, invokes it. This makes `goog.dispose()` a better choice for fields that may be null.
3. Similar to the previous case, because `AutoSave` adds a listener, it is responsible for making sure it gets removed. Because `AutoSave` follows the standard set forth by the W3C DOM Level 2 Events Specification, it needs to maintain references to both `container` and `eventListener` so that it can call `removeEventListener()` in `disposeInternal()`. This is somewhat awkward, and as noted in `AutoSave`'s constructor, not the recommended way to manage events in the Closure Library (this code will not even work in Internet Explorer 8 and earlier). A better example will be provided in the following chapter.

   It is not required to wait until `disposeInternal` is called to remove event listeners. For example, when `failureDialog` is created, it may have its own click handler. If `failureDialog` is torn down but `AutoSave` still exists, any handlers associated with `failureDialog` should also be removed to release the memory associated with those listeners.

   Like item 2, `AutoSave` is only responsible for removing listeners that it owns. Generally, the class that added the listener should be considered its owner, but also like item 2, it may be necessary to delegate that ownership to another class. If that

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

should be added. (Recall that many conventions in Closure are focused on reducing code size, so such shortcuts are common.) Because the names for native DOM events can vary by browser, the type argument should always be specified as a value from the `goog.events.EventType` enum when adding a listener for such an event (also, to avoid typos). The values of the enum may vary depending on the browser in which the Closure library is being executed, so it is imperative to use this abstraction.

`listener`, the third argument, is either a function to be called when the event fires or an object with a property named `handleEvent` that references a function to call when the event fires. (In the latter case, `this` will be bound to the listener when `handleEvent` is called, not `opt_handler`, so it is effectively a method invocation.) In either case, the function will receive the event as its only argument when it is called.

`opt_capture, the fourth argument,` is a boolean that indicates whether the listener will be called during the capturing or bubbling phase. Like all optional boolean arguments in Closure, the default value is `false`, so listeners will be triggered during the bubbling phase by default. Bubbling versus capturing is discussed in greater detail in the next section, `goog.events.EventTarget`.

`opt_handler`, the fifth argument, is the object to bind the `this` argument to when the listener function is called. If `opt_handler` is unspecified, then `this` will be bound to the `src` argument instead. This is a convention that dates back to the DOM Level 0 event system:

```
HTML (inline event registration):
<div onclick="alert(this.style.color)" style="color: red" id="red-div">

JavaScript (traditional event registration):
document.getElementById('red-div').onclick = function() {
    alert(this.style.color);
};
```

In the previous example, both methods of adding the `onclick` handler are equivalent. Both define an anonymous function that contains a reference to `this`. In either case, when the listener is called, `this` will be bound to the `red-div` element that is the source of the click event, which is exactly what would happen if the equivalent were done in Closure:

```
goog.events.listen(goog.dom.getElement('red-div'),
                   goog.events.EventType.CLICK,
                   function(e) { alert(this.style.color); });
```

Because the events system will do the right substitution when calling the listener, `opt_handler` does not need to be specified.

Note that this fixes an important bug on Internet Explorer. Consider the other two methods for event registration supported natively by web browsers:

```
// Microsoft event registration; available on Internet Explorer.
document.getElementById('red-div').attachEvent('onclick',
    function(e) { alert(this.style.color); });
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

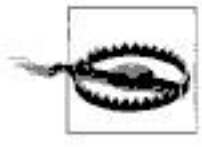
```
example.Inbox.prototype.poll = function() {
  var messages = this.getNewMessages();
  if (messages.length) {
    this.dispatchEvent(example.Inbox.EventType.NEW_MAIL);
  }
};

/** @enum {string} */
example.Inbox.EventType = {
  NEW_MAIL: goog.events.getUniqueId('new_mail')
};
```

A listener to update the unread count in the UI could be added as follows:

```
// inbox is of type example.Inbox.
inbox.addEventListener(example.Inbox.EventType.NEW_MAIL,
    function(e) {
      var inbox = /** @type {!example.Inbox} */ (e.target);
      var count = inbox.getUnreadCount();
      goog.dom.setTextContent(goog.dom.getElement('unread-count'), count);
    });
```

Because `example.Inbox` is a `goog.events.EventTarget`, it has a `dispatchEvent()` method that it invokes in `poll()`. Although all it passes to `dispatchEvent()` is a string, the Closure events system repackages this information as a `goog.events.Event` whose `type` is `goog.events.getUniqueId('new_mail')` and whose `target` is the `example.Inbox` that dispatched the event. This is a powerful system that requires little setup on the part of the developer.

As demonstrated by the example, it is common in Closure to declare a semantic "inner class" on a `goog.events.EventTarget` named `EventType`. The `EventType` class is actually an enum of type `string` whose values correspond to the types of events dispatched by the `EventTarget` to which it "belongs." Examples of classes that follow this pattern are `goog.events.KeyHandler.EventType` and `goog.dom.FontSizeMonitor.EventType`, though there are many others in the Closure codebase.



`goog.events.getUniqueId()` adds a unique suffix to the event name so that it will be distinct from another event type with the identifier `'new_mail'`. (It is not uncommon for a generic identifier such as `'update'` to be used for more than one event type in an application.) An event whose name is amended using `goog.events.getUniqueId()` must be referenced via a variable name (`example.Inbox.EventType.NEW_MAIL`) rather than the original string literal (`'new_mail'`) to ensure that the event type is referenced consistently. Note that `goog.events.getUniqueId()` is not applied to the values of the `goog.events.EventType` enum because the browser relies on the exact name of its built-in events when registering a listener.

Note that `goog.events.EventTarget` extends `goog.Disposable`, so classes that extend `goog.events.EventTarget` will already have the `disposeInternal()` method available.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
var eventWithNullTarget = new goog.events.Event('test', null);
alert(eventWithNullTarget.target == null); // alerts true
goog.events.dispatchEvent(eventTarget, eventWithNullTarget);
alert(eventWithNullTarget.target == null); // alerts false
```

The correct way to pass data to an event listener is to create a custom event by subclassing `goog.events.Event`.

Consider the `Inbox` example, in which an `Inbox` dispatches an event when it discovers new mail. Suppose the listener needs the email address of the `Inbox`'s owner as well as the timestamp when the event was fired. First, a new class must be created for the custom event:

```
goog.provide('example.Inbox.NewMailEvent');

goog.require('goog.events.Event');

/**
 * @param {example.Inbox} inbox
 * @param {string} email
 * @param {Date} timestamp
 * @constructor
 * @extends {goog.events.Event}
 */
example.Inbox.NewMailEvent = function(inbox, email, timestamp) {
  goog.events.Event.call(this, example.Inbox.EventType.NEW_MAIL, inbox);

  /**
   * @type {string}
   * @private
   */
  this.email_ = email;

  /**
   * @type {Date}
   * @private
   */
  this.timestamp_ = timestamp;
};
goog.inherits(example.Inbox.NewMailEvent, goog.events.Event);

/** @return {string} */
example.Inbox.NewMailEvent.prototype.getEmail = function() {
  return this.email_;
};

/** @return {Date} */
example.Inbox.NewMailEvent.prototype.getTimestamp = function() {
  return this.timestamp_;
};
```

Next, the call to `dispatchEvent()` will have to be changed accordingly:

```
example.Inbox.prototype.poll = function() {
  var messages = this.getNewMessages();
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Along those lines, be careful when introducing new optional arguments to a function that is used as a listener function. Consider the following example:

```
// ... inside the constructor ...
this.handler.listen(button, goog.events.EventType.CLICK, this.save);

// Developers often omit the click event from the list of
// parameters to save a byte or because they're too lazy to add an
// @param tag to the JSDoc.
goog.SomeClass.prototype.save = function() {
  // The details of the click event are unimportant; the developer
  // just wants save() to be called when a button is clicked.
  this.writeContentsTo('default.txt');
  this.updateStatusMessage();
};
```

Generally speaking, appending a new optional argument to a list of parameters should not break any existing code, but consider the following change to the `save()` method:

```
/**
 * @param {string=} fileName Defaults to 'default.txt'.
 */
goog.SomeClass.prototype.save = function(fileName) {
  fileName = fileName || 'default.txt';
  this.writeContentsTo(fileName);
  this.updateStatusMessage();
};
```

The developer assumes that previously, no callers were passing any arguments to `save()`, so it should be safe to add an optional parameter. Unfortunately, this is not the case because when `this.save` is called as a listener of a click event, it will be invoked with a `goog.events.Event` as its only parameter, which `save()` will now treat as the optional `fileName` parameter, resulting in a confusing error. The solution is to separate the save logic (so it is reusable) from the event handler as follows:

```
// ... inside the constructor ...
this.handler.listen(button, goog.events.EventType.CLICK,
    this.onSaveClicked_);

/**
 * @param {goog.events.BrowserEvent} e
 * @private
 */
goog.SomeClass.prototype.onSaveClicked_ = function(e) {
  this.save();
};

goog.SomeClass.prototype.save = function() {
  this.writeContentsTo('default.txt');
  this.updateStatusMessage();
};
```

This way, `save()` can be modified and reused without changing the behavior of the event listener. Do not be concerned about the additional bytes introduced by breaking

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CHAPTER 7
Client-Server Communication

The `goog.net` package contains various utilities for communicating with a server from a web browser. With the growing popularity of Ajax, the most common technique is to use an `XMLHttpRequest` object to request updates from the server. As this chapter will explain, that is not the only option, but it is certainly a good place to start.

Server Requests

Like all modern JavaScript libraries, Closure provides a cross-browser abstraction for sending an `XMLHttpRequest` to the server. This section introduces the classes in `goog.net` that support this abstraction.

goog.net.XmlHttp

In the simplest case, the built-in `XMLHttpRequest` object is used to load the contents of a URL asynchronously and then pass the data to another function:

```
var getDataAndAlertIt = function(url) {
  // This line will not work on Internet Explorer 6.
  var xhr = new XMLHttpRequest();
  xhr.open('GET', url, true /* load the URL asynchronously */);
  xhr.onreadystatechange = function() {
    // When an XHR enters ready state 4, that indicates that the request
    // is complete and that all of the data is available.
    if (xhr.readyState == 4) {
      // In this case, alert() is used to handle the data.
      alert('This alert will display second: ' + xhr.responseText);
    }
  };
  xhr.send(null);
};

// Because the XHR will load the data asynchronously, this call will return
// immediately even though the data is not loaded yet.
getDataAndAlertIt('http://www.example.com/testdata.txt');
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Customizing the request

In the examples shown so far, `goog.net.XhrIo` is only used to make `GET` requests, but its `send()` methods have several optional arguments that can be used to configure the parameters of the request. The signatures of the two methods are as follows:

```
/**
 * Static send that creates a short lived instance of XhrIo to send the
 * request.
 * @param {string|goog.Uri} url Uri to make request to.
 * @param {Function=} opt_callback Callback function for when request is
 *     complete.
 * @param {string=} opt_method Send method, default: GET.
 * @param {string|GearsBlob=} opt_content POST (or PUT) data. This can be a
 *      Gears blob if the underlying HTTP request object is a Gears HTTP request.
 * @param {Object|goog.structs.Map=} opt_headers Map of headers to add to the
 *     request.
 * @param {number=} opt_timeoutInterval Number of milliseconds after which an
 *     incomplete request will be aborted; 0 means no timeout is set.
 */
goog.net.XhrIo.send = function(url, opt_callback, opt_method, opt_content,
                               opt_headers, opt_timeoutInterval) { /* ... */ };

/**
 * Instance send that actually uses XMLHttpRequest to make a server call.
 * @param {string|goog.Uri} url Uri to make request too.
 * @param {string=} opt_method Send method, default: GET.
 * @param {string|GearsBlob=} opt_content POST (or PUT) data. This can be a
 *      Gears blob if the underlying HTTP request object is a Gears HTTP request.
 * @param {Object|goog.structs.Map=} opt_headers Map of headers to add to the
 *     request.
 */
goog.net.XhrIo.prototype.send = function(url, opt_method, opt_content,
                                         opt_headers) { /* ... */ };
```

The two signatures are nearly identical, but the static method has two extra parameters. The first is `opt_callback`, which will be called as a listener of the `COMPLETE` event. This is not present in the instance method because clients are expected to add their own listeners to the instance rather than specifying a callback.

The second extra parameter is `opt_timeoutInterval`, which is used to set an upper bound on how long a request will try to fetch data from the server. If the request exceeds the timeout, the request is aborted. Clients of the instance method should invoke the `setTimeoutInterval()` method to set the length of the timeout (the default is 0, which means no timeout limit is set). If the `goog.net.XhrIo` is reused, it will use the same timeout interval for subsequent requests.

The `url` argument specifies the destination of the request, so it is not optional.

The `opt_method` argument specifies the method of the request. Valid method values are defined by the HTTP specification: *http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html*. The default value for both `send()` methods is `'GET'`.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

as private because they are designed to be used exclusively by `goog.net.XhrManager`. For example, instead of calling `request.setAborted(true)` to abort a request, the manager's `abort()` method should be used.

For every XHR managed by a `goog.net.XhrManager`, it dispatches all of the events listed in Table 7-1 except for `TIMEOUT` and `READY_STATE_CHANGE`. When dispatching an event, instead of using `goog.events.Event`, it uses its own subclass, `goog.net.XhrManager.Event`, which includes the `id` of the request as well as the underlying `goog.net.XhrIo`.

Because requests are reused by the `goog.net.XhrManager`, it is important to extract all properties of interest from the XHR in the callback rather than in a timeout scheduled by the callback as demonstrated in the previous section.

goog.Uri and goog.uri.utils

A `goog.Uri` is a class in the Closure Library that represents a URI, which is a string that identifies a resource on the Internet. URI stands for "Uniform Resource Identifier." The term is commonly (and erroneously) used interchangeably with URL, which stands for "Uniform Resource Locator," which is the string of characters you type into a browser to go to a website, such as `http://www.google.com/`. There are many resources online that will explain the differences between a URI and a URL in detail, but the most important thing to know is that every URL is a URI, so `http://www.google.com/` is both a URL and a URI. Therefore, any method in the Closure Library that accepts a URI must also accept a URL.

Although it is common to simply pass a URI as a string in Closure, it is helpful to have a class that represents a URI in order to access or modify its various components. For example, consider the following URI represented as a string in JavaScript:

```
var uri = 'https://username:passwd@test.example.com:8080/' +
    'foo/bar/index.html?param1=a&param2=b#ch3';
```

In order to extract the query string of the URI (which is the part represented by `param1=a&param2=b`), a fair bit of parsing would have to be done on `uri`. Fortunately, `goog.Uri` can take care of this for you:

```
var googUri = new goog.Uri(uri);
googUri.getScheme();    // 'https'
googUri.getUserInfo();  // 'username:passwd'
googUri.getDomain();    // 'test.example.com'
googUri.getPort();      // 8080
googUri.getPath();      // '/foo/bar/index.html'
googUri.getQuery();     // 'param1=a&param2=b'
googUri.getFragment();  // 'ch3'
```

Note that the values returned by `goog.Uri`'s getters differ slightly from the properties of the `window.location` object in JavaScript. For example, if the user were at the URL specified by `uri` in a browser, `window.location.protocol` would include a trailing colon (`'https:'`), `window.location.hash` would include the hash character (`'#ch3'`), and `window`

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goog.net.ImageLoader

Another utility for loading resources in parallel is `goog.net.ImageLoader`, though as you might guess, it works only with images. Images are registered with the loader via its `addImage(id, image)` method. The `image` argument is either a URL to the image itself or an `<img>` element whose `src` may not have loaded yet. In either case, the `id` must be unique within the `goog.net.ImageLoader` (if the `id` is reused, it will replace the previous image), so the URL of the image is a good default value to use as an identifier:

```
<img id="logo" src="http://www.google.com/intl/en_ALL/images/logo.gif">
<script>
var loader = new goog.net.ImageLoader();

var addToLoader = function(image) {
  // Use the URL as the id for addImage().
  var id = goog.isString(image) ? image : image.src;
  loader.addImage(id, image);
};

// Either a URL or an <img> element may be used as an argument to addImage().
addToLoader('http://www.google.com/favicon.ico');
addToLoader(goog.dom.getElement('logo'));
</script>
```

Like `goog.net.BulkLoader`, it is possible to register listeners before kicking off the loading of resources. There are three event types of interest, the target of which will be a JavaScript `Image` object. The event types are as follows:

goog.events.EventType.LOAD
: Dispatched once for each image that successfully loads.

goog.net.EventType.ERROR
: Dispatched once for each image that fails to load.

goog.net.EventType.COMPLETE
: Fired after a load or error event has been fired for each image in the loader. Note that this is not the same as the `goog.net.EventType.SUCCESS` event dispatched by `goog.net.BulkLoader`, as a `COMPLETE` event will be fired even when some images fail to load.

Once the listeners are in place, the loader can be kicked off via its `start()` method:

```
/** @type {!Array.<Image>} */
var failedImages = [];

/** @type {!Array.<Image>} */
var loadedImages = [];

var eventTypes = [
    goog.events.EventType.LOAD,
    goog.net.EventType.ERROR,
    goog.net.EventType.COMPLETE
];
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Table 7-2. URLs that are considered to have a different origin than http://example.com/mypage.html.

URL	Why access is denied
http://google.com/	Different domain
https://example.com/mypage.html	Different protocol
http://example.com:8080/index.jsp	Different port
http://www.example.com/mypage.html	Different subdomain

This may seem overly restrictive, but there are compelling security issues behind the same origin policy.

goog.net.jsonp

Although an XHR must obey the same-origin policy, the URL supplied to the `src` attribute of an `<img>` or `<script>` tag does not, creating an interesting loophole for fetching data across domains. A popular "exploit" of this loophole is a technique called JSONP (which is short for "JSON with padding"), which is used to fetch data from another domain.

The mechanics behind JSONP are fairly simple: instead of responding to a GET request with data in a raw format such as JSON, respond to the request with JavaScript code that calls a function with the data as an argument. For example, if a request to `http://example.com/birthdays?callback=showBirthdays` returned the following:

```
showBirthdays([
  {"name": "Abraham Lincoln", date: "February 12, 1809"},
  {"name": "George Washington", date: "February 22, 1732"}
]);
```

It would then be possible for a page on a third-party site to use this data with the following script tags:

```
<ul id="birthdays"></ul>
<script>
var showBirthdays = function(birthdayData) {
  var ul = goog.dom.getElement('birthdays');
  for (var i = 0; i < birthdayData.length; i++) {
    var datum = birthdayData[i];
    var li = goog.dom.createElement('li');
    goog.dom.setTextContent(li, datum['name'] + ' ' + datum['date']);
    ul.appendChild(li);
  }
};
</script>
<script src="http://example.com/birthdays?callback=showBirthdays"></script>
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
    // Once the child channel is created, try to establish the connection.
    var connectCallback = function() {
      // Give the parent a chance to connect by using setTimeout().
      setTimeout(function() {
        childChannel.send('echo', 'Hello world!');

        // This will be serialized to JSON automatically.
        childChannel.send('car', {"from": "EWR", "to": "JFK", "when":"15:00"});
      }, 1000);
    };
    childChannel.connect(connectCallback);
  } else {
    alert('No xpc parameter provided!');
  }
});
```

As suggested by the use of `setTimeout()`, establishing channel connections is asynchronous, so the parent-to-child connection may be established before the child-to-parent connection, and vice versa. Unfortunately, `goog.net.xpc.CrossPageChannel` does not provide its own API to verify that the channel is working in both directions: its `isConnected()` method verifies only the connection for the channel on which it is invoked. If you are averse to using `setTimeout()`, one option is to establish your own handshake using `registerService()` and `send()`. For example, the outer peer could register a service named `ping` that would send a response to an inner peer service named `ack`. The code for the outer peer would be as follows:

```
var isInnerPeerConnected = false;

// Code in the outer peer.
parentChannel.registerService('ping', function() {
  if (!isInnerPeerConnected) {
    isInnerPeerConnected = true;
    parentChannel.channel_.send('ack', null);
    // Do any work related to establishing the initial bi-directional
    // communication channel.
  }
});
```

The inner peer would set up its `ack` service as follows:

```
var isOuterPeerConnected = false;

childChannel.registerService('ack', function() {
  var isInitialConnection = !isOuterPeerConnected;
  isOuterPeerConnected = true;
  if (isInitialConnection) {
    // Do any work related to establishing the initial bi-directional
    // communication channel.
  }
});

var tryPing = function() {
  if (isOuterPeerConnected) {
    return;
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In `streamXhr`, there is always one XHR that is connected to the server, and updates from the server are streamed down to the client using that connection. This means that sending updates to the server must be made using other individual XHRs. Dylan Schiemann, the author of "Scaling with Comet," is also a cofounder of Comet Daily, LLC, so you can also find even more information about Comet techniques on the company's website: *http://cometdaily.com*.

Incomplete pieces of Comet library code can be seen in the `goog.net` codebase. For example, the `htmlfile` trick to prevent the clicking sound in Internet Explorer when using the "forever frame" technique described in "Scaling with Comet" can be seen in the `tridentGet_` method of `goog.net.ChannelRequest`. Unfortunately, the server code that these libraries were designed to be used with has not been open-sourced, so it is not easy to use this code directly. (There is a simpler helper function for the forever frame technique in `goog.net.IframeIo` called `handleIncrementalData` that has better documentation, but it lacks the `htmlfile` workaround for Internet Explorer.) Basically, if you have already been exposed to Comet and are curious to see how Google's client-side implementation works (even though you cannot use it straight out of the box), then take a look at `goog.net.BrowserChannel`, `goog.net.ChannelRequest`, and `goog.net.IframeIo`.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

To display a component, it is clear from the diagram that rendering and decorating are mutually exclusive, and are achieved by calling `render()` or `decorate()`, respectively. To add custom behavior to either display method, override `createDom()` to change the behavior of `render()`, or override `decorateInternal()` to change the behavior of `decorate()`. Neither `render()` nor `decorate()` should be overridden directly.

By convention in the Closure Library, a method whose name ends in `Internal` (such as `disposeInternal()` or `decorateInternal()`, for example) is almost always a protected method that is designed to be overridden.

Another important result from this diagram is that it is acceptable for `createDom()` to call `decorateInternal()`, but it is not acceptable for it to call `decorate()` because that would result in `enterDocument()` being called more than once, which is prohibited. As the examples in this chapter demonstrate, calling `decorateInternal()` from `createDom()` is not uncommon.

Instantiation

When a `goog.ui.Component` is constructed, the only argument the constructor takes is a `goog.dom.DomHelper`. The argument is optional, but it is good practice to always supply it in case your application evolves to use multiple frames. When it is not specified, `goog.dom.getDomHelper()` will be used to produce the `goog.dom.DomHelper`, but that returns a `goog.dom.DomHelper` for the document in which the JavaScript is loaded, which may be different than the document into which the component will be added. (The latter is the correct one.) Most likely you will have a reference to the DOM element to which the component will be added after constructing it, so that element can be used to produce the appropriate `goog.dom.DomHelper`:

```
// parentElement is the element to which the component will be added. By
// using it as the argument to goog.dom.getDomHelper(), it ensures that
// parentElement and dom will refer to the same document.
var dom = goog.dom.getDomHelper(parentElement);

// Create the Component using the DomHelper.
var component = new goog.ui.Component(dom);

// Render the component into parentElement. By construction, it is guaranteed
// that parentElement and component will operate on the same document.
component.render(parentElement);
```

Similarly, when adding a component as a child of another component, it is possible to get the appropriate `goog.dom.DomHelper` from the parent and use it when creating the child:

```
var dom = existingComponent.getDomHelper();
var child = new goog.ui.Component(dom);
existingComponent.addChild(child, true /* opt_render */);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Note that it may be tempting to include display logic that is common to both rendering and decorating in `enterDocument()` so that it is sure to be called, regardless of which display method is used. In many cases, this is a mistake because `enterDocument()` may be called multiple times for a child component, as explained in the next section. It is better to refactor such one-time display logic into a common method that is exercised by both `createDom()` and `displayInternal()` so that it is only executed once.

Disposal

When `dispose()` is called on a component that has been displayed, its `exitDocument()` method will be called in addition to `disposeInternal()`. The implementation of `exitDocument()` should complement the `enterDocument()` method, so any listeners that were added in `enterDocument()` should be removed in `exitDocument()`. Fortunately, the default implementation of `exitDocument()` calls `this.getHandler().removeAll()`, so it is not often necessary to override `exitDocument()`.

Like `enterDocument()`, `exitDocument()` may be called multiple times for a child component, so any cleanup logic that should be run exactly once should be defined in `disposeInternal()` rather than `exitDocument()`.

Components with Children

Components in Closure can be built up in a treelike fashion to create sophisticated user interfaces. Just like the DOM, a tree of Closure components is mutable, and nodes in the tree may be removed and added back later. This means that when a child component is added to a parent component, it is possible to remove the child from the parent without invoking its `dispose()` method. This introduces some important extensions to the life cycle introduced in the previous section, as shown in Figure 8-3.

When a hierarchy of components is built up, the `setParentEventTarget()` method that a component inherits from `goog.events.EventTarget` is used to ensure that the component hierarchy matches the event target hierarchy. Therefore, an important invariant for components is that the element of a child component (as determined by its `getElement()` method) must be a descendant of the parent component's element. This ensures that both DOM events and Closure Library events flow through the component hierarchy in a consistent manner.

Adding a child

The children of a component are organized as a list. A component can be appended as the last child of another component using `addChild(child, opt_render)`, or it can be added to a specific position among the children using `addChildAt(child, index, opt_render)`. Calling `parentComponent.addChild(child, opt_render)` is equivalent to calling `parentComponent.addChildAt(child, parentComponent.getChildCount(), opt_render)`. If `index` is out of bounds, then a `goog.ui.Component.Error.CHILD_INDEX_OUT_OF_BOUNDS` error will be thrown.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Name	Description
getChildCount()	Returns the number of children of this component.
getChildIds()	Returns an array containing the IDs of the children of this component.
getParent()	Returns the component's parent, if any.
hasChildren()	Returns a boolean indicating whether this component has children.
indexOfChild(child)	Returns the 0-based index of the given child component, or –1 if no such child is found.

Events

As shown in Figure 8-1, `goog.ui.Component` is a subclass of `goog.events.EventTarget`. This is because `goog.ui.Component` is designed to dispatch many of its own events, which are defined in the enum `goog.ui.Component.EventType`. These event types are listed in Table 8-2.

Table 8-2. Values of the goog.ui.Component.EventType enum.

Name	Description
`BEFORE_SHOW`	Dispatched before a component becomes visible.
`SHOW`	Dispatched before a component becomes visible.
`HIDE`	Dispatched before a component becomes hidden.
`DISABLE`	Dispatched before a component becomes disabled.
`ENABLE`	Dispatched before a component becomes enabled.
`HIGHLIGHT`	Dispatched before a component becomes highlighted.
`UNHIGHLIGHT`	Dispatched before a component becomes unhighlighted.
`ACTIVATE`	Dispatched before a component becomes activated.
`DEACTIVATE`	Dispatched before a component becomes deactivated.
`SELECT`	Dispatched before a component becomes selected.
`UNSELECT`	Dispatched before a component becomes unselected.
`CHECK`	Dispatched before a component becomes checked.
`UNCHECK`	Dispatched before a component becomes unchecked.
`FOCUS`	Dispatched before a component becomes focused.
`BLUR`	Dispatched before a component becomes blurred.
`OPEN`	Dispatched before a component is opened (expanded).
`CLOSE`	Dispatched before a component is closed (collapsed).
`ENTER`	Dispatched after a component is moused over.
`LEAVE`	Dispatched after a component is moused out of.
`ACTION`	Dispatched after the user activates a component.
`CHANGE`	Dispatched after the external-facing state of a component is changed.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Handling User Input

Because a `goog.ui.Control` is responsible for interpreting user input, it registers a number of event listeners in its `enterDocument()` method. Table 8-5 lists all of the possible events a `goog.ui.Control` may listen for by default and the corresponding method that is used as an event handler. To change the behavior of how a control responds to an input event, override the appropriate method in Table 8-5.

Table 8-5. Event listeners registered by goog.ui.Control.

Event	Method	Notes
`goog.events.EventType.MOUSEOVER`	`handleMouseOver()`	Added only when `isHandleMouseEvents()` returns `true`, which is its default behavior.
`goog.events.EventType.MOUSEDOWN`	`handleMouseDown()`	Added only when `isHandleMouseEvents()` returns `true`, which is its default behavior.
`goog.events.EventType.MOUSEUP`	`handleMouseUp()`	Added only when `isHandleMouseEvents()` returns `true`, which is its default behavior.
`goog.events.EventType.MOUSEOUT`	`handleMouseOut()`	Added only when `isHandleMouseEvents()` returns `true`, which is its default behavior.
`goog.events.EventType.DBLCLICK`	`handleDblClick()`	Added only when `isHandleMouseEvents()` returns `true`, which is its default behavior. Unlike the other mouse event listeners, this one is added only for Internet Explorer.
`goog.events.KeyHandler.EventType.KEY`	`handleKeyEvent()`	Added only when the control supports the `goog.ui.Component.State.FOCUSED` state and its `getKeyEventTarget()` method returns an `Element`. There is also a protected `handleKeyEventInternal()` method that may be more appropriate to override.
`goog.events.EventType.FOCUS`	`handleFocus()`	Added only when the control supports the `goog.ui.Component.State.FOCUSED` state and its `getKeyEventTarget()` method returns an `Element`. The focus listener will be added to this element.
`goog.events.EventType.BLUR`	`handleBlur()`	Added only when the control supports the `goog.ui.Component.State.FOCUSED` state and its `getKeyEventTarget()` method returns an `Element`. The blur listener will be added to this element.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

When creating your own subclass of `goog.ui.Control`, you should override `decorate()` and `createDom()` in the associated subclass of `goog.ui.ControlRenderer` rather than in the `goog.ui.Control` subclass itself. There is also an `initializeDom()` method called from the control's `enterDocument()` method that is a candidate for overriding.

Generally, a renderer is designed to be stateless so that it can be shared by multiple instances of a control. For this reason, many of the methods of `goog.ui.ControlRenderer` take a `goog.ui.Control` as an argument rather than storing it as a field.

Be aware that the default implementation of `goog.ui.ControlRenderer`'s `decorate()` method uses the id of the control's element (if available) to use as the control's id. This means that ids for controls and DOM nodes can end up sharing the same namespace.

In terms of updating its display, rather than the renderer establishing itself as a listener for the many events dispatched by a control, the control calls the renderer's `setState()` method when a state transition occurs. Because the renderer does not register itself as a listener of the control, the control does not need to support transition events for the state in order for the renderer to be notified of the change.

When the renderer's `setState()` method is called by the control, it takes the element of the control (as returned by its `getElement()` method) and updates the CSS classes defined on the element to reflect the new state of the control. Each state has a corresponding CSS class that is a combination of the result of the renderer's `getStructuralCssClass()` method and the name of the state. For example, because `getStructuralCssClass()` returns `'goog-control'` for `goog.ui.ControlRenderer`, a `goog.ui.ControlRenderer` will add the CSS class `goog-control-hover` to a control's DOM when the control transitions into the `HOVER` state. If the control leaves the `HOVER` state, then the CSS class will be removed. Therefore, it is possible to determine the state of a `goog.ui.Control` by inspecting the CSS classes defined on its element.

To determine the CSS class that corresponds to a state for a `goog.ui.ControlRenderer`, use the all-lowercase name of the state as the suffix and append that to the value of `getStructuralCssClass()` plus a hyphen. This class name can also be constructed programmatically by calling `goog.getCssName(renderer.getStructuralCssClass(), 'hover')`.

In addition to using CSS classes to represent the state of a control, a control also uses a special CSS class to identify itself in the DOM. This identifier class is returned by the renderer's `getCssClass()` method. For example, the `getCssClass()` method of `goog.ui.ControlRenderer` returns `'goog-control'`, just like its `getStructuralCssClass()` method (a renderer's `getStructuralCssClass()` method delegates to its `getCssClass()` method by default). Although classes that reflect state, such as `goog-control-hover`, may be

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Example: Responding to a Mouseover Event

This intricate system gives you several options for updating the display of a `goog.ui.Control` in response to a mouseover event:

- You can leverage the CSS pseudo-class `:hover` to update the presentation of a control when the user mouses over it. Unfortunately, using this class may cause performance problems and works only on anchor elements in IE6, so this is not a good solution.
- You can register a listener on the control for a `goog.ui.Component.EventType.ENTER` event, which is dispatched only when the mouse initially enters the root element for the control (but not when it enters one of its child elements). This event will be dispatched regardless of whether the control is enabled, though the listener can always test whether the control is enabled by invoking its `isEnabled()` method.
- You can call `setDispatchTransitionEvents(goog.ui.Component.State.HOVER)` and then register a listener for a `goog.ui.Component.EventType.HIGHLIGHT` event. This is identical to listening for `goog.ui.Component.EventType.ENTER`, except it is dispatched only when the control is enabled.
- You can override the control's `handleMouseOver()` method, but that will receive every mouseover event for the control's element and mouse over events from its descendant elements which bubble up. You are likely only interested in mouseover events for the root element, which would be filtered out for you by the control if you listened for either its `goog.ui.Component.EventType.ENTER` or `goog.ui.Component.EventType.HIGHLIGHT` events, so this is likely not a good solution.
- You can use a custom `goog.ui.ControlRenderer` and override its `setState()` method to handle the case where the state of `goog.ui.Component.State.HOVER` changes.
- You can define rules for the CSS class that will be added to the control by its renderer, such as `goog-control-hover`. This makes it possible to change the presentation without writing any JavaScript code (or adding another listener), and avoids the limitations of the `:hover` pseudo-selector.

Each of these options has its own pros and cons. For example, if the response to the mouseover event is universal to all instances of a control, then it should be encoded in the `setState()` method of their common renderer. However, it is better to encode behavior unique to a single instance of a control in an event listener so it can be applied individually. Using `goog.ui.Control` makes it possible to choose the appropriate granularity for responding to user input.

goog.ui.Container

A `goog.ui.Container` is a component that serves as a container for other `goog.ui.Control` objects. Even though it subclasses `goog.ui.Component`, its methods for adding and removing components take only `goog.ui.Control` objects as arguments. In the

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

`Card` to determine what changes you want to make in the version of `hovercard.css` that exists in your project. This will also make it easier to inspect the DOM of the widget in a tool like Firebug, which will reveal all CSS classes used in the widget. You may wish to add rules for these CSS classes in your own stylesheet.

Be aware that some widgets, including `goog.ui.Bubble`, `goog.ui.Checkbox`, and `goog.ui.MenuItem`, specify a `background-image` in their default stylesheet, so if the CSS from the stylesheet is used as-is, then the images they use must also be copied into your project and served as part of your application. It is likely that you will need to modify the path to the image in the value of the `background-image` rule to match its location on your application's server.

Another common issue with CSS and the Closure Library is supporting the use of multiple instances of a widget that should be styled differently. For example, consider the case where some hovercards are supposed to be green to represent contacts that are your friends, and other hovercards are supposed to be red to represent contacts to avoid. One solution is to include an additional CSS class on the root element of the widget and then add some CSS rules that apply only when multiple classes are present:

```
/* Original CSS rule defined in hovercard.css */
.goog-hovercard div {
  background-color: white;
}

/* The following CSS rules will take precedence over the original rule if
   either .good-contact or .bad-contact is also defined as a class on the
   element with the .goog-hovercard class. */
.goog-hovercard.good-contact div {
  background-color: green;
}
.goog-hovercard.bad-contact div {
  background-color: red;
}
```

As explained earlier, the one problem with this approach is that IE6 does not support multiple class selectors, so if supporting IE6 is a requirement, then using multiple class selectors is not an option (unless the widget is a `goog.ui.Control`, in which case `getIe6ClassCombinations()` can be used to work around this issue). A similar trick is to use descendant selectors, which works on all modern browsers, including IE6:

```
.good-contact .goog-hovercard div {
  background-color: green;
}
.bad-contact .goog-hovercard div {
  background-color: red;
}
```

Though this requires adding the `goog.ui.HoverCard` to an element that has either `good-contact` or `bad-contact` defined on it:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

But because goog-inline-block is so useful, it is common to include the definition of goog-inline-block in any user interface built with the Closure Library, regardless of whether goog.ui.INLINE_BLOCK_CLASSNAME is required.

Note that the function goog.style.setInlineBlock(element) that was introduced in Chapter 4 programmatically updates the style of element to behave as if the goog-inline-block class were applied to it. This is useful in an environment where goog-inline-block is not defined in the CSS, for whatever reason. Because it is implemented in code, goog.style.setInlineBlock() uses goog.userAgent rather than CSS selectors to determine the appropriate styles to add.

Example of Rendering a Component: goog.ui.ComboBox

There are several components in the goog.ui package that explicitly prohibit display via decoration, so render() must be used to display them:

- goog.ui.ColorPicker
- goog.ui.ComboBox
- goog.ui.HsvPalette
- goog.ui.PopupColorPicker
- goog.ui.PopupDatePicker

For each widget in this list, its canDecorate(element) method returns false regardless of the value of element. Most of these widgets build up a complex DOM structure in their respective createDom() methods, so verifying such a structure for decoration would be tedious. Therefore, even if decorate() is your preferred way of displaying widgets, it may not be an option for every component you use in your application.

The most commonly used component from the list is goog.ui.ComboBox. A combobox is a combination of a text input and a <select> element: it allows a user to type in text freely, but also offers suggested values via a drop-down menu. An example of goog.ui.ComboBox in action can be seen in Figure 12-1 where a combobox is used to enter a URL to JavaScript source code in the Closure Compiler Service UI. Although the user may type in any URL he wishes, the combobox suggests URLs to common JavaScript libraries, such as *http://ajax.googleapis.com/ajax/libs/jquery/1.3.2/jquery.js*.

To use a goog.ui.Combobox in your own application, first open up its corresponding demo page to determine which CSS files you will need. The stylesheet links can be found under the head element:

```
<link rel="stylesheet" href="css/demo.css">
<link rel="stylesheet" href="../css/menus.css">
<link rel="stylesheet" href="../css/combobox.css">
```

As explained in the previous section, it is not recommended to include demo.css in your application, though the definition of goog-inline-block should be included from

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goog.ui.CustomButton

Building a UI becomes more interesting when using `goog.ui.CustomButton`, which makes it possible to decorate an arbitrary `<div>` so that it looks and behaves like a button:

```
<div id="empty-div"></div>
<script>
var button1 = new goog.ui.CustomButton(null /* content */);
button1.decorate(goog.dom.getElement('empty-div'));
button1.setCaption('The button label');
</script>
```

After decoration, the HTML for the `<div>` is:

```
<div id="empty-div"
     class="goog-inline-block goog-custom-button"
     role="button"
     style="-moz-user-select: none;"
     tabindex="0">
  <div class="goog-inline-block goog-custom-button-outer-box">
    <div class="goog-inline-block goog-custom-button-inner-box">
      The button label
    </div>
  </div>
</div>
```

This is considerably more sophisticated than the decoration performed by `goog.ui.Button`. It adds extra attributes to make the `<div>` behave more like a native button. For instance, the `role="button"` attribute conforms to the W3C specification for accessibility (WAI-ARIA) so that a screen reader will recognize the element as a button. There is also a `tabindex` attribute so that it is possible to navigate to the `goog.ui.CustomButton` and give it keyboard focus, just like a native button.

The WAI-ARIA (Web Accessibility Initiative-Accessible Rich Internet Applications) specification provides an ontology of *roles* and *states* that can be used to annotate the DOM in order to communicate the semantic meaning of an element to a screen reader. Traditionally, screen readers have used the DOM elements themselves to interpret their function: a `<select>` would indicate a list of choices that could be read to a visually impaired person by a screen reader. But with the rise of DHTML to create more sophisticated web interfaces, a `<select>` may now be constructed out of `<div>`s, which do not communicate their semantic meaning by default. Using the roles and state attributes defined in the WAI-ARIA specification fills this communication gap. An enumeration of ARIA roles and states and logic for applying them to DOM elements can be found in the `goog.dom.a11y` package in the Closure Library. (As "internationalization" is frequently abbreviated to "i18n", it is becoming commonplace to abbreviate "accessibility" to "a11y".)

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Next, let's take a look at the control itself:

```
/**
 * A control that displays a ChecklistItem.
 * @param {example.ChecklistItem=} item
 * @param {example.ui.ChecklistItemRenderer=} renderer
 * @constructor
 * @extends {goog.ui.Control}
 */
example.ui.ChecklistItem = function(item, renderer) {
  goog.base(this, null /* content */, renderer);
  this.setSupportedState(goog.ui.Component.State.CHECKED, true);
  this.setAutoStates(goog.ui.Component.State.CHECKED, false);
  this.setSupportedState(goog.ui.Component.State.FOCUSED, false);

  if (!item) {
    item = {id: 'temp-' + goog.ui.IdGenerator.getInstance().getNextUniqueId(),
            text: '',
            checked: false};
  }

  this.setModel(item);
};
goog.inherits(example.ui.ChecklistItem, goog.ui.Control);

/**
 * @return {!example.ChecklistItem}
 * @override
 */
example.ui.ChecklistItem.prototype.getModel;

/** @return {boolean} */
example.ui.ChecklistItem.prototype.isItemChecked = function() {
  return this.getModel().checked;
};

/** @return {string} */
example.ui.ChecklistItem.prototype.getItemText = function() {
  return this.getModel().text;
};

/** @inheritDoc */
example.ui.ChecklistItem.prototype.enterDocument = function() {
  goog.base(this, 'enterDocument');
  var checkbox = this.getChildAt(0);
  this.getHandler().listen(checkbox,
      [goog.ui.Component.EventType.CHECK, goog.ui.Component.EventType.UNCHECK],
      this.onCheckChange_);
};

/**
 * Update the internal ChecklistItem when the checked state of the checkbox
 * changes.
 * @param {goog.events.Event} e
 * @private
 */
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

However, its `decorate()` method is considerably different because it does not instantiate or add any child components explicitly, though somehow they appear to have been added because `checklistContainer.forEachChild()` is used to iterate over them to extract the data to build up the model for `example.ui.Checklist`. It turns out that `ContainerRenderer`'s `decorate()` method delegates to a `decorateChildren()` method that iterates over its child elements and attempts to decorate them and add them as child components using decorator functions available in `goog.ui.registry`. Therefore, the child components are created and added using the call to the `decorate()` method in the superclass. If `goog.ui.ControlRenderer` provided similar functionality, it would have been used in `example.ui.ChecklistItemRenderer`'s implementation of `decorate()` as well, but unfortunately, it does not.

As a container, `example.ui.Checklist` has to do very little work at all:

```
/**
 * @param {example.Checklist=} checklist
 * @constructor
 * @extends {goog.ui.Container}
 */
example.ui.Checklist = function(checklist) {
  goog.base(this, goog.ui.Container.Orientation.VERTICAL,
      example.ui.ChecklistRenderer.getInstance());
  this.setModel(checklist || null);
  this.setFocusable(false);
};
goog.inherits(example.ui.Checklist, goog.ui.Container);

/**
 * @return {example.Checklist}
 * @override
 */
example.ui.Checklist.prototype.getModel;

/** @inheritDoc */
example.ui.Checklist.prototype.enterDocument = function() {
  goog.base(this, 'enterDocument');
  this.getHandler().listen(this,
      [goog.ui.Component.EventType.CHECK, goog.ui.Component.EventType.UNCHECK],
      this.onCheckChange_);
};

/**
 * @param {goog.events.Event} e
 * @private
 */
example.ui.Checklist.prototype.onCheckChange_ = function(e) {
  // The example.ui.Checklist class chooses to keep CHECK and UNCHECK events to
  // itself by preventing such events from bubbling upward. Instead, it expects
  // clients to listen to its custom CHECKED_COUNT_CHANGED events for updates.
  e.stopPropagation();
  this.dispatchEvent(new goog.events.Event(
      example.ui.Checklist.EventType.CHECKED_COUNT_CHANGED, this));
};
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Conclusions

As there is no allowance for adding or removing items, this to-do list leaves much to be desired, but it does exercise many of the features of the `goog.ui` package discussed in this chapter. For starters, it provides an example of a subclass for each of the base classes introduced in Figure 8-1. Further, it demonstrates how to support both rendering and decorating in components as well as renderers. It also shows that even when building a custom component (`example.ui.ChecklistItem`), a lot of effort can be saved by incorporating existing ones in the process (`goog.ui.Checkbox`). Finally, it alludes to the benefits of generating functions for producing HTML from a template, which will be fully realized in Chapter 11.

The one aspect of this component that you may find disconcerting is the number of listeners it adds. It turns out that a control added to a container will have its listeners supplanted by the container's, but the control's children will not have their listeners modified. That means that for every item in this list, the `goog.ui.Checkbox` that is a child of `example.ui.ChecklistItem` adds 11 listeners to the system. Using techniques such as moving listeners from children to a common parent can help reduce this number.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The Closure Editor leans more towards the hands-off approach, but as you can tell from the size of the library, it is clearly doing more than just wrapping the native browser control. It uses `contentEditable` and `execCommand`, so the browser handles text input, layout, drawing the selection, and most of the formatting commands. However, it steps in when the browser's implementation is just too buggy or when an application needs to provide more cross-browser consistency.

One major reason for taking this approach was that the editor was started in 2004 when browser performance was nowhere near as good as it is today. The largest client of the editor was a web application that required a lot of memory to hold all of its state and code: Gmail. The most popular browser at the time, IE 6, had a garbage collector that ran every 256 object allocations, and garbage collection time was dependent on the amount of memory in use. See *http://pupius.co.uk/blog/2007/03/garbage-collection-in-ie6* for more details on IE 6 garbage collection. It was impossible to do any complicated work on every keypress, because if enough object allocations were done to cause a garbage collection, it would lead to considerable lag when typing. Throughout the editor code, one key goal is to never let the user experience lag. To achieve this, the typing code paths are protected. You'll learn more about how this is done using the plugin API in "Extending the Editor: The Plugin System" on page 253.

goog.editor.BrowserFeature

The Closure Editor uses a technique not found elsewhere in the Closure Library for handling browser differences: `goog.editor.BrowserFeature`. This object is a set of constants for different browser features (or bugs) and the browsers that they apply to. Elsewhere in the Closure Library, when browser-specific code is needed, there is usually just an if statement with some feature detection, or a user agent check. However, most of the features, quirks, and bugs that plague rich text are not discoverable using feature detection, so they must be specified ahead of time. This also gives the benefit of making it possible to compile a version that is just for a particular browser, greatly reducing code size. All of these constants are in a single file, which also makes it easier to add support for a new browser, as a new browser's quirks can be added in just one place, rather than needing to weed through the entire library looking for browser checks to modify. In fact, initial Opera support was added by adding `goog.userAgent.OPERA` to a few features in this file, and changing one `if (goog.userAgent.GECKO)` check in the rest of the code base!

Creating an Editable Region

The first step in creating an editor is to create the area on a page where it will go. The Closure editing package provides two basic types of editable regions: `goog.editor.Field` and `goog.editor.SeamlessField`. `goog.editor.Field` is the most simple type of editable area and is like a text area. It is isolated from the rest of your application and lives inside its own `<iframe>`. You can think of it as a "white box" on the page; it is a

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

batched event instead of the change event so that your application does not react too frequently (and expensively) to a rapidly changing DOM.

To track changes, the field listens for certain events known to cause DOM changes and to explicit calls to `handleChange`. These changes are relatively common, so to maintain good performance, the field does as little work as possible on every change, and essentially just queues up the delayed change event if one is not already pending. When change events are "on," the field tracks modifications and when change events are "off," the field does not track changes. Unless you are implementing your own editing commands, you will never need to turn change events on or off. If you are implementing your own editing commands, the editor provides APIs for starting, stopping, and forcing change events. As you'll see in the next section, the `goog.editor.Plugin` API provides abstractions so that you normally will not need to use any of the field's change event APIs directly, but for cases it doesn't handle, there are several methods available for working with change events:

`stopChangeEvents(opt_stopChange, opt_stopDelayedChange`
: Stops handling of change and/or delayed change and the dispatching of the corresponding events. Dispatches any pending events before events of that type are stopped. Should be called before you make changes that you do not want tracked.

`startChangeEvents(opt_fireChange, opt_fireDelayedChange)`
: Restarts both change and delayed change events. Additionally (and optionally), immediately handles changes and fires the appropriate events. Should be called after you are done making changes.

`clearDelayedChange()`
: Immediately causes delayed change to be processed, firing the event if pending. Does nothing if there is no pending delayed change.

`dispatchChange(opt_noDelay)`
: Equivalent to `startChangeEvents(true, opt_noDelay)`.

`dispatchBeforeChange()`
: Should be called immediately before changes are made, to signal to the editor that something is about to change. `dispatchChange` should be called immediately after the changes are made to complete the transaction.

`handleChange()`
: Notifies the editor that a change has occurred. If change events are stopped, this does nothing.

`manipulateDom(func, opt_preventDelayedChange, opt_handler)`
: Calls a function to manipulate the editable region. This is the preferred method, rather than manually starting and stopping change events, as it takes care of the starting and stopping automatically, and ensures you never forget to turn change events back on when you are done.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

`setModelMode/inModalMode`
: Sets/gets if the field is currently in modal interaction mode. This is normally used by dialogs. When in modal mode, no changes should be made to the editable DOM by any code other than that which enabled modal mode.

`setBaseZIndex/getBaseZIndex`
: Sets/gets the z-index of the field. Used to make sure additional UI renders on top of the editable field.

A few other useful getters for working with your field:

`getEditableDomHelper`
: Returns the `goog.dom.DomHelper` for the root editable node. Note that this will not be the same as the `goog.dom.DomHelper` for the original node you made editable, as the editable node is inside an `<iframe>`.

`getElement`
: Returns the root editable node, if one exists. If the field is not yet editable, this will return `null`. Otherwise, it points to the body element in the `<iframe>`.

`getOriginalElement`
: Returns the original node that was transformed into the editable node. Note that this node may no longer be in the document.

`getRange`
: Returns the `goog.dom.AbstractRange` (covered later in the section "goog.dom.AbstractRange" on page 281) for the user's current selection in the field. Will always return `null` if the field is not yet editable.

goog.editor.SeamlessField

Sometimes you will want your editable area to blend in more with your application than a `goog.editor.Field`. For these cases, you can use `goog.editor.SeamlessField`. `goog.editor.SeamlessField` is a subclass of `goog.editor.Field`, and switching to it from `goog.editor.Field` requires very few changes. You initialize it the same way, and it dispatches the same events and supports the same public API. The biggest differences are implementation details in the field itself and how the field matches the look and feel of your page. The editable region in `goog.editor.Field` is always implemented using a fixed size `<iframe>`, with an editable `<body>` element. Thus, `myField.usesIframe()` and `myField.isFixedHeight()` are always `true` for instances of `goog.editor.Field`. Because it is inside an `<iframe>`, events do not propagate from the editor to the host page, nor does it inherit styles from the page. `goog.editor.SeamlessField` on the other hand is usually implemented using an editable `<div>` and can be a fixed height or grow with its contents. In cases where it is not implemented with a `<div>` (due to `contentEditable` not existing or being too buggy for that platform), an `<iframe>` is used, but `<div>` behavior is simulated.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

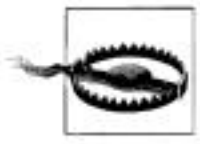
This is normally done in response to a click on a toolbar button or a keyboard shortcut. We'll cover hooking up a toolbar in "Toolbar" on page 274. When `execCommand` is called, the field looks at the installed plugins, and if there is a plugin that knows how to respond to the `goog.editor.Command.BOLD` command, the field allows the plugin to handle it. If there is no plugin registered that can handle the command, nothing will happen.

A complementary method exists to find out the current state of a command. This API was also named for a native browser API, `queryCommandValue(commands)`. `queryCommandValue` takes either a single command or an array of commands as a parameter, to avoid needing to call the method separately for every command, as multiple commands can be enabled at a given time:

```
// Assume the user's cursor is inside some bold, underlined text:
// <b><u>My text</u></b>.

// Returns true
myField.queryCommandValue(goog.editor.Command.BOLD);

// Returns true for bold and underline, but false for italic.
var result = myField.queryCommandValue([goog.editor.Command.BOLD,
    goog.editor.Command.ITALIC, goog.editor.Command.UNDERLINE]);
// Alerts Bold: true, Italic: false, Underline: true.
alert('Bold: ' + result[goog.editor.Command.BOLD] + ',' +
    'Italic: ' + result[goog.editor.Command.ITALIC] + ',' +
    'Underline: ' + result[goog.editor.Command.UNDERLINE]);
```

With `execCommand` and `queryCommandValue`, your application code calls the method on the field, the field checks if any plugins can handle the command, and if so, calls the corresponding method on the plugin.

Other calls into plugin code are less direct. For example, the field listens for key events, and when key events are received, the field also allows plugins to process the events, even though the plugins do not listen for the events, and your application code doesn't ask the field or plugin to handle the events. We'll cover these interactions in full detail next.



Although plugins have a fairly large public API, calls to plugins other than initialization should be done via the `goog.editor.Field`. For example, your application code should never call `execCommand` directly on a plugin instance, and should call it on the field instead. This allows the field to do any additional preparation or cleanup, and ensures consistency.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

styling, so if a user creates some bold text in Internet Explorer, just using the browser's `execCommand` would not unbold it in Firefox. For a single user, this situation does not come up all that often, as users do not usually use many different browsers, but for collaborative products, this becomes a big issue. `goog.editor.plugins.BasicTextFormatter` gets around this problem by rewriting the HTML to be a form that the particular browser understands, if necessary. Note that the plugin could instead implement the bold command itself, but—following the principle of leveraging the browser whenever possible because it will be faster and less code to let the browser handle the `execCommand`—it just does this one-time initial cleanup when the HTML is set. The relevant snippet of JavaScript is:

```
goog.editor.plugins.BasicTextFormatter.prototype.prepareContentsHtml =
    function(originalHtml, styles) {
  if (goog.editor.BrowserFeature.CONVERT_TO_B_AND_I_TAGS) {
    originalHtml = originalHtml.replace(/<(\/?)strong([^\w])/gi, '<$1b$2');
  }
  return originalHtml;
};
```

`cleanContentsHtml` is like the inverse of `prepareContentsHtml`. It takes in the string of `originalHtml` from the editable field, and returns a string of HTML that is suitable for storing server-side. `cleanContentsDom` is used for the same purposes, but provides a copy of the editable DOM rather than the HTML string. Changes should be made directly to the `fieldCopy` if using `cleanContentsDom`, as there is no return value. Plugins should use whichever is more efficient for their needs. Both are called by `goog.editor.Field` when the contents are fetched from the field via `myField.getCleanContents()`.

Unlike the other commands we have seen so far, the `prepareContentsHtml` and `cleanContents*` methods are reducing commands, so the results are passed from one plugin to the next. They are also called even for disabled plugins.

Advanced customization

- `goog.editor.Plugin.prototype.setAutoDispose(autoDispose)` and `goog.editor.Plugin.prototype.isAutoDispose()`
- `goog.editor.Plugin.prototype.activeOnUneditableFields()`

For most plugins, the default values of these should be sufficient, but if you are writing more advanced plugins, you may wish to set them. `setAutoDispose` sets whether the plugin is disposed when the field it is registered on is disposed, and defaults to `true`. `activeOnUneditableFields` returns whether the plugin should operate on uneditable fields as well as editable fields. The default is `goog.functions.FALSE`, and the plugin is automatically disabled when `makeUneditable` is called on the field.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

mostly using their native implementations, resulting in a `<br>` in Firefox and a `<div>` in all other browsers.

`goog.editor.plugins.TagOnEnterHandler` is a subclass of `goog.editor.plugins.EnterHandler` that ensures a block element is created when Enter is pressed. It is most commonly used with `<div>` or `<p>`, but technically any block element can be used. It also implements an additional piece of the plugin API: `queryCommandValue`. This is used by other plugins to determine which type of block they should create, if necessary, without having a direct dependence on the plugin. This plugin provides complete consistency for Enter behavior to all browsers, but is the slowest performing Enter handler in Firefox.

`goog.editor.plugins.Blockquote` is a plugin that does nothing on its own, and should be installed only alongside one of the Enter handlers. It handles the specific case of pressing Enter inside of a `<blockquote>` element, but implements only `execCommandInternal`, rather than any key handlers. Instead, when the user is inside of a `<blockquote>`, the Enter-handling plugin issues an `execCommand` for `goog.editor.plugins.Blockquote.SPLIT_COMMAND`, which invokes this plugin. This plugin is useful for applications that anticipate usage of `<blockquote>`, like mail applications, and can be left out for others. The plugin takes a boolean `requiresClassNameToSplit` and an optional string `opt_className` as parameters, which configure whether the plugin should operate on all blockquotes, or only a subset with a given class name. This is used to distinguish one type of blockquote from another, because `<blockquote>` elements can be used for other purposes, like regular indentation, and in those cases, should be split following normal block rules, not the special quoted handling.

goog.editor.plugins.LoremIpsum

Lorem ipsum text is dummy text used by the publishing industry as a placeholder for real text. This plugin serves the same purpose; it allows your application to provide some default text in the field before the user enters their real text, given as the `message` string in the constructor. The lorem ipsum text is shown only when the field is not focused, and as soon as the user focuses in the field, it is removed.

This plugin supports three commands: `goog.editor.Command.CLEAR_LOREM`, which removes the lorem text from the field, `goog.editor.Command.UPDATE_LOREM`, which determines whether lorem text is needed and if so, inserts it, and `goog.editor.Command.USING_LOREM`, which is used to determine whether lorem text is currently in the field. The first two are used with `execCommand` and the last is used with `queryCommandValue`, because it is querying state, rather than performing an action. This plugin is also active for uneditable fields, so lorem text can be used even when uneditable. One interesting thing to note about the lorem ipsum plugin is that it is not invoked by user actions. All calls to its `execCommand` and `queryCommand` are made directly by `goog.editor.Field`, resulting in this plugin having a tighter coupling with the field than most plugins. For example, this is necessary when getting the field's contents. If the field has not been edited by the user, you want your application to save the empty string, not

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
  var dom = this.fieldObject.getEditableDomHelper();
  var marqueePlaceholder = dom.createDom(goog.dom.TagName.DIV,
      example.MarqueePlugin.MARQUEE_CLASS_NAME);

  // Wrap the text with the placeholder.
  // Even though we are modifying the DOM here, we do not need to dispatch
  // any events, because the plugin base class handles that in execCommand.
  var range = this.fieldObject.getRange();
  marqueePlaceholder = range.surroundContents(marqueePlaceholder);
  goog.dom.Range.createFromNodeContents(marqueePlaceholder).select();
};
```

We also take advantage of the plugin API and implement only `handleKeyboardShortcut` rather than `handleKeyDown`, `handleKeyUp`, `handleKeyPress`, or directly listening to any keyboard events:

```
/** @inheritDoc */
example.MarqueePlugin.prototype.handleKeyboardShortcut =
    function(e, key, isModifierPressed) {
  // Also insert a marquee if the user hits ctrl + m.
  if (isModifierPressed && key == 'm') {
    this.fieldObject.execCommand(example.MarqueePlugin.COMMAND);
    return true;
  }
  return false;
};
```

To show the toolbar button in the depressed state if the user's selection is inside our special marquee text, we must implement `queryCommandValue` to return `true` for these cases:

```
/** @inheritDoc */
example.MarqueePlugin.prototype.queryCommandValue = function(command) {
  // Determine if the user is currently inside a marquee placeholder.
  // If so, return true.
  var range = this.fieldObject.getRange();
  var container = range && range.getContainer();
  var ancestor = goog.dom.getAncestorByTagNameAndClass(container,
      goog.dom.TagName.DIV, example.MarqueePlugin.MARQUEE_CLASS_NAME);

  return !!ancestor;
};
```

Finally, because we use the special placeholders inside the editor, to make sure the application saves using `<marquee>` tags, we will need to implement `cleanContentsDom` to transform our special `<div>`s to `<marquee>`s, and to make sure the editor loads with our special `<div>`s instead of `<marquee>`s we'll need to implement `prepareContentsHtml`:

```
/** @inheritDoc */
example.MarqueePlugin.prototype.prepareContentsHtml = function(originalHtml,
    styles) {
  // Replace the marquee with a div for use inside the editable region, because
  // editable regions in some browsers cannot support marquee tags.
  // Transforms <marquee class='scrolling-marquee'>scrolling text</marquee> ->
  // <div class='scrolling-marquee'>scrolling text</div>;
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

respectively, but you can customize the buttons using `addOkButton`, `addCancelButton`, and `addButton` to add any arbitrary button. If any buttons are added explicitly, the defaults will not be used.

Creating a custom dialog

In "Creating a custom dialog plugin" on page 268, we introduced a dialog plugin, `example.DialogPlugin`, that allowed the user to insert an image. The plugin handled the image insertion and interactions with the editable field, but created an instance of `example.Dialog` to show the dialog and collect input from the user. This section will demonstrate how to create custom dialogs, by implementing `example.Dialog`. The example dialog will look like Figure 9-5 and allow the user to enter the URL for the source of the image to insert. It will dispatch an event that contains the URL when the user clicks OK. As you saw in "goog.editor.plugins.AbstractDialogPlugin" on page 265, the dialog plugin inserts the image when it receives this event.

Figure 9-5. example.Dialog.

Like all editing dialogs, we'll start by subclassing `goog.ui.editor.AbstractDialog`:

```
/**
 * Creates a dialog for the user to enter the URL of an image to insert.
 * @param {goog.dom.DomHelper} dom DomHelper to be used to create the
 *     dialog's DOM structure.
 * @constructor
 * @extends {goog.ui.editor.AbstractDialog}
 */
example.Dialog = function(dom) {
  goog.ui.editor.AbstractDialog.call(this, dom);
};
goog.inherits(example.Dialog, goog.ui.editor.AbstractDialog);
```

To create the dialog itself, subclasses implement `createDialogControl`. You can create any type of dialog you want , but if you want an instance of a `goog.ui.Dialog`, you can use `goog.ui.editor.AbstractDialog.Builder` for convenience. We'll use this builder with the default OK and Cancel buttons. The only customization we need to do is to set the title and contents. Once customizations are complete, we call `build()` on the builder to create the dialog:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
var marqueeButton =
    goog.ui.editor.ToolbarFactory.makeToggleButton(
        example.MarqueePlugin.COMMAND, 'Insert marquee', 'Marquee');
// Mark it as queryable so the button state is automatically updated.
marqueeButton.queryable = true;
```

Styling the toolbar

If you have followed along and created a toolbar using the previous steps, you may notice that it doesn't actually look like the ones presented in the figures in this chapter. This is most likely because you are missing the necessary CSS files. The toolbar requires all of the following CSS to properly render the default buttons and menus:

```
<link rel="stylesheet" href="css/button.css" />
<link rel="stylesheet" href="css/menus.css" />
<link rel="stylesheet" href="css/toolbar.css" />
<link rel="stylesheet" href="css/colormenubutton.css" />
<link rel="stylesheet" href="css/palette.css" />
<link rel="stylesheet" href="css/colorpalette.css" />
<link rel="stylesheet" href="css/editortoolbar.css" />
```

The images used in the toolbar are all stored on Google's static content servers (`ssl.gstatic.com/editor/`). If you'd rather host the images yourself, you'll need to override the CSS that references the images. In each of the CSS files, find the references to `gstatic.com`, and add new rules that use your own paths instead:

```
/* Override the toolbar icon sprite */
.tr-icon {
  background-image: url(../path/to/your/version/editortoolbar.png);
}
```

Selections

In previous sections, we mentioned updating the toolbar state from the user's selection and we modified selections in the plugin examples. In this section, we'll dig deeper into what a selection is, how to extract information from it, how to modify it, and how to make changes to the DOM using it.

Users form selections when interacting with the application in many ways: with a mouse, with arrow keys, with keyboard shortcuts (such as Control-A for selecting all the contents in a region), or with actions under the Edit menu. There are two main types of selections. When a user selects something on a page such that the start and end points are different, this is known as an *expanded selection*. Browsers normally display expanded selections using a dark background and white font. The other type of selection is a *caret* (or a collapsed selection), which occurs when the start and end are identical (nothing is actually selected), and the browser displays this as a blinking cursor.

A *range* is any subsection of the DOM between two endpoints. Ranges are often used to change the user's selection by selecting a given range. You can also retrieve a range

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Besides the endpoints themselves, you may want to find the deepest node in the DOM that contains the entire range. For this, use `getContainer` (or `getContainerElement` to find the deepest element).

Setting endpoints of a range

To modify which nodes a range starts and ends in, use `moveToNodes(startNode, startOffset, endNode, endOffset, isReversed)`. For the simple case of collapsing, you can also use `collapse(toAnchor)` to change the range into a caret at either the focus or anchor position of the range. Remember that these affect only the range, not the selection.

To change the user's selection to equal any range, just create a range at the position you want, and call `select()` on that range object.

Extracting data from a range

Besides the endpoints, you will also often want to know what DOM contents are enclosed by the range. There are several different methods you can use to get the contents, depending the type of information you want. Each of these methods returns a string of HTML. `getText` is the simplest and returns the plain-text version of the range. `getHtmlFragment` returns a string of the exact HTML enclosed, but this can be invalid HTML. `getValidHtml` is like `getHtmlFragment`, but makes sure that all context nodes are included so the HTML is valid. Finally, `getPastableHtml` is like `getValidHtml`, but contains a fully complete DOM that can be pasted anywhere. To illustrate these differences, consider the HTML for a simple table:

```
<table>
  <tbody>
    <tr>
      <td id='td1'><b>cell</b>1</td>
      <td id='td2'>cell2</td>
    </tr>
    <tr>
      <td>cell3</td>
      <td>cell4</td>
    </tr>
  </tbody>
</table>
```

And a selection that selects all of the first and second cells:

```
// Select "cell1cell2".
range = goog.dom.Range.createFromNodes(goog.dom.$('td1'), 0,
    goog.dom.$('td2'), 1);
range.select();
```

The content getters each return different results:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
  // For text nodes, we just need to be sure all the text is selected.
  if (startNode.nodeType == goog.dom.NodeType.TEXT) {
    return startOffset == 0 && endOffset == startNode.length;
  } else {
    // For elements, we just need to make sure only one child is selected.
    return endOffset == startOffset + 1;
  }
}

/**
 * Determines whether the range fully selects a single node.
 * @param {goog.dom.AbstractRange} range The range.
 * @return {boolean} Whether the range fully selects a single node.
 */
var isSingleNodeFullySelected = function(range) {
  var startNode = range.getStartNode();
  var endNode = range.getEndNode();
  var startOffset = range.getStartOffset();
  var endOffset = range.getEndOffset();

  return isSingleNodeFullySelectedHelper_(startNode,
      startOffset, endNode, endOffset);
}
```

This function works fine if the range was created using the same node as both endpoints (either the text node, the bold node, or the bold's parent node), but what if the range was created with a mix, like:

```
var range = goog.dom.Range.createFromNodes($('b'), 0, $('b').firstChild, 4);
```

`example.isSingleNodeFullySelected` would incorrectly determine that more than one node is selected, because the range starts and ends in different nodes. To work around the inconsistencies caused by multiple DOM positions representing the same visual position, use `goog.editor.range.expand` to expand the range to contain all tags that are fully selected. The range returned is visually equivalent to the original, but the start and end will expand as far as possible. This allows us to write an improved version of `isSingleNodeFullySelected`:

```
var isSingleNodeFullySelected2 = function(range) {
  var expandedRange = goog.editor.range.expand(range);
  return isSingleNodeFullySelected(expandedRange);
}
```

This time, the start and end node will both be the `<div>`, and the start offset will be 1, and end offset will be 2, so `isSingleNodeFullySelected2` will properly return `true`.

`goog.editor.range.narrow` provides similar functionality to `goog.editor.range.expand`, except that it narrows the range to contain fewer tags rather than expanding to include more. However, the methods are not as equivalent as the names imply. Though `expand` works with the current range to expand it as far as possible (taking an optional stop node as a parameter), `narrow` requires an additional parameter, which is the element to narrow the range to. Specifically, `narrow` will move the endpoints in only as far as that element, even if a narrower range exists.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Once constructed, a `goog.debug.LogRecord` has getter and setter methods for each of its properties, such as timestamp and logging level. Although its setter methods will rarely be of interest, its getter methods will be used by user interfaces to display logging information.

goog.debug.Logger.Level

Not all logging messages are of equal importance, so a `goog.debug.Logger.Level` is associated with a log message to indicate its priority. For example, when debugging an email application, it is probably not important to get an informational message that a click has been registered on the "check for new mail" button, but it is important to get a warning message that the mail server is slow to respond. By logging these messages with the appropriate level, it makes it easier to configure a logger so that only the significant messages are displayed in a logging UI.

Note that the level does not dictate priority in terms of the order in which messages will be displayed—messages will always be displayed in the order in which they are received by the logger. Instead, the level determines whether a message is published by a logger, and it may also affect how the message is presented in a logging UI.

Each logging level corresponds to a number, and higher numbers correspond to more important messages. The logging package defines a number of levels to encourage the consistent use of levels across modules. As shown in Table 10-1, the predefined levels for `goog.debug.Logger.Level` match those defined by the Java class `java.util.logging.Level`, though Closure contains one additional level, `SHOUT`, which is higher than the highest Java level, `SEVERE`.

Table 10-1. Predefined levels for goog.debug.Logger.Level.

Name	Value	Meaning
`OFF`	Infinity	Special value that indicates logging should be disabled
`SHOUT`	1200	Extra debugging loudness
`SEVERE`	1000	Serious failure
`WARNING`	900	Potential problem
`INFO`	800	Informational message
`CONFIG`	700	Static configuration message
`FINE`	500	Tracing information
`FINER`	400	Fairly detailed tracing message
`FINEST`	300	Highly detailed tracing message
`ALL`	0	Special value that indicates that all messages should be logged

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

This behavior makes it possible to finely control which records appear in a logging UI. Consider the following logging configuration:

```
// Even though there is nothing like the following in base.js:
//   var logger = goog.debug.Logger.getLogger('goog');
// The 'goog' logger gets created as a side effect of creating a logger
// deeper in the hierarchy, such as 'goog.ui.ComboBox'.
// Even if there were no loggers created in under the 'goog' namespace,
// the following call would dynamically create such a logger.
goog.debug.Logger.getLogger('goog').setLevel(goog.debug.Logger.Level.OFF);

goog.debug.Logger.getLogger('goog.ui.ComboBox').
    setLevel(goog.debug.Logger.Level.INFO);

goog.debug.Logger.getLogger('example').setLevel(goog.debug.Logger.Level.ALL);
```

In this example, setting the level of the `'goog'` logger to `OFF` effectively disables all logging in the `goog` namespace because all of the loggers in `goog` are descendants of the `'goog'` logger. However, setting the level of the `'goog.ui.ComboBox'` logger to `INFO` will override the `OFF` setting for the `'goog.ui.ComboBox'` logger and all of its descendants. Meanwhile, all records dispatched by loggers that descend from the `'example'` logger will be published. This leads to the following behavior:

```
// These records will not be published because they inherit OFF from goog.
goog.debug.Logger.getLogger('goog.ui').
    severe('goog.ui severe');
goog.debug.Logger.getLogger('goog.ui.ComboBoxItem').
    shout('goog.ui.ComboBoxItem shout');

// These records will be published because they inherit INFO from
// goog.ui.ComboBox.
goog.debug.Logger.getLogger('goog.ui.ComboBox').
    warning('goog.ui.ComboBox warning');
goog.debug.Logger.getLogger('goog.ui.ComboBox.EventType').
    info('goog.ui.ComboBox.EventType info');

// These records will not be published because they are below INFO.
goog.debug.Logger.getLogger('goog.ui.ComboBox').
    fine('goog.ui.ComboBox fine');
goog.debug.Logger.getLogger('goog.ui.ComboBox').
    config('goog.ui.ComboBox config');

// These records will be published because they inherit ALL from example.
goog.debug.Logger.getLogger('example.Parent').
    finer('example.Parent finer');
goog.debug.Logger.getLogger('example.Child').
    finest('example.Child finest');
```

Recall that the debugging level is checked only once by the initial logger when a record passes through the logger hierarchy. This is what makes it possible for a record logged by the `'goog.ui.ComboBox'` logger to reach the root logger even though the level of the intermediate `'goog'` logger is `OFF`. This is done deliberately to enable a specific logger

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CHAPTER 11

Closure Templates

In a web application, it is extremely common to dynamically build up a string of HTML and then insert it into the page. Most web frameworks provide a server-side template language, such as PHP or JSP, to address this issue. By comparison, Closure Templates (also known as "Soy") provides a templating solution that can be used on both the server and the client from either JavaScript or Java. Soy generates JavaScript code from a template at build time, so it avoids the runtime parsing cost encountered by most JavaScript template systems. Further, because the code it generates for each template is a JavaScript function, it can be called or passed around as a functor in a way that is natural for JavaScript programmers. This chapter will explore the many features and benefits of using Soy.

Limitations of Existing Template Systems

Before diving into why Soy works the way it does, it is important to consider the problems that Soy is trying to address in other template systems that are available today.

Server-Side Templates

When using a template system that only works on the server, such as JSP or PHP, a template often resembles an entire HTML file with placeholders for variable text. In this way, the user of the template can insert values for the template variables at runtime to dynamically create a string of HTML that can be sent down to the client. The following is an example of a PHP file with a placeholder to insert the current year:

```
<html>
<body>
  <h3>The year is <span style="color:red"><?= date('Y') ?></span></h3>
</body>
</html>
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
 * @param rows A list of objects with various properties.
 *             The first row should have a property named units, subsequent rows
 *             should have properties named symbol and value, and the final row
 *             should have a property named total.
 * @param? includeSampleCss true to include sample CSS with the template
 * @param oddRowBgColor css color for odd rows, such as 'gray' or '#CCC'
 * @param? evenRowBgColor css color for even rows, defaults to '#FFF'
 */
{template .portfolio}
// Only include the CSS if it is requested.
<style>
  {if $includeSampleCss}
    {call .portfolioCss /}
    {call .portfolioColors}
      {param odd: $oddRowBgColor /}
      {param even}
        {if $evenRowBgColor}
          {$evenRowBgColor}
        {else}
          #FFF
        {/if}
      {/param}
    {/call}
  {/if}
</style>

<div class="{css portfolio}">
{foreach $row in $rows}
  {if isFirst($row)}
    <div class="{css row}">
      <div class="{css cell}">Symbol</div>
      <div class="{css cell} {css units-cell}">{$row.units}</div>
    </div>
  {elseif not isLast($row)}
    <div id="{$row.symbol|id}"
         class="{css row}{sp}
         {if index($row) % 2 == 0}{css even-row}{else}{css odd-row}{/if}">
      // This assumes that the caller has made sure that the ticker symbol does
      // not contain any malicious HTML.
      {call .symbolRow data="$row" /}
    </div>
  {else}
    <div class="{css row}">
      <div class="{css cell}">Total ({length($rows) - 2}):</div>
      <div class="{css cell} {css value-cell}">{print $row.total}</div>
    </div>
  {/if}
{ifempty}
  <span style="color: red">Error: data was empty!</span>
{/foreach}
</div>
{/template}
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
length($rows) - 2 == 0

// Evaluates to a random item in the list $rows
$rows[randomInt(length($rows))]
```

If there is a value that you want to display in your template, but it cannot be created using the Soy expression language, then you must either compute the value before passing it to the template, or extend the template language as explained in "Defining a Custom Function" on page 328. For example, it is not possible to define an array literal in Soy, but it is possible to pass one in as Soy data.

Referencing map and list data

In addition to literal values, template data may also be composed of lists and maps. In JavaScript, an array can be used as a Soy list and an object can be used as a Soy map. In Java, `java.util.List` and `java.util.Map` can be used for the respective template data types. (Soy also provides its own Java classes, `SoyListData` and `SoyMapData`, which can be used for template data as well.)

Referencing the data contained in these data structures from a Soy template uses the same syntax as JavaScript for referencing properties:

```
$users[2].password      // $users is a list. Its third item must be
                        // a map with a property named "password".

$users.2.password       // Although this would not be valid JavaScript,
                        // it is allowed in Soy and is equivalent to the
                        // above example.

$users[2]['password']   // This will work as well, though the generated
                        // JavaScript will quote the password property,
                        // which may be a problem when using the Advanced
                        // mode of the Closure Compiler, as explained later
                        // in this chapter.
```

Of the three examples, the `$users[2].password` style is used most often.

Referencing global variables

A name without a preceding dollar sign is treated as a reference to a global variable in a Soy template:

```
{Math.PI}
{example.CardinalDirection.WEST}
```

That means that forgetting a dollar sign when referencing a parameter will not emit a compile-time error when compiling a Soy template for JavaScript:

```
{evenRowBgColor}   // No error even though this should be {$evenRowBgColor}.
```

However, it will produce a runtime error when the template is used if the global variable cannot be resolved.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
{template .formletter}
  {call .formletterHelper}
    {param name}
      {$title} {$surname}{if $suffix}, {$suffix}{/if}
    {/param}
  {/call}
{/template}

/**
 * @param name
 */
{template .formletterHelper}
<div>
  Dear {$name}:<br>
  <p>
  With a name like {$name}, shouldn't
  you have your own theme song? We can help!
  </p>
</div>
{/template}
```

Identifying CSS Classes with {css}

One command that may come across as odd is the `{css}` command, as it appears to simply print a value like the `{print}` command. Indeed, that is its default behavior, though this behavior can be overridden to rewrite the representation of a CSS class. For example, when using a Soy template to generate JavaScript that is compatible with the Closure Library, it will wrap the argument in a call to `goog.getCssName()` when the following flag to `SoyToJsSrcCompiler.jar` is used:

```
--cssHandlingScheme GOOG
```

The default behavior of the `{css}` command is to print the value without HTML escaping, so it behaves like the `id` print directive. The behavior of the `{css}` command can be configured at compile time.

Internationalization (i18n)

Internationalization (also known by its much shorter numeronym, i18n) is the process of designing a software application so that it can be made available in another language (written language, not programming language!). For many web applications, i18n means creating a process that allows translated strings (in the user interface) to be substituted with their English equivalents. Because the means by which translations are acquired may differ from project to project, the mechanics of how i18n is done may vary, but the principle of substituting translated strings is always the same.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Another option is to use Soy to create a document fragment that can be appended to the DOM at a later point in time:

```
goog.require('example.formletter.templates');
goog.require('soy');

var fragment = soy.renderAsFragment(example.formletter.templates.formletter,
                                    data);
```

Therefore, to build up the content of your own `goog.ui.Component` using Soy, override `createDom()` as follows (note that this requires the result of `soy.renderAsFragment()` to be an `Element`, not just a `Node` or `DocumentFragment`):

```
/** @inheritDoc */
example.MyComponent.prototype.createDom = function() {
  var element = soy.renderAsFragment(example.formletter.templates.formletter,
                                     data);
  this.setElementInternal(/** @type {Element} */ (element));
};
```

Because the DOM of the component is determined by a Soy template, it makes it easier to take advantage of decoration because server code written in Java can use the Soy template to produce HTML that `example.MyComponent` can decorate.

Compiling a Template for Java

This section explains how to use a Soy template from Java.

Compilation

Unlike `SoyToJsSrcCompiler.jar`, which generates a JavaScript file from the template that can be compiled later by the Closure Compiler, no Java source code is generated when compiling a Soy template for Java. Instead, a Soy template is used to create an in-memory Java object called a `SoyTofu`. The `SoyTofu` represents the template and can be populated with values in order to generate the corresponding HTML.

There is experimental code for generating Java source code from a Soy template in `com.google.template.soy.SoyToJavaSrcCompilerExperimental`. Unfortunately, it is not being actively developed, and there are no plans to develop it further, so creating a `SoyTofu` object is the only option for using Soy from Java code at this time.

Because Soy compilation is done programmatically in Java rather than using a command-line tool, you need to add `soy.jar` to your Java classpath in order to write Java code that uses the Soy API. The "lite" version of the Javadoc for the Soy API is available at *http://closure-templates.googlecode.com/svn/trunk/javadoc-lite/index.html*, which documents all of the classes that you will need in order to work with Soy templates in Java.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
# Use this script on Mac or Linux.
java -classpath SoyToJsSrcCompiler.jar:build/classes \
     com.google.template.soy.SoyToJsSrcCompiler \
     --outputPathFormat build/{INPUT_FILE_NAME_NO_EXT}.soy.js \
     --pluginModules example.ExampleModule \
     phone.soy
```

This will make it possible to compile the following Soy template, `phone.soy`:

```
{namespace example.phone.templates}

/**
 * @param phoneNumber
 */
{template .phone}
({substring($phoneNumber, 0, 3)}){sp}
 {substring($phoneNumber, 3, 6)}-
 {substring($phoneNumber, 6)}
{/template}
```

Now `example.phone.templates.phone` can be used in place of `example.formatPhone Number`. If `ExampleModule` is to be used to compile a Soy template for Java, it can be done by using Guice directly:

```
// Use Guice to create a SoyFileSet.Builder
Injector injector = Guice.createInjector(new SoyModule(), new ExampleModule());
SoyFileSet.Builder builder = injector.getInstance(SoyFileSet.Builder.class);

// Once the builder is created, the Java code is the same as it was before.
builder.add(new File("phone.soy"));
SoyTofu soyTofu = builder.build().compileToJavaObj();
SoyMapData data = new SoyMapData("phoneNumber", "2018675309");
final SoyMsgBundle msgBundle = null;
System.out.println(soyTofu.render("example.phone.templates.phone",
    data, msgBundle));
```

The plugin system also supports creating your own print directives in a similar manner.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Using the Compiler to track down these errors automatically can save lots of time that would normally be spent debugging JavaScript code. Further, developers will have more confidence in the correctness of their code when it compiles without any errors or warnings. This does not provide the same assurances that manual or unit testing do, but the guarantees that are provided by the Compiler can be achieved with far less effort. The Compiler is not a substitute for testing, but it provides some level of verification even when no tests are present.

Protecting Code Through Obfuscation

Unfortunately, unscrupulous developers have been known to try to clone existing websites in order to steal their business. Although the open nature of the Web may make it easy to learn how to write HTML and how to use CSS, it also makes it easy to copy the design of an existing site (which may have cost a lot of money to produce) and use it for your own purposes. Before there were any JavaScript minification tools, the same was true of the JavaScript code on the Web—most sites would include their JavaScript code exactly as it was originally written for any visitor of the site to see. In the earlier days of the Web, when JavaScript was used sparingly to provide minor cosmetic enhancements to web pages, reusing such JavaScript code from other sites was fairly innocuous. But now that web applications have become more sophisticated and the number of lines of JavaScript code exceeds the number of lines of server code, "borrowing" kilobytes of JavaScript from a popular website cannot be taken lightly.

Using the Closure Compiler to obfuscate JavaScript source code does not prevent malicious users from downloading your code and reverse-engineering it to figure out what it does and then use it, but it certainly makes it a lot harder. By using the Closure Compiler to rename functions and variables, the output from the Compiler is unintelligible to the casual developer. Some of the compiler optimizations discussed in the next chapter, such as inlining, will change the structure of your program in such a way that makes it much harder to map back to its original form. This means that even for those who figure out how to make the calls into your obfuscated code to get it to do what they want, it will still be difficult for them to deobfuscate it into the form of the original source code. By denying access to the code's original structure, it makes it harder for an outsider to modify and maintain the code for his own purposes.

It is also important to remember that source code often contains information that is not meant for public consumption, such as developer email addresses or references to unreleased products or features. Removing comments from the source code before deploying it publicly can help reduce embarrassing information leaks.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Though if you are still determined to use conditional comments, it is possible to use `eval()` to ensure that the comment is preserved and evaluated:

```
var isOldScriptEngine = eval('false /*@cc_on || (@_jscript_version < 5) @*/');
if (isOldScriptEngine) {
  document.write('You need a more recent script engine.<br>');
}
```

Simple

The default compilation mode is Simple. The goal of Simple mode is to tolerate any valid JavaScript code as input and to produce minified output by using techniques such as simplifying expressions and renaming local variables within functions. It also performs special optimizations for some of the functions in the Closure Library, as explained in "Processing Closure Primitives" on page 417.

When using Simple mode on the `foo` function from the previous example, the output is even smaller:

```
var foo=function(a){return a==3?"\"'":1E4};
```

Like Whitespace Only mode, the behavior of the code is unchanged, but some additional changes have been made to the input that make the output even smaller than before:

- The local variable `bar` has been renamed to `a` to save bytes.
- The `if` and `else` statements have been replaced with the `?:` ternary operator to save bytes.
- The `('baz').length` expression has been evaluated at compile time and replaced with the result: `3`.
- The string literals in the `if` clause have been concatenated together at compile time.

Because local variables are renamed in Simple mode, a common technique for getting additional compression is to declare functions inside an anonymous function that is called immediately:

```
var loggingClickHandler;

(function() {
  // Functions declared inside an anonymous function.
  var recordClick = function(id) { console.log(id + ' was clicked'); };
  var clickHandler = function(e) { recordClick(e.target.id); };

  // This makes clickHandler available externally by assigning to a variable
  // in the global environment.
  loggingClickHandler = clickHandler;
})();
```

In this way, `recordClick` and `clickHandler` are simply local variables that can be renamed using Simple mode. Because variables that are declared inside the anonymous function are not available externally by default, they need to be "exported" by assigning

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Print input delimiter

When compiling multiple files into a single JavaScript file, it can be helpful to delimit file boundaries so that it is possible to determine which sections of the compiled code correspond to a particular input file. When the "print input delimiter" option is enabled, the Compiler will insert a comment of the form `Input X` at the start of each file boundary where `X` is a number starting with zero:

```
// Input 0
alert("I am a statement from the first input file");
// Input 1
alert("I am a statement from the second input file");
```

One common use of "print input delimiter" is to split the output into chunks to determine which input files are contributing the most code to the compiled output. Because the Compiler has the ability to remove unused functions in Advanced mode, the largest input file may not contribute much code to the output if it contains many unused library functions.

Because the input delimiter is simply a number, it must be mapped back to the list of input files to the Compiler to determine the file that corresponds to the delimiter.

Warning Levels

When compiling code, the Compiler will issue warnings for code that it believes to contain mistakes. Warnings are different from errors, which identify code the Compiler knows to be problematic, such as code that will not parse. (Google maintains a comprehensive list of Compiler warnings and errors at *http://code.google.com/closure/compiler/docs/error-ref.html*.) The degree to which the Compiler will criticize your code is determined by the warning level.

Quiet

When Quiet is used, the Compiler will not issue any warnings, but it may still yield errors that would prevent the input code from running at all, such as syntax errors.

Default

The default warning level is simply named "Default." It has warnings for common coding errors, such as an early `return` statement that results in unreachable code:

```
var strengthenYourBones = function() {
  takeMilkOutOfRefrigerator();
  pourAGlassOfMilk();
  return true;
  putMilkAway(); // Oh no, the milk will spoil because this is unreachable!
};
```

Warnings in Default mode apply to JavaScript code that is written in any style, not just the annotated style of the Closure Library.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
java -jar compiler.jar --js a.js --js b.js --js c.js \
                       --compilation_level=ADVANCED_OPTIMIZATIONS \
                       --warning_level=VERBOSE \
                       --formatting=PRETTY_PRINT \
                       --formatting=PRINT_INPUT_DELIMITER \
                       --output_file d.js
```

The Closure Compiler Application has some important options beyond what is available to the Closure Compiler Service.

Fine-grained control over warnings and errors

Many of the warnings issued by the Compiler can be suppressed or promoted using command-line flags. Such warnings are listed on the wiki for the Closure Compiler project at *http://code.google.com/p/closure-compiler/wiki/Warnings*. The messaging of a warning can have one of three states:

`OFF`
: This suppresses the warning: nothing will printed to the console when the Compiler encounters code that would normally trigger the warning.

`WARNING`
: The warning message will be printed to the console, but it will not disrupt compilation.

`ERROR`
: The warning message will be printed to the console, and no compiled JavaScript code will be generated.

Each level has a corresponding flag (`--jscomp_off`, `--jscomp_warning`, `--jscomp_error`) that takes the name of the warning as an argument. Because each warning can be configured independently, each flag can be used multiple times:

```
java -jar compiler.jar --js a.js --js b.js --js c.js --output_file d.js \
                       --jscomp_off=checkTypes \
                       --jscomp_off=missingProperties \
                       --jscomp_warning=deprecated \
                       --jscomp_warning=visibility \
                       --jscomp_error=unknownDefines \
                       --jscomp_error=fileoverviewTags
```

Using these flags is particularly helpful when working on an existing JavaScript codebase to make it Compiler-compliant. For example, it may be easier to start out by enabling only a few warnings until all of the problems identified by the warning are fixed. After that point, it makes sense to treat warnings as errors by using the `--jscomp_error` flag instead to ensure that the problematic code does not reenter the codebase. Developers on your team may find warnings easy to ignore, so using errors to prevent compilation will ensure that problems get fixed. For new projects, it is recommended to use `--jscomp_error` for all available warnings from the outset so that problems are caught as early as possible.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

frequent operation when developing using Closure Tools, it is important to be able to perform the same compilation operation over and over with much less typing. A script that automates this type of work is known as a *build script*. By providing a common script that all team members are expected to use create and deploy software, this establishes a *build process*. Having an automated build process introduces consistency and eliminates errors.

The thought of having to create a build process for JavaScript may be foreign to most web developers. Historically, JavaScript files were treated as static resources that could simply be dumped into a folder and served, much like stylesheets and images. Now that the JavaScript files that you will serve to your users will be generated using the Compiler, it is necessary to have a script that can produce those files automatically. Furthermore, because the Compiler can find errors in your code, it is important to run the Compiler regularly in order to find bugs as early as possible in the development process. The best way to do this is to make the Compiler part of your build process to ensure that it gets run.

It is important that a build process be fast so that developers do not spend a lot of time waiting for their code to build. In the case of the Compiler, that means using the Closure Compiler Application rather than the Closure Compiler Service API because the network latency incurred by the Service API will make the build process too slow. Even worse, if the build process depends on the Service API and Google App Engine is temporarily unavailable (which has been known to happen: *http://www.techcrunch.com/2008/06/17/google-app-engine-goes-down-and-stays-down/*), it will not be possible to build at all during that time.

For projects that use a batch or shell script for an existing build process, integrating the Compiler is simply a matter of copying the appropriate commands from the examples of running the Closure Compiler Application from the previous section. Those using Make as their build tool of choice can do the same.

If you find it hard to keep track of the various shell scripts that are required to use Closure, you may want to consider using plovr, a build tool designed specifically for projects that use Closure. You can find information on plovr in Appendix C.

A slightly more complicated, but not uncommon, setup is to use Apache Ant (*http://ant.apache.org*) for the build process. As a mature cross-platform build tool, Ant is integrated with a number of IDEs (such as Eclipse) and has comprehensive documentation online that is rife with examples. Indeed, both the Closure Compiler and Closure Templates use Ant scripts to manage their respective build processes. As an example, create the following file named `build.xml` in the `hello-world` directory from Chapter 1:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The `templates` target builds `hello.soy.js` by leveraging the `generate-template` macro, and the `hello-compiled` target builds `hello-compiled.js` by leveraging the `closure-compile` macro. The `build` target is simply an alias for the `hello-compiled` target. Because multiple targets can be passed to `ant`, it is common to do a clean build by running `ant clean build`, which is the same as running `ant clean` followed by `ant build`.

At 80 lines, this may seem like a lot of code just to compile one JavaScript file; however, this contains a lot of reusable material, so creating more compilation targets in the future will be much easier. For example, this is a build target for `hello-compiled-for-firefox-on-windows.js` from Chapter 1:

```
<target name="hello-compiled-for-ff-on-win"
        depends="templates"
        description="generates hello-compiled-for-firefox-on-windows.js">
  <mkdir dir="${build.dir}" />
  <closure-compile inputfile="hello.js"
                   outputfile="${build.dir}/hello-compiled-for-firefox-on-windows.js"
                   compilationlevel="ADVANCED_OPTIMIZATIONS">
    <extrapaths>
      <arg line='-p "${build.dir}/templates"' />
    </extrapaths>
    <extraflags>
      <arg line='-f "--define=goog.userAgent.ASSUME_GECKO=true"' />
      <arg line='-f "--define=goog.userAgent.ASSUME_WINDOWS=true"' />
      <arg line='-f "--define=goog.userAgent.jscript.ASSUME_NO_JSCRIPT=true"' />
    </extraflags>
  </closure-compile>
</target>
```

Most of the common flags passed to `calcdeps.py` are encapsulated in the `closure-compile` macro, so they do not need to be redeclared in the `hello-compiled-for-firefox-on-windows` target. Now that there are two major deliverables in this build script, it may be worth redefining the `build` target to include both of them:

```
<target name="build" depends="hello-compiled, hello-compiled-for-ff-on-win" />
```

Although implementing `build.xml` may initially seem like a substantial cost, the time saved and errors prevented by being able to type `ant` instead of messing with a multiline shell command will quickly make the investment in Ant pay for itself.

Partitioning Compiled Code into Modules

One feature of the Compiler that has not been mentioned thus far is its ability to partition its compiled output into separate files rather than one large file. Each partition is known as a *module*, and the Closure Library has code to help with loading modules dynamically at runtime. Dividing code into modules is attractive because it enables a web application to download only the JavaScript it needs for the initial page load so that it renders quickly. As the user navigates to the other parts of the application, the modules that contain the code to support such features can be loaded on demand.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
goog.exportSymbol('example.api.load', function(callback) {
  moduleManager.execOnLoad('api', callback);
});

goog.exportSymbol('example.api.isLoaded', function() {
  var moduleInfo = moduleManager.getModuleInfo('api');
  return moduleInfo ? moduleInfo.isLoaded() : false;
});
```

The first thing it does is define the button click handler for `example.App` so that it dynamically loads the settings module. Once it sets the button click handler, it initializes the singleton instance of `example.App` via `example.App.install()`. This is done before the subsequent configuration so that the user sees the UI for the main application as soon as possible.

Once `example.App` is displayed, `app_init.js` configures `goog.module.ModuleManager`. It loads the information for the dependency graph via some global variables that will be defined later. (By keeping the graph information in a separate file, it is easier to dynamically generate it as part of a build process.) Once the graph has been loaded, the main module notifies the manager that it has been loaded via its `setLoaded()` method.

Finally, it exports a public function named `example.api.load()` that will dynamically load the user script API. It takes a callback function so that a client of the API will be notified once the user script API is available. Similarly, it also exports a public function named `example.api.isLoaded()` that can be used to test whether the API has already been loaded.

settings_init.js

When the settings module is loaded, it includes a file named `settings_init.js` in addition to `settings.js`:

```
goog.require('example.App');
goog.require('example.Settings');
goog.require('goog.module.ModuleManager');

var app = example.App.getInstance();
var settings = new example.Settings(app.getDomHelper());
app.addChild(settings, true /* opt_render */);

// This tells the module manager that the 'settings' module has been loaded;
// otherwise, the module manager will assume that loading has timed out and it
// will try again.
goog.module.ModuleManager.getInstance().setLoaded('settings');
```

The initialization logic in `settings_init.js` assumes that the `example.App` singleton has already been created and that the settings module has been loaded in order to add `example.Settings` as a child of `example.App`. Once the settings UI is displayed, `settings_init.js` notifies the module manager that it has been loaded, so any pending callbacks that were waiting for the module to load will now be called. In this case, the

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Now that this system is in place, it is fairly easy to iterate on the various modules to make them fully featured. Unfortunately, as the list of dependencies for the application changes, it will require updates to the compilation script because the partitioning is done manually. See Appendix C to learn about plovr, a Closure build tool that can help with automating this process.

Refining the Partitioning

One unfortunate aspect of this example is that the initial module that is loaded contains 60 files, but 30 of those files are used to support `goog.module.ModuleManager`, so they have nothing to do with displaying `example.App`. The extra code means that the `app` module takes longer to parse, which means it takes longer to display the UI. One solution is to divide the initial module into two parts: `app_core` and `app`. The former would be responsible for loading `example.App`, and the latter would be responsible for loading `goog.module.ModuleManager`. By loading `app_core` immediately and delay-loading `app`, the UI will be displayed sooner.

Fortunately, this can be achieved with a slight change to the `--module` arguments to the Compiler:

```
--module app_core:30 \
--module app:30:app_core \
--module api:2:app \
--module settings:2:app \
```

By adding the following line to `app.js`:

```
goog.exportSymbol('example.App.install', example.App.install);
```

The application could now be loaded as follows:

```
<!doctype html>
<html>
<head>
  <title>Test page for refined module loading</title>
</head>
<body>

<div id="content"></div>

<script src="build/module_app_core.js"></script>
<script>
// Draw the UI.
example.App.install('content');
</script>
<script src="moduleinfo.js"></script>
<script>
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

What Happens During Compilation

The principal optimizations that are performed in Advanced mode that are not performed in Simple mode are full variable renaming and dead code elimination. In Simple mode, only local variables to a function are renamed, but in Advanced mode, all variables are renamed. Similarly, in Simple mode, all functions in the input will also appear in the output, but in Advanced mode, only the functions that could be executed during the lifetime of the application are preserved in the output. In order for Advanced mode to perform these types of optimizations, it must build a complete dependency graph of the application. That is, for each variable in the program, the Compiler must determine what other variables in the program it depends on. This is what makes it possible for the Compiler to determine which functions are called in the program, and ensures that all calls to a function are renamed consistently and uniquely in the application. This process is best illustrated with an example. First, consider the following JavaScript code:

```
var COST_PER_VALUE_MENU_ITEM = 0.99;
var SALES_TAX = .05;
var DECATHLON_COST = 10 * (1 + SALES_TAX) * COST_PER_VALUE_MENU_ITEM;

var a = function()  { return 1; };
var b = function(f) { f(); return 42; };
var c = function()  { return b(a) * SALES_TAX; };
var d = function(n) { return (n > 0) ? a() : n * d(n - 1); };
var e = function()  { return [b, c].length; };
e();
```

The dependency graph is computed as follows: for each variable declaration, create a new node in the graph. For each variable used in the definition of the declared variable, draw an arrow from the new node to node that represents the variable used in the definition. Note that for recursive functions, such as `d`, a node is able to point to itself. If the definition is an ordinary statement, such as `10 * (1 + SALES_TAX) * COST_PER_VALUE_MENU_ITEM`, then the dependencies are simply the variables used in the statement: `SALES_TAX`, `COST_PER_VALUE_MENU_ITEM`. But if the definition is a function, such as `function() { return a(b) * DECATHLON_COST; }`, then the dependencies are the union of the dependencies for all of the statements within the function: `a`, `b`, `DECATHLON_COST`. For top-level statements that are not variable declarations, such as `e()`, draw an arrow from a special node named `*global*` to all the dependencies in the statement: `e()`. The dependencies of the previous code are illustrated in the graph in Figure 13-1.

Note how there are no arrows emitted from `b` even though `b` includes a function call: `f()`. This is because `f` is not the name of a function but the name of a variable in `b` that refers to a function. Because `f` is passed in to `b` (`f` is also known as a *free variable* of `b`), `b` does not depend on `f`. Instead, the function that passes `f` to `b` depends on `f`. In this case, `c` calls `b` with `a` as the value of `f`, so `c` depends on both `a` and `b`, but neither `a` nor `b` depend on one another.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The declaration for `parseInt()` is particularly interesting because the second argument, `base`, is optional in practice but is declared as `{number}` rather than `{number=}`. As the comment states, this is done intentionally to avoid the following programming error:

```
/**
 * @param {string} timeString in HH:MM format.
 * @return {{hour:number, minutes:number}}
 */
var parseTime = function(timeString) {
  var parts = timeString.split(':');
  return { hour: parseInt(parts[0]), minutes:parseInt(parts[1]) };
};

// returns { hour : 10, minutes : 0 } instead of { hour: 10, minutes: 9 }
parseTime('10:09');
```

The value of `minutes` is `0` instead of `9` because `parseInt('09')` interprets `09` as an octal value rather than a decimal value because of the leading `0`. The way to fix this is to specify the appropriate value for `base` when using `parseInt()`. Using `10` instructs `parseInt()` to treat `num` as a decimal number, so `parseInt('09', 10)` returns `9` as expected. By making `base` a required argument in the extern definition of `parseInt()`, the Compiler will produce a warning if `parseInt()` is called with only one argument, assuming that type checking is enabled in the Compiler. Editing the definition of `parseTime()` so that `parseInt()` specifies a `base` will eliminate the warning and avoid a subtle programming error.

Those who find this behavior annoying may wish to define their own function for parsing integers and use that instead:

```
/**
 * @param {*} num
 * @return {number}
 */
var myParseInt = function(num) { return parseInt(num, 10) };
```

This has the advantage that `myParseInt()` can be renamed by the Compiler, whereas `parseInt()` cannot because it is an extern (unless `aliasExternals` is used, as explained in "aliasExternals" on page 441). An alternative solution would be to edit `es3.js` and create your own version of `compiler.jar` with the modified externs file, but that is considerably more work.

The file that contains the extern definition for `parseInt()` is named `es3.js` because the externs it declares correspond to the functions that must be provided in any JavaScript environment according to the third edition of the ECMAScript specification. Many of the externs files correspond to types defined by a public specification, such as `w3c_event.js` and `w3c_dom1.js`, whereas others correspond to proprietary, device-specific APIs, such as `iphone.js`.

The types of externs are able to refer to one another. The following snippets from several externs files demonstrate how the familiar `window`, `top`, and `document` variables are defined as externs:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

property disambiguation, which makes it possible to distinguish `Math.max` from `example.NumSet.prototype.max`. This logic makes it possible for the Compiler to rename `example.NumSet.max` without renaming `Math.max`. But because it is an experimental feature of the Compiler, it is not exposed via a command-line flag, so it must be enabled by using the Compiler's Java API, which is explained in the next chapter.

One final important aspect of extern functions is that they are assumed to have a side effect unless they are annotated with `@nosideeffects` in their JSDoc. This is significant with respect to dead code elimination because the Compiler cannot remove code that has a side effect without changing the behavior of the program. In `es3.js`, `Array.prototype.indexOf()` has the `@nosideeffects` annotation, but `Array.prototype.unshift()` does not. Because of this, the compiled version of the following code snippet:

```
[].indexOf(7);
[].unshift(11);
```

is:

```
[].unshift(11);
```

Because `Array.prototype.indexOf()` does not have a side effect, the Compiler can safely remove it without changing the behavior of the code, but the same is not true for `Array.prototype.unshift()`, so it remains part of the compiled output even though the array is never used.

Externs versus exports

Because externs do not get renamed, it is a common mistake to use externs as a mechanism to prevent variable or property renaming. (The Compiler will still remove unused code, even if it is declared as an extern.) Making a value available via a particular variable name after compilation is known as *exporting* a variable or property, and should be done by using either the `goog.exportSymbol()` or `goog.exportProperty()` function in the Closure Library.

For example, suppose you have the following JavaScript code:

```
goog.provide('example');

example.generateStandardsModeHtmlDoc = function() {
  return '<!doctype html>' +
    '<html>' +
    '<head><title>This page is in standards mode!</title></head>' +
    '<body></body>' +
    '</html>';
};
```

that is meant to be used with the following HTML:

```
<iframe src="javascript:parent.example.generateStandardsModeHtmlDoc()"></iframe>
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
example.Chocolate.prototype.hasGoldenTicket_;

/** @return {boolean} */
example.Chocolate.prototype.hasGoldenTicket = function() {
  return this.hasGoldenTicket_;
};

/**
 * @return {string}
 * @protected
 */
example.Chocolate.prototype.getSecretIngredient = function() {
  return 'anchovies';
};


/** @constructor */
example.ChocolateFactory = function() {};

/** @return {example.Chocolate} */
example.ChocolateFactory.prototype.createChocolate = function() {
  var hasGoldenTicket = Math.random() < .0001;
  var chocolate = new example.Chocolate(hasGoldenTicket);
  this.recordChocolate(chocolate.getSecretIngredient());
  return chocolate;
};

/**
 * @param {string} ingredient
 * @protected
 */
example.ChocolateFactory.prototype.recordChocolate = function(ingredient) {
  window.console.log('Created chocolate made with: ' + ingredient);
};
```

Even though the constructor for `example.Chocolate` is marked `@private`, an `example.Chocolate` can still be instantiated by `example.ChocolateFactory`. Although `example.ChocolateFactory` is a different class, it is defined in the same file as `example.Chocolate`, so access is allowed. This is also what makes it possible for `example.ChocolateFactory` to access `getSecretIngredient` even though it is not a subclass of `example.Chocolate`.

The second file is named `slugworthchocolatefactory.js`, and it tries to access information that the visibility modifiers in `chocolate.js` are trying to protect:

```
goog.provide('example.SlugworthChocolateFactory');

goog.require('example.Chocolate');
goog.require('example.ChocolateFactory');

// Try to create a Chocolate with a golden ticket.
var choc1 = new example.Chocolate(true /* hasGoldenTicket */);
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

CHAPTER 14

Inside the Compiler

As demonstrated in the previous chapter, the Closure Compiler is a tremendous tool, but it turns out that it has even more to offer if you are willing to take a peek at the source code. For starters, there are additional optimizations and checks that you can enable if you use the Compiler via its programmatic API, as explained in the section "Hidden Options" on page 436. Then in "Example: Extending CommandLineRunner" on page 450, you will learn how to create your own Compiler executable that will run exactly the compiler passes that you want. Finally, you will explore the Compiler's internal representation of a JavaScript program in "Example: Visualizing the AST Using DOT" on page 452, before creating your own compiler passes in "Example: Creating a Compiler Check" on page 456 and "Example: Creating a Compiler Optimization" on page 460. But before diving into any of those things, we will take a tour of the codebase of the Closure Compiler.

Tour of the Codebase

Chapter 12 provided a simple example of using the Closure Compiler via its programmatic API. This section will examine the classes from the `com.google.javascript.jscomp` package used in that example, as well as some others that will be necessary to follow the code samples that appear later in this chapter.

Getting and Building the Compiler

Although it is possible to explore the code by visiting *http://code.google.com/p/closure-compiler/source/browse/*, it will be much easier to poke around and experiment with the code if you have a local copy of the repository on your machine. As explained in "Closure Compiler" on page 12, it is possible to check out the code using Subversion and then build the code by running `ant jar`. Though if you use the Eclipse IDE (*http://www.eclipse.org*), it is even easier to work with the code using Eclipse with the Subclipse plugin for accessing Subversion repositories (*http://subclipse.tigris.org*). The Closure Compiler codebase contains `.project` and `.classpath` files that contain

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

pass. Although this example focuses on eliminating naming collisions with externs, disambiguation can also create new opportunities for other compiler passes as well, such as removing unused methods.

Note that there may be cases in which `disambiguateProperties` cannot determine which object a property refers to, such as the case where a subclass overrides a superclass method. We saw a similar issue in Chapter 13 with devirtualizing prototype methods because of how dynamic dispatch works:

```
/** @param {example.House} house */
example.sales.putHouseOnTheMarket = function(house) {
  // It is unclear whether this should become either of the following:
  // house.example_House_prototype$paint('blue');
  // house.example_Mansion_prototype$paint('blue');
  house.paint('blue');
};
```

A similar issue arises if there is a union type that includes types with the same property:

```
/**
 * Makes it rain Cat or Dog, but not both!
 * @param {example.Dog|example.Cat} pet
 */
example.makeItRain = function(pet) {
  // pet.example_Dog_prototype$getName or pet.example_Cat_prototype$getName?
  if (pet.getName() == 'Sven') {
    alert('It\'s raining Sven! Hallelujah!');
  }
};
```

If either of these cases occurs for a particular property, `disambiguateProperties` will ensure that the `getName()` methods of both `example.Cat` and `example.Dog` are renamed to the same thing.

ambiguateProperties

The `ambiguateProperties` option renames unrelated properties to the same name using type information. This helps reduce code size because more properties can be given short names, and helps improve gzip compression because the same names are more commonly used. When `ambiguateProperties` is used together with `disambiguateProperties` on the previous example, the result is:

```
var example = {};

example.a = function(a) {
  // dogName_ has been renamed to b
  this.b = a;
};

// getName has been renamed to c
example.a.prototype.c = function() {
  return this.b;
};
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
var a = [];
// We want this to result in a Compiler error.
if (a == !a) {
  alert('a should not be equal to its negation!');
}

var checkIfNullOrUndefined = function(b) {
  // We want this JSDoc to suppress the Compiler error.
  if (/** @suppress {double-equals} */ (b == null)) {
    alert('b is neither null nor undefined!');
  }
};
```

Before trying to write the Java code for this compiler pass, use `MyCommandLineRunner` from the previous section to generate the AST for `equals.js` with the following command:

```
# This has the side effect of producing ast.svg.
java -jar mycompiler.jar --js=equals.js
```

Figure 14-5 shows the resulting representation of the AST.

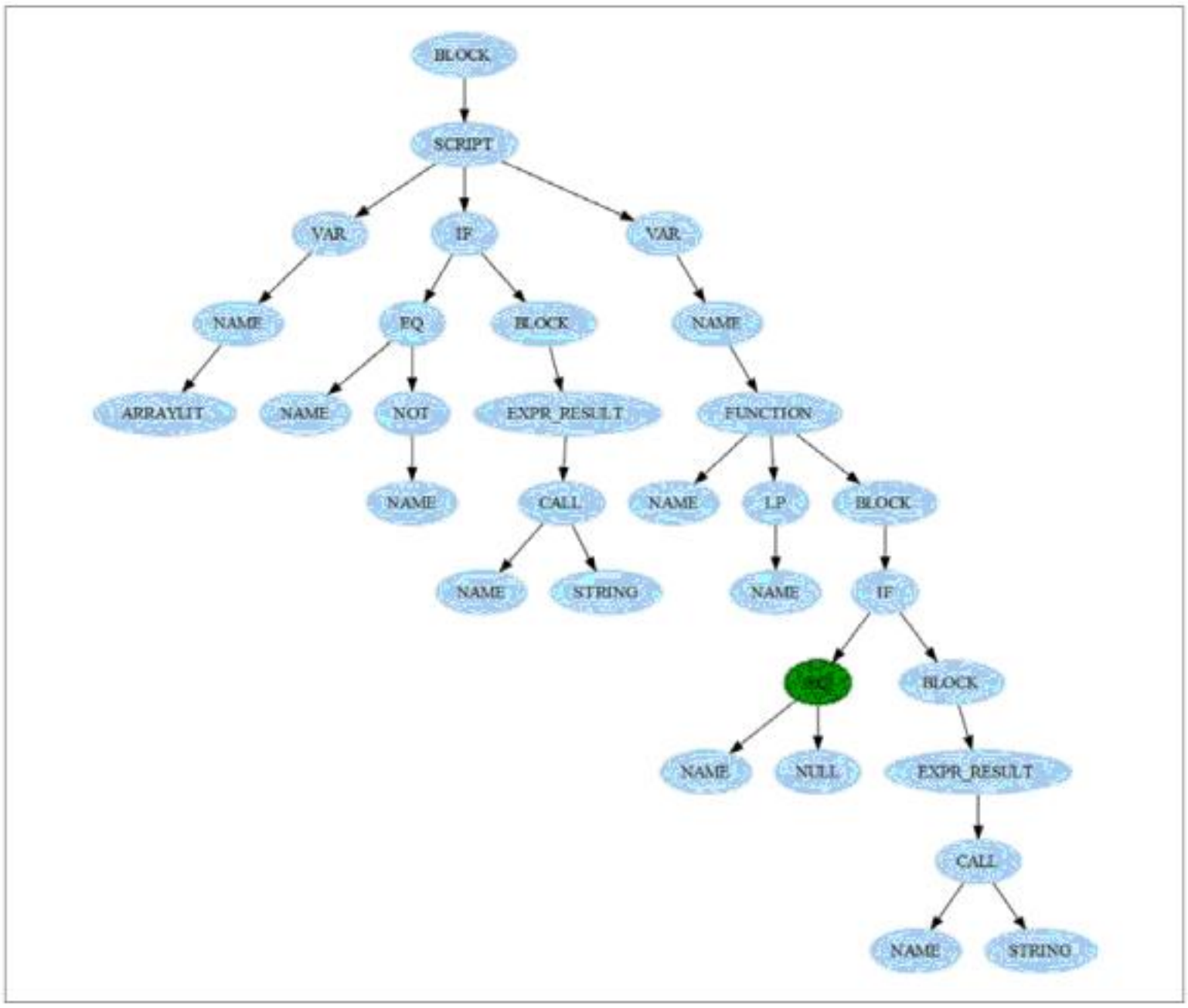


Figure 14-5. The AST for equals.js.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

As was the case for `CheckDoubleEquals`, most of the interesting work happens in an `AbstractPostOrderCallback`. Using the `Node` methods `getQualifiedName()` and `getType()`, the `visit()` method examines the `Node` to see whether it matches the structure of an `example.styles.addStylesheet()` call. Note how the argument to `addStylesheet()` is added only if it is a string literal—if it is a variable or the result of an expression, then the call should be left alone.

Each argument to `addStylesheet()` is added to a `LinkedHashSet` so that there are no duplicates, and the order in which `addStylesheet()` calls appear in the source code matches the order in which the CSS files will be concatenated. When a value is added to `cssFiles`, its corresponding call is removed from the AST, removing the call from the output, as desired. This pass can be integrated into `MyCommandLineRunner` just like the others, though it should be run after the checks are complete:

```
customPasses.put(CustomPassExecutionTime.BEFORE_OPTIMIZATIONS,
    new CollectCssPass(getCompiler(), "styles/", new File("styles.css")));
```

Once `mycompiler.jar` is recreated to include the new pass, then the source code can be compiled as follows:

```
java -jar mycompiler.jar --js=example/styles.js \
                         --js=example/component1.js \
                         --js=example/component2.js \
                         --js=example/main.js \
                         --formatting=PRETTY_PRINT > main-compiled.js
```

which produces a file named `main-compiled.js`:

```
var example = {};
example.styles = {};
example.styles.addStylesheet = function(b) {
  var a = document.createElement("link");
  a.rel = "stylesheet";
  a.type = "text/css";
  a.href = b;
  document.getElementsByTagName("head")[0].appendChild(a)
};example.component1 = {};example.component2 = {};
```

The compiled JavaScript file no longer has any calls to `example.styles.addStylesheet()`, as desired. If this code were compiled in Advanced mode, the declaration of `example.styles.addStylesheet` would also be removed because it is not used (it is not removed by `CollectCssPass` in case there are calls to it that do not pass a string literal).

Finally, the generated file `styles.css` is the concatenation of `styles1.css` and `styles2.css`, so the generated CSS and JavaScript can be used together in a web page as follows:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

using it to initialize `G_testRunner`. Instead of installing global functions, as `goog.testing.ContinuationTestCase` does, `goog.testing.AsyncTestCase` requires test functions to invoke methods on the instance of the test case, to indicate changes in control flow due to asynchronous logic.

The two methods of interest are `waitForAsync()` and `continueTesting()`. Invoking `waitForAsync()` indicates that the test should not end when the current thread of execution is reached. Instead, the test case waits for `continueTesting()` to be invoked in a new thread of execution, and resumes the test there. Like `goog.testing.ContinuationTestCase`, these calls can be chained, so if `waitForAsync()` is invoked after `continueTesting()` in the new thread of execution, the test will not end until `continueTesting()` is called again, as demonstrated in this example:

```
// This test passes.
var testDoubleContinuation = function() {
  // We use this technique again to determine that the innermost callback
  // scheduled with this test is called before the test ends.
  reachedFinalContinuation = false;

  // It is reasonable to invoke waitForAsync() either before or after the
  // asynchronous logic is scheduled with setTimeout().
  asyncTestCase.waitForAsync();

  setTimeout(function() {
    // Because this is a callback to setTimeout(), this code will run in a
    // different thread of execution than the initial call to
    // testDoubleContinuation().

    // This line is optional because the final call to continueTesting() will
    // be within one second of the original call to waitForAsync(); however,
    // the test may be easier to follow when calls to waitForAsync() and
    // continueTesting() complement one another.
    asyncTestCase.continueTesting();

    setTimeout(function() {
      // This should be the final thread of execution that this test runs, so
      // continueTesting() must be called here.
      asyncTestCase.continueTesting();

      // Once again, assertTrue(reachedFinalContinuation) is
      // called in tearDown() to affirm that this line of code was reached.
      reachedFinalContinuation = true;
    }, 300);

    // Here waitForAsync() is called after setTimeout(). The important thing is
    // that it is called before the current thread of execution ends.
    asyncTestCase.waitForAsync();
  }, 200);
};
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Drawbacks to the Functional Pattern

This section enumerates problems with using the functional pattern, particularly when writing JavaScript that will be compiled with the Closure Compiler.

Instances of Types Take Up More Memory

Every time `phone()` is called, two new functions are created (one per method of the type). Each time, the functions are basically the same, but they are bound to different values.

These functions are not cheap because each is a closure that maintains a reference for every named variable in the enclosing function in which the closure was defined. This may inadvertently prevent objects from being garbage collected, causing a memory leak. The Closure Library defines `goog.bind()` and `goog.partial()` in `base.js` to make it easier to create closures that maintain only the references they need, making it possible for other references to be removed when the enclosing function exits.

This is not a concern when `Phone()` is called because of how it takes advantage of prototype-based inheritance. Each method is defined once on `Phone.prototype` and is therefore available to every instance of `Phone`. This limits the number of function objects that are created and does not run the risk of leaking memory.

Methods Cannot Be Inlined

When possible, the Compiler will inline methods, such as simple getters. This can reduce code size as well as improve runtime performance. Because the methods in the functional pattern are often bound to variables that cannot be referenced externally, there is no way for the Compiler (as it exists today) to rewrite method calls in such a way that eliminates the method dispatch. By comparison, the following code snippet:

```
// phone1 and phone2 are of type Phone
var caller = phone1.getPhoneNumber();
var receiver = phone2.getPhoneNumber();
operator.createConnection(caller, receiver);
```

could be rewritten to the following by the Compiler:

```
operator.createConnection(caller.phoneNumber_, receiver.phoneNumber_);
```

Superclass Methods Cannot Be Renamed (Or Will Be Renamed Incorrectly)

When the Closure Compiler is cranked up to 11, one of the heuristics it uses for renaming is that any property that is not accessed via a quoted string is allowed to be renamed, and the Compiler will do its best to rename it. All quoted strings will be left alone. You are not required to use the Compiler with this aggressive setting, but the potential reduction in code size is too big to ignore.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

In the previous example, `map` does not map `foo` to `"foo"` and `bar` to `"bar"`. When `foo` and `bar` are used as keys for `map`, they are converted into strings using their respective `toString()` methods. This results in mapping the `toString()` of `foo` to `"foo"` and the `toString()` of `bar` to `"bar"`. Because both `foo.toString()` and `bar.toString()` are `"[object Object]"`, the previous code is equivalent to:

```
var map = new Object();
map["[object Object]"] = "foo";
map["[object Object]"] = "bar";

alert(map["[object Object]"]);
```

Therefore, `map[bar] = "bar"` replaces the mapping of `map[foo] = "foo"` on the previous line.

There Are Several Ways to Look Up a Value in an Object

There are several ways to look up a value in an object, so if you learned JavaScript by copying and pasting code from other websites, it may not be clear that the following code snippets are equivalent:

```
// (1) Look up value by name:
map.meaning_of_life;

// (2) Look up value by passing the key as a string:
map["meaning_of_life"];

// (3) Look up value by passing an object whose toString() method returns a
// string equivalent to the key:
var machine = new Object();
machine.toString = function() { return "meaning_of_life"; };
map[machine];
```

Note that the first approach, "Look up value by name," can be used only when the name is a valid JavaScript identifier. Consider the example from the previous section where the key was `"[object Object]"`:

```
alert(map.[object Object]); // throws a syntax error
```

This may lead you to believe that it is safer to always look up a value by passing a key as a string rather than by name. In Closure, this turns out not to be the case because of how variable renaming works in the Compiler. This is explained in more detail in Chapter 13.

Single-Quoted Strings and Double-Quoted Strings Are Equivalent

In some programming languages, such as Perl and PHP, double-quoted strings and single-quoted strings are interpreted differently. In JavaScript, both types of strings are

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
var Rectangle2 = function() {
  // This adds bindings to each new instance of Rectangle2 rather than adding
  // them once to Rectangle2.prototype.
  this.width = 3;
  this.height = 4;
};

var rect1 = new Rectangle();
var rect2 = new Rectangle2();

rect1.hasOwnProperty('width'); // evaluates to false
rect2.hasOwnProperty('width'); // evaluates to true

delete rect1.width;
delete rect2.width;

rect1.width; // evaluates to 3
rect2.width; // evaluates to undefined
```

Finally, note that the `__proto__` properties in the diagram are not set explicitly in the sample code. These relationships are managed behind the scenes by the JavaScript runtime.

The Syntax for Defining a Function Is Significant

There are two common ways to define a function in JavaScript:

```
// Function Statement
function FUNCTION_NAME() {
  /* FUNCTION_BODY */
}

// Function Expression
var FUNCTION_NAME = function() {
  /* FUNCTION_BODY */
};
```

Although the function statement is less to type and is commonly used by those new to JavaScript, the behavior of the function expression is more straightforward. (Despite this, the Google style guide advocates using the function expression, so Closure uses it in almost all cases.) The behavior of the two types of function definitions is not the same, as illustrated in the following examples:

```
function hereOrThere() {
  return 'here';
}

alert(hereOrThere()); // alerts 'there'

function hereOrThere() {
  return 'there';
}
```

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
  // Declaration of i which puts it in function scope.
  // The value 42 is never used.
  var i = 42;
};
```

Like the case in which redeclaring a variable goes unnoticed by the interpreter but is flagged by the Compiler, the Compiler will also issue a warning (again, with the Verbose warnings enabled) if a variable declaration appears after its first use within the function.

It should be noted that even though blocks do not introduce new scopes, functions can be used in place of blocks for that purpose. The `if` block in `getAllKeys()` could be rewritten as follows:

```
if (typeof value == 'object') {
  var functionWithNewScope = function() {
    // This is a new function, and therefore a new scope, so within this
    // function, map is a new variable because it is prefixed with 'var'.
    var map = value;

    // Because keys is not prefixed with 'var', the existing value of keys
    // from the enclosing function is used.
    keys = keys.concat(getAllKeys(map));
  };
  // Calling this function will have the desired effect of updating keys.
  functionWithNewScope();
}
```

Although this approach will work, it is less efficient than replacing `var map` with `var newMap` as described earlier.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

be toggled during development, it is possible to redefine them using query data, as explained in "Serve Command" on page 535.

The format for a config file is a superset of JSON that has limited support for comments. The following is the contents of `hello-config.js`, a plovr config file for the "Hello World" example from Chapter 1:

```
{
  "id": "hello-world",
  "inputs": "hello.js",
  "paths": "."
}
```

A config must have a unique id, which is specified by the `id` property, so in this case the id is `hello-world`. It must also have at least one input file to compile, which is specified by the `inputs` property, so the input for `hello-config.js` is `hello.js`. The `paths` property in the config identifies paths where potential dependencies can be found. In this way, both `inputs` and `paths` act the same as the respective `--input` and `--path` flags to `calcdeps.py`.

Both `inputs` and `paths` can have multiple values, so each may be an array of strings rather than a single string. Each value can identify either a file or a directory, and specifying a directory includes JavaScript and Soy files in its subdirectories. Recall from Chapter 1 that both `hello.js` and `hello.soy` are in the same directory, so this `hello-config.js` could also be defined as:

```
{
  "id": "hello-world",
  "inputs": "hello.js",
  "paths": ["hello.js", "hello.soy"]
}
```

Unlike the case where `calcdeps.py` is used, there is no need to include `../closure/library` or `../closure-templates/javascript/soyutils_usegoog.js` as values for `paths`. This is because `plovr.jar` includes the JavaScript source code for both the Closure Library and `soyutils_usegoog.js` and always includes them as potential dependencies.

To use your own version of the Closure Library instead of the one that is bundled with plovr, add a property named `closure-library` in the config that identifies your path to the Library as a string. This can be helpful if you are making changes to the Library, or if you want to be able to insert `debugger` statements into the Library's source code.

Finally, //-style comments may appear in a config file if they are on their own line. (This simplification makes them easier to strip before passing them to the JSON parser.) This means that `hello-config.js` could be documented as follows:

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The beginning of the warning includes a hyperlink to `http://localhost:9810/view?id=hello-world&name=hello.js#6`, which will display `hello.js` scrolled up to line 6 where the warning was found. This is shown in Figure C-2.

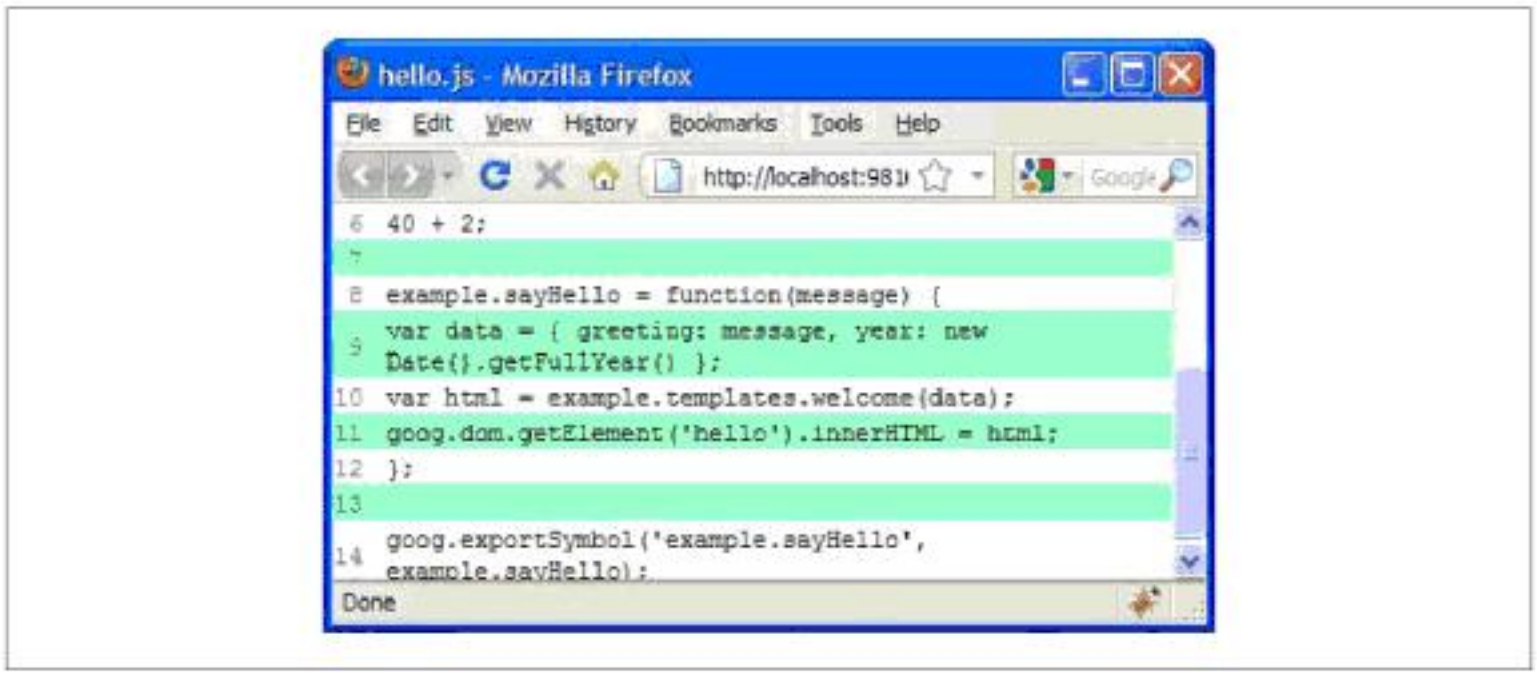


Figure C-2. plovr displaying the line where the Compiler warning was found.

Using plovr in this way can help identify problems in your code quickly, which helps speed up development. For this reason, it is a good idea to use Verbose warnings regardless of which compilation mode you are using. However, recall that in `RAW` mode, the Compiler is not used at all. This means that plovr will not report any warnings when `RAW` mode is used, regardless of the specified warning level.

Auditing Compiled Code Size

plovr makes it easy to see how good the Compiler is at reducing the size of your JavaScript. Visiting `http://localhost:9810/size?id=hello-world` will display a table that compares original size with compiled size for each file used to compile `hello.js`. As shown in Figure C-3, the compiled version of the Hello World example is only 22% of its original size.

The `/size` URL also honors the `mode` query parameter, so checking the effect that Advanced mode has on compiled code size is easily achieved by visiting `http://localhost:9810/size?id=hello-world&mode=advanced`. As shown in Figure C-4, the compiled version of the Hello World example is only 0.7% of its original size, as more than half of its inputs are completely removed from the compiled output!

Although all screenshots of plovr features are subject to change as plovr is improved, the UI for the `/size` URL will almost certainly change: the way percentages are currently displayed is misleading, and it would be more helpful if the contents of each compiled file could be examined.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

A

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

F

G

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

P

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

U

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Closure: The Definitive Guide

If you're ready to use Closure to build rich web applications with JavaScript, this hands-on guide has precisely what you need to learn this suite of tools in depth. Closure makes it easy for experienced JavaScript developers to write and maintain large and complex codebases—as Google has demonstrated by using Closure with Gmail, Google Docs, and Google Maps.

Author and Closure contributor Michael Bolin has included numerous code examples and best practices, as well as valuable information not available publicly until now. You'll learn all about Closure's Library, Compiler, Templates, testing framework, and Inspector—including how to minify JavaScript code with the Compiler, and why the combination of the Compiler and the Library is what sets Closure apart from other JavaScript toolkits.

- Learn how the Compiler significantly reduces the amount of JavaScript users have to download when visiting your site
- Discover several ways to use the Compiler as part of your build process
- Learn about type expressions, primitives, and common utilities
- Understand how Closure emulates classes and class-based inheritance
- Use Closure Templates on the server and the client from either JavaScript or Java
- Test and debug your JavaScript code, even when it's compiled

Previous programming experience is recommended.

"Closure: The Definitive Guide *is an incredible resource for both beginner and expert developers using Google Closure Tools. This book surprised me: it contains things I didn't know about before.*"

—Erik Arvidsson
co-creator of the Closure Library

Michael Bolin, a former Google engineer, worked on Google Calendar, Google Tasks, and the Closure Compiler. As a frontend developer, he used the Closure Tools suite on a daily basis and made several contributions to it. His last project at Google was to open-source the Closure Compiler.

O'REILLY®
oreilly.com

US $49.99 CAN $57.99
ISBN: 978-1-449-38187-5